Introduction

Rustemo is a LR/GLR parser generator for Rust (a.k.a. compiler-compiler).

Basically, this kind of tools, given a formal grammar of the language, produce a program that can transform unstructured text (a sequence of characters, or more generally a sequence of tokens) to a structured (tree-like or graph-like) form which is more amenable to further programmatical analysis.

One interesting feature of Rustemo is Abstract-Syntax Tree[1] (AST) auto-generation, i.e. based on a formal grammar of the language Rustemo will deduce Rust types and actions used to create and represent AST of your language. These types and actions can further be incrementally manually tuned to your likings. You can find more info in the section on the default builder.

Rustemo tries to provide sensible defaults but is made with a flexibility in mind. Thus, you can plug-in your own builder or/and lexer.

See the project README for features, aspirations and the road-map.

There are multiple alternatives to this project. Some of them are listed in the Similar projects section in the README. I advise you to check them also to be able to make an informed decision of what approach would suit you the best.

Installation and setup

Rustemo uses `cargo` for the project management. There are two crates of interest:

- `rustemo-compiler` - is a crate you will use during the development. This crate provides the compiler CLI `rcomp` . You will also used this crate as a build dependency if the compiler is called from the `build.rs` script. See the configuration chapter for more information.
- `rustemo` - is Rustemo runtime. This crate will be used by the generated parser. If you use the default string lexer you will need additional dependencies. See the calculator tutorial.

A usual workflow

To work with Rustemo a usual sequence of steps is as follows (after installing `rustemo-compiler` crate):

1. Write a grammar in a textual file with `.rustemo` extension. For example, a JSON grammar might look like this (see the examples directory):

```
Value: False | True | Null | Object | Array | JsonNumber |
JsonString;
Object: "{" Member*[Comma] "}";
Member: JsonString ":" Value;
Array: "[" Value*[Comma] "]";

terminals
False: 'false';
True: 'true';
Null: 'null';
Comma: ',';
JsonNumber: /-?\d+(\.\d+)?(e|E[-+]?\d+)?/;
JsonString: /"((\\")|[^"])*"/;

OBracket: '[';
CBracket: ']';
OBrace: '{';
CBrace: '}';
Colon: ':';
```

2. Run `rcomp` compiler (a binary installed from `rustemo-compiler` crate) with the given grammar to produce the parser code and optional builder actions (enabled by default).

3. Fix errors reported by `rcomp` if any and repeat from step 2 until there is no errors. See the chapter on handling errors.

4. Call `parse` method from the generated parser with the input you want to parse (see the calculator tutorial for the details).

Instead of calling `rcomp` you can setup `build.rs` script to generate the parser whenever you build your crate. You can find detailed instruction on how to call the parser from the `build.rs` script and use the generated modules in the configuration section.

Where to start?

The best place to start at the moment is the calculator tutorial with a reference to the grammar language section and other sections as needed.

Besides the tutorial and this docs, another source of information are integration tests. Tests are usually implemented by a grammar where Rustemo compiler is called from the build.rs script. The result of each test is persisted to a `.ast` (or `.err` if error is expected) file if it doesn't exist, or compared with the expected file if the file exists. Be sure to check these tests as they provide a good overview of the Rustemo possibilities.

Footnotes

[1] See the note in the section on default builders.

Grammar language

This section describe the grammar language, its syntax and semantics rules.

The Rustemo grammar specification language is based on BNF with syntactic sugar extensions which are optional and build on top of a pure BNF. Rustemo grammars are based on Context-Free Grammars (CFGs) and are written declaratively. This means you don't have to think about the parsing process like in e.g. PEGs. Ambiguities are dealt with explicitly (see the section on conflicts).

The structure of the grammar

Each grammar file consists of two parts:

- derivation/production rules,
- terminal definitions which are written after the keyword `terminals`.

Each derivation/production rule is of the form:

```
<symbol>: <expression> ;
```

where `<symbol>` is a grammar non-terminal and `<expression>` is one or more sequences of grammar symbol references separated by choice operator `|`.

For example:

```
Fields: Field | Fields "," Field;
```

Here `Fields` is a non-terminal grammar symbol and it is defined as either a single `Field` or, recursively, as `Fields` followed by a string terminal `,` and than by another `Field`. It is not given here but `Field` could also be defined as a non-terminal. For example:

```
Field: QuotedField | FieldContent;
```

Or it could be defined as a terminal in terminals section:

```
terminals
Field: /[A-Z]*/;
```

This terminal definition uses regular expression recognizer.

Terminals

Terminal symbols of the grammar define the fundamental or atomic elements of your language, tokens or lexemes (e.g. keywords, numbers).

Terminals are specified at the end of the grammar file, after production rules, following the keyword `terminals` .

Tokens are recognized from the input by a `lexer` component. Rustemo provides a string lexer out-of-the-box which enable lexing based on recognizers provided in the grammar. If more control is needed, or if non-textual context has been parsed a custom lexer must be provided. See the lexers section for more.

Each terminal definition is in the form:

```
<terminal name>: <recognizer>;
```

where `<recognizer>` can be omitted if custom lexer is used.

The default string lexer enables specification of two kinds of terminal recognizers:

- String recognizer
- Regex recognizer

String recognizer

String recognizer is defined as a plain string inside single or double quotes. For example, in a grammar rule:

```
MyRule: "start" OtherRule "end";
```

`"start"` and `"end"` will be terminals with string recognizers that match exactly the words `start` and `end` . In this example we have recognizers inlined in the grammar rule.

For each string recognizer you must provide its definition in the `terminals` section in order to define a terminal name.

```
terminals
Start: "start";
End: "end";
```

You can reference the terminal from the grammar rule, like:

```
MyRule: Start OtherRule End;
```

or use the same string recognizer inlined in the grammar rules, like we have seen before. It is your choice. Sometimes it is more readable to use string recognizers directly. But, anyway you must always declare the terminal in the `terminals` section for the sake of providing names which are used in the code of the generated parser.

Regular expression recognizer

Or regex recognizer for short is a regex pattern written inside slashes (`/.../`).

For example:

```
terminals
Number: /\d+/;
```

This rule defines terminal symbol `Number` which has a regex recognizer that will recognize one or more digits from the input.

> **Note**
>
> You cannot write regex recognizers inline like you can do with string recognizers. This constraint is introduced because regexes are not that easy to write and they don't add to readability so it is always better to reference regex terminal by name in grammar rules.

> **Warning**
>
> During regex construction a `^` prefix is added to the regex from the grammar to make sure that the content is matched at the current input position. This can be an issue if you use a pattern like `A|B` in your regex as it translates to `^A|B` which matches either `A` at the current position or `B` in the rest of the input. So, the workaround for now is to use `(A|B)` , i.e. always wrap alternative choices in parentheses.

Usual patterns

This section explains how some common grammar patterns can be written using just a plain Rustemo BNF-like notation. Afterwards we'll see some syntax sugar extensions which can be used to write these patterns in a more compact and readable form.

One or more

This pattern is used to match one or more things.

For example, `Sections` rule below will match one or more `Section`.

```
Sections: Section | Sections Section;
```

Notice the recursive definition of the rule. You can read this as

> `Sections` is either a single Section or `Sections` followed by a `Section`.

Note

Please note that you could do the same with this rule:

```
Sections: Section | Section Sections;
```

which will give you similar result but the resulting tree will be different. Notice the recursive reference is now at the end of the second production.

Previous example will reduce sections early and then add another section to it, thus the tree will be expanding to the left. The example in this note will collect all the sections and than start reducing from the end, thus building a tree expanding to the right. These are subtle differences that are important when you start writing your semantic actions. Most of the time you don't care so use the first version as it is more efficient in the context of the LR parsing.

Zero or more

This pattern is used to match zero or more things.

For example, `Sections` rule below will match zero or more `Section`.

```
Sections: Section | Sections Section | EMPTY;
```

Notice the addition of the `EMPTY` choice at the end. This means that matching nothing is a valid `Sections` non-terminal. Basically, this rule is the same as one-or-more except that matching nothing is also a valid solution.

Same note from the above applies here to.

Optional

When we want to match something optionally we can use this pattern:

```
OptHeader: Header | EMPTY;
```

In this example `OptHeader` is either a `Header` or nothing.

Syntactic sugar - BNF extensions

Previous section gives the overview of the basic BNF syntax. If you got to use various BNF extensions (like Kleene star) you might find writing patterns in the previous section awkward. Since some of the patterns are used frequently in the grammars (zero-or-more, one-or-more etc.) Rustemo provides syntactic sugar for this common idioms using a well known regular expression syntax.

Optional

Optional match can be specified using `?`. For example:

```
A: 'c'? B Num?;
B: 'b';

terminals

Tb: 'b';
Tc: 'c';
Num: /\d+/;
```

Here, we will recognize `B` which is optionally preceded with `c` and followed by `Num`.

Lets see what the parser will return optional inputs.

In this test:

```
#[test]
fn optional_1_1() {
    let result = Optional1Parser::new().parse("c b 1");
    output_cmp!(
        "src/sugar/optional/optional_1_1.ast",
        format!("{result:#?}")
    );
}
```

for input `c b 1` the `result` will be:

```
Ok(
    A {
        tc_opt: Some(
            Tc,
        ),
        b: Tb,
        num_opt: Some(
            "1",
        ),
    },
)
```

If we leave the number out and try to parse `c b`, the parse will succeed and the result will be:

```
Ok(
    A {
        tc_opt: Some(
            Tc,
        ),
        b: Tb,
        num_opt: None,
    },
)
```

Notice that returned type is `A` struct with fields `tc_opt` and `num_opt` of `Optional` type. These types are auto-generated based on the grammar. To learn more see section on AST types/actions code generation.

Note

Syntax equivalence for `optional` operator

```
S: B?;

terminals
B: "b";
```

is equivalent to:

```
S: BOpt;
BOpt: B | EMPTY;

terminals
B: "b";
```

Behind the scenes Rustemo will create `BOpt` rule. All syntactic sugar additions operate by creating additional rules in the grammar during parser compilation.

One or more

One-or-more match is specified using `+` operator.

For example:

```
A: 'c' B+ Ta;
B: Num;

terminals

Ta: 'a';
Tc: 'c';
Num: /\d+/;
```

After `c` we expect to see one or more `B` (which will match a number) and at the end we expect `a`.

Let's see what the parser will return for input `c 1 2 3 4 a`:

```
#[test]
fn one_or_more_2_2() {
    let result = OneOrMore2Parser::new().parse("c 1 2 3 4 a");
    output_cmp!(
        "src/sugar/one_or_more/one_or_more_2_2.ast",
        format!("{result:#?}")
    );
}
```

The result will be:

```
Ok(
    [
        "1",
        "2",
        "3",
        "4",
    ],
)
```

> **Note**
>
> We see in the previous example that default AST building actions will drop string matches as fixed content is not interesting for analysis and usually represent syntax noise which is needed only for performing correct parsing. Also, we see that one-or-more will be transformed to a `Vec` of matched values (using the `vec` annotation, see bellow). Of, course, this is just the default. You can change it to fit your needs. To learn more see the section on builders.

> **Note**
>
> Syntax equivalence for `one or more` :
>
> ```
> S: A+;
>
> terminals
> A: "a";
> ```
>
> is equivalent to:
>
> ```
> S: A1;
> @vec
> A1: A1 A | A;
>
> terminals
> A: "a";
> ```

Zero or more

Zero-or-more match is specified using `*` operator.

For example:

```
A: 'c' B* Ta;
B: Num;

terminals

Ta: 'a';
Tc: 'c';
Num: /\d+/;
```

This syntactic sugar is similar to `+` except that it doesn't require rule to match at least once. If there is no match, resulting sub-expression will be an empty list.

Let's see what the parser based on the given grammar will return for input `c 1 2 3 a`.

```
#[test]
fn zero_or_more_2_1() {
    let result = ZeroOrMore2Parser::new().parse("c 1 2 3 a");
    output_cmp!(
        "src/sugar/zero_or_more/zero_or_more_2_1.ast",
        format!("{result:#?}")
    );
}
```

The result will be:

```
Ok(
    Some(
        [
            "1",
            "2",
            "3",
        ],
    ),
)
```

But, contrary to one-or-more we may match zero times. For example, if input is `c a` we get:

```
Ok(
    None,
)
```

Note

Syntax equivalence for `zero or more`:

```
S: A*;

terminals
A: "a";
```

is equivalent to:

```
S: A0;
@vec
A0: A1 {nops} | EMPTY;
@vec
A1: A1 A | A;

terminals
A: "a";
```

So using of `*` creates both `A0` and `A1` rules. Action attached to `A0` returns a list of matched `a` and empty list if no match is found. Please note the usage of `nops`. In case `prefer_shift` strategy is used, using `nops` will perform both `REDUCE` and `SHIFT` during GLR parsing if what follows zero or more might be another element in the sequence. This is most of the time what you need.

Repetition modifiers

Repetitions (`+`, `*`, `?`) may optionally be followed by a modifier in square brackets. Currently, this modifier can only be used to define a separator. The separator is defined as a terminal rule reference.

For example, for this grammar:

```
A: 'c' B Num+[Comma];
B: 'b' | EMPTY;

terminals
Num: /\d+/;
Comma: ',';
Tb: 'b';
Tc: 'c';
```

We expect to see `c`, followed by optional `B`, followed by one or more numbers separated by a comma (`Num+[Comma]`).

If we give input `c b 1, 2, 3, 4` to the parser:

```
#[test]
fn one_or_more_1_1_sep() {
    let result = OneOrMore1SepParser::new().parse("c b 1, 2, 3, 4");
    output_cmp!(
        "src/sugar/one_or_more/one_or_more_1_1_sep.ast",
        format!("{result:#?}")
    );
}
```

we get this output:

```
Ok(
    A {
        b: Some(
            Tb,
        ),
        num1: [
            "1",
            "2",
            "3",
            "4",
        ],
    },
)
```

Note

Syntax equivalence of `one or more with separator` :

```
S: A+[Comma];

terminals
A: "a";
Comma: ",";
```

is equivalent to:

```
S: A1Comma;
@vec
A1Comma: A1Comma Comma A | A;

terminals
A: "a";
Comma: ",";
```

Making the name of the separator rule a suffix of the additional rule name makes sure that only one additional rule will be added to the grammar for all instances of `A+[Comma]` , i.e. same base rule with the same separator.

Parenthesized groups

Danger

This is not yet implemented.

You can use parenthesized groups at any place you can use a rule reference. For example:

```
S: a (b* a {left} | b);
terminals
a: "a";
b: "b";
```

Here, you can see that `S` will match `a` and then either `b* a` or `b` . You can also see that meta-data can be applied at a per-sequence level (in this case `{left}` applies to sequence `b* a`).

Here is a more complex example which uses repetitions, separators, assignments and nested groups.

```
S: (b c)*[comma];
S: (b c)*[comma] a=(a+ (b | c)*)+[comma];
terminals
a: "a";
b: "b";
c: "c";
comma: ",";

Syntax equivalence `parenthesized groups`:

    S: c (b* c {left} | b);
    terminals
    c: "c";
    b: "b";

is equivalent to:

    S: c S_g1;
    S_g1: b_0 c {left} | b;
    b_0: b_1 | EMPTY;
    b_1: b_1 b | b;
    terminals
    c: "c";
    b: "b";

So using parenthesized groups creates additional `_g<n>` rules
(`S_g1` in the
example), where `n` is a unique number per rule starting from `1`.
All other
syntactic sugar elements applied to groups behave as expected.
```

Greedy repetitions

> **Danger**
>
> This is not yet implemented.

`*`, `+`, and `?` operators have their greedy counterparts. To make an repetition operator greedy add `!` (e.g. `*!`, `+!`, and `?!`). These versions will consume as much as possible before proceeding. You can think of the greedy repetitions as a way to disambiguate a class of ambiguities which arises due to a sequence of rules where earlier constituent can match an input of various length leaving the rest to the next rule to consume.

Consider this example:

```
S: "a"* "a"*;
```

It is easy to see that this grammar is ambiguous, as for the input:

```
a a
```

We have 3 solutions:

```
1:S[0->3]
a_0[0->1]
    a_1[0->1]
    a[0->1, "a"]
a_0[2->3]
    a_1[2->3]
    a[2->3, "a"]
2:S[0->3]
a_0[0->0]
a_0[0->3]
    a_1[0->3]
    a_1[0->1]
        a[0->1, "a"]
    a[2->3, "a"]
3:S[0->3]
a_0[0->3]
    a_1[0->3]
    a_1[0->1]
        a[0->1, "a"]
    a[2->3, "a"]
a_0[3->3]
```

If we apply greedy zero-or-more to the first element of the sequence:

```
S: "a"*! "a"*;
```

We have only one solution where all `a` tokens are consumed by the first part of the rule:

```
S[0->3]
a_0[0->3]
    a_1[0->3]
    a_1[0->1]
        a[0->1, "a"]
    a[2->3, "a"]
a_0[3->3]
```

`EMPTY` built-in rule

There is a special `EMPTY` rule you can reference in your grammars. `EMPTY` rule will reduce without consuming any input and will always succeed, i.e. it is empty recognition.

Named matches (*assignments*)

In the section on builders you can see that struct fields deduced from rules, as well as generated semantic actions parameters, are named based on the `<name>=<rule reference>` part of the grammar. We call these `named matches` or `assignments`.

`Named matches` enable giving a name to a rule reference directly in the grammar.

In the calculator example:

```
E: left=E '+' right=E {Add, 1, left}
 | left=E '-' right=E {Sub, 1, left}
 | left=E '*' right=E {Mul, 2, left}
 | left=E '/' right=E {Div, 2, left}
 | base=E '^' exp=E {Pow, 3, right}
 | '(' E ')' {Paren}
 | Num {Num};

terminals

Plus: '+';
Sub: '-';
Mul: '*';
Div: '/';
Pow: '^';
LParen: '(';
RParen: ')';
Num: /\d+(\.\d+)?/;
```

we can see usage of assignments to name recursive references to `E` in the first four alternatives as `left` and `right` since we are defining binary operations, while the fifth alternative for power operation uses more descriptive names `base` and `exp`.

Now, with this in place, generated type for `E` and two operations (`/` and `^`), and the semantic action for `+` operation will be:

```rust
#[derive(Debug, Clone)]
pub struct Div {
    pub left: Box<E>,
    pub right: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Pow {
    pub base: Box<E>,
    pub exp: Box<E>,
}
#[derive(Debug, Clone)]
pub enum E {
    Add(Add),
    Sub(Sub),
    Mul(Mul),
    Div(Div),
    Pow(Pow),
    Paren(Box<E>),
    Num(Num),
}
pub fn e_add(_ctx: &Ctx, left: E, right: E) -> E {
    E::Add(Add {
        left: Box::new(left),
        right: Box::new(right),
    })
}
```

> **Note**
>
> This is just a snippet from the calculator example for the sake of brevity.

Notice the names of fields in `Div` and `Pow` structs. Also, the name of parameters in `e_add` action. They are derived from the assignments.

Without the usage of assignments, the same generated types and action would be:

```
#[derive(Debug, Clone)]
pub struct Div {
    pub e_1: Box<E>,
    pub e_3: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Pow {
    pub e_1: Box<E>,
    pub e_3: Box<E>,
}
#[derive(Debug, Clone)]
pub enum E {
    Add(Add),
    Sub(Sub),
    Mul(Mul),
    Div(Div),
    Pow(Pow),
    Paren(Box<E>),
    Num(Num),
}
pub fn e_add(_ctx: &Ctx, e_1: E, e_3: E) -> E {
    E::Add(Add {
        e_1: Box::new(e_1),
        e_3: Box::new(e_3),
    })
}
```

Where these names are based on the name of the referenced rule and the position inside the production.

Rule/production meta-data

Rules and productions may specify additional meta-data that can be used to guide parser construction decisions. Meta-data is specified inside curly braces right after the name of the rule, if it is a rule-level meta-data, or after the production body, if it is a production-level meta-data. If a meta-data is applied to the grammar rule it is in effect for all production of the rule, but if the same meta-data is defined for the production it takes precedence.

Note

See the example bellow.

Currently, kinds of meta-data used during parser construction are as follows:

- disambiguation rules
- production kinds

- user meta-data

Disambiguation rules

These are special meta-data that are used during by Rustemo during grammar compilation to influence decision on LR automata states' actions.

> **Note**
>
> See sections on parsing and resolving LR conflicts.

There are some difference on which rules can be specified on the production and terminal level.

Disambiguation rules are the following:

- *priority* - written as an integer number. Default priority is 10. Priority defined on productions have influence on both reductions on that production and shifts of tokens from that production. Priority defined on terminals influence the priority during tokenization. When multiple tokens can be recognized on the current location, those that have higher priority will be favored.

- *associativity* - `right` / `left` or `shift` / `reduce` . When there is a state where competing shift/reduce operations could be executed this meta-data will be used to disambiguate. These meta-data can be specified on both productions and terminals level. If during grammar analysis there is a state where associativity is defined on both production and terminal the terminal associativity takes precedence.

Note

```
    See the [calculator tutorial](./tutorials/calculator/calculator.md) for an
```

example of priority/associativity usage. There is also an example in the section on resolving LR conflicts.

- *global shift preference control* - `nops` and `nopse` . One of the standard techniques to resolve shift/reduce conflicts is to prefer shift always which yields a greedy behavior. This global settings can be altered during grammar compilation. `nops` (*no prefer shift*) can be used on a production level to disable this preference for the given production if enabled globally. `nopse` (*no prefer shift over empty*) is used to disable preferring shift over empty reductions only.

Production kinds

These meta-data are introduced to enable better deduction of function/parameter names in the generated code. The way to write the production kind is to write an identifier in camel-case.

For example:

```
E: E '+' E {Add, 1, left}
 | E '-' E {Sub, 1, left}
 | E '*' E {Mul, 2, left}
 | E '/' E {Div, 2, left}
 | Number;

terminals
Number: /\d+(\.\d+)?/;
Plus: '+';
Minus: '-';
Mul: '*';
Div: '/';
```

`Add`, `Sub`, `Mul` and `Div` are production kinds. These will influence the name of the parameters, fields etc. in the generated code.

See the section of improving AST in the calculator tutorial for more info.

User meta-data

Arbitrary meta-data can be attached to rules or productions. The form of each is `<name>: <value>` where `<name>` should be any valid Rust identifier while `<value>` can be any of:

- integer number
- float number
- string in double or single quotes
- keywords `true` or `false` for boolean values

These meta-data are supported syntactically but are not used at the moment. In the future semantic actions will have access to these values which could be used do alter building process in a user defined way.

Example

This test shows various meta-data applied at both rule and production level.

```rust
#[test]
fn productions_meta_data_inheritance() {
    let grammar: Grammar = r#"
        S {15, nopse}: A "some_term" B {5} | B {nops};
        A {bla: 10}: B {nopse, bla: 5} | B {7};
        B {left}: some_term {right} | some_term;
        terminals
        some_term: "some_term";
        "#
    .parse()
    .unwrap();
    assert_eq!(grammar.productions.len(), 7);

    assert_eq!(grammar.productions[ProdIndex(1)].prio, 5);
    // Inherited
    assert!(grammar.productions[ProdIndex(1)].nopse);
    assert_eq!(grammar.productions[ProdIndex(1)].meta.len(), 0);

    // Inherited
    assert_eq!(grammar.productions[ProdIndex(2)].prio, 15);
    assert!(grammar.productions[ProdIndex(2)].nops);
    // Inherited
    assert!(grammar.productions[ProdIndex(2)].nopse);

    assert_eq!(
        5u32,
        match grammar.productions[ProdIndex(3)].meta.get("bla").unwrap() {
            crate::lang::rustemo_actions::ConstVal::Int(i) => i.into(),
            _ => panic!(),
        }
    );
    assert_eq!(grammar.productions[ProdIndex(3)].meta.len(), 1);

    // Inherited
    assert_eq!(grammar.productions[ProdIndex(4)].prio, 7);
    assert_eq!(
        10u32,
        match grammar.productions[ProdIndex(4)].meta.get("bla").unwrap() {
            crate::lang::rustemo_actions::ConstVal::Int(i) => i.into(),
            _ => panic!(),
        }
    );

    assert_eq!(
        grammar.productions[ProdIndex(5)].assoc,
        Associativity::Right
    );

    // Inherited
    assert_eq!(grammar.productions[ProdIndex(6)].assoc, Associativity::Left);
}
```

Rule annotations

Rule annotation are written before grammar rule name using `@action_name` syntax. Annotations are special built-in meta-data used to change the generated AST types and/or actions.

Currently, there is only one annotation available - `vec`, which is used to annotate rules that represent zero-or-more or one-or-more patterns. When this annotation is applied the resulting AST type will be `Vec`. Automatically generated actions will take this into account if default builder is used (see the section on builders).

`vec` annotation is implicitly used in `*` and `+` syntax sugar. See the relevant sections for the equivalent grammars using the `vec` annotation.

For example, you can use `@vec` annotation in grammar rules that have the following patterns:

```
// This will be a vector of Bs. The vector may be empty.
@vec
A: A B | B | EMPTY;

// This is the same but the vector must have at least one element after
// a successful parse (and here we've changed the order in the first production)
@vec
A: B A | B;
```

This is just a convenience and a way to have a default type generated up-front. You can always change AST types manually.

Grammar comments

In Rustemo grammar, comments are available as both line comments and block comments:

```
// This is a line comment. Everything from the '//' to the end of line is a comment.

/*
  This is a block comment.
  Everything in between `/*`  and '*/' is a comment.
*/
```

Handling keywords in your language

Not implemented

This is currently not implemented.

By default parser will match given string recognizer even if it is part of some larger word, i.e. it will not require matching on the word boundary. This is not the desired behavior for language keywords.

For example, lets examine this little grammar:

```
S: "for" name=ID "=" from=INT "to" to=INT;

terminals
ID: /\w+/;
INT: /\d+/;
```

This grammar is intended to match statement like this one:

```
for a=10 to 20
```

But it will also match:

```
fora=10 to20
```

which is not what we wanted.

Rustemo allows the definition of a special terminal rule `KEYWORD`. This rule must define a regular expression recognizer. Any string recognizer in the grammar that can be also recognized by the `KEYWORD` recognizer is treated as a keyword and is changed during grammar construction to match only on word boundary.

For example:

```
S: "for" name=ID "=" from=INT "to" to=INT;

terminals
ID: /\w+/;
INT: /\d+/;
KEYWORD: /\w+/;
```

Now,

```
fora=10 to20
```

will not be recognized as the words `for` and `to` are recognized to be keywords (they can be matched by the `KEYWORD` rule).

This will be parsed correctly:

```
for a=10 to 20
```

As `=` is not matched by the `KEYWORD` rule and thus doesn't require to be separated from the surrounding tokens.

Note

```
Rustemo uses integrated scanner so this example:

    for for=10 to 20

will be correctly parsed. `for` in `for=10` will be recognized as `ID` and
not as a keyword `for`, i.e. there is no lexical ambiguity due to tokenizer
separation.
```

Handling whitespaces and comments (a.k.a Layout) in your language

The default string lexer skips whitespaces. You can take control over this process by defining a special grammar rule `Layout`. If this rule is found in the grammar the parser will use it to parse layout before each token. This is usually used to parse whitespaces, comments, or anything that is not relevant for the semantics analysis of the language.

For example, given the grammar:

```
// Digits with some words in between that should be ignored.
S: Digit TwoDigits Digit+;
TwoDigits: Digit Digit;
Layout: LayoutItem+;
LayoutItem: Word | WS;

terminals
Digit: /\d/;
Word: /[a-zA-Z]+/;
WS: /\s+/;
```

We can parse an input consisting of numbers and words but we will get only numbers in the output.

```
    let result = LayoutParser::new().parse("42 This6 should be 8 ignored 9 ");
```

If default AST builder is used, the result will be:

```
Ok(
    S {
        digit: "4",
        two_digits: TwoDigits {
            digit_1: "2",
            digit_2: "6",
        },
        digit1: [
            "8",
            "9",
        ],
    },
)
```

You can see that all layout is by default dropped from the result. Of course, you can change that by changing the generated actions. The layout is passed to each action through the `Context` object (`ctx.layout`).

For example, the generic tree builder preserves the layout on the tree nodes. The result from the above parse if generic tree builder is used will be:

```
Ok(
    NonTermNode {
        prod: S: Digit TwoDigits Digit1,
        location: [1,0-1,30],
        children: [
            TermNode {
                token: Digit("\"4\"" [1,0-1,1]),
                layout: None,
            },
            NonTermNode {
                prod: TwoDigits: Digit Digit,
                location: [1,1-1,8],
                children: [
                    TermNode {
                        token: Digit("\"2\"" [1,1-1,2]),
                        layout: None,
                    },
                    TermNode {
                        token: Digit("\"6\"" [1,7-1,8]),
                        layout: Some(
                            " This",
                        ),
                    },
                ],
                layout: None,
            },
            NonTermNode {
                prod: Digit1: Digit1 Digit,
                location: [1,19-1,30],
                children: [
                    NonTermNode {
                        prod: Digit1: Digit,
                        location: [1,19-1,20],
                        children: [
                            TermNode {
                                token: Digit("\"8\"" [1,19-1,20]),
                                layout: Some(
                                    " should be ",
                                ),
                            },
                        ],
                        layout: Some(
                            " should be ",
                        ),
                    },
                    TermNode {
                        token: Digit("\"9\"" [1,29-1,30]),
                        layout: Some(
                            " ignored ",
                        ),
                    },
                ],
                layout: Some(
                    " should be ",
                ),
            },
        ],
        layout: None,
    },
)
```

Here is another example that gives support for both line comments and block comments like the one used in the grammar language itself:

```
Layout: LayoutItem*;
LayoutItem: WS | Comment;
Comment: '/*' Corncs '*/' | CommentLine;
Corncs: Cornc*;
Cornc: Comment | NotComment | WS;

terminals
WS: /\s+/;
CommentLine: /\/\/.*/;
NotComment: /((\*[^\/])|[^\s*\/]|\/[^\*])+/;
```

Parsing

This section is a short refresher on the basics of parsing, and LR parsing in particular. For a more in-depth coverage I suggest the following books:

- Dick Grune, Ceriel J.H. Jacobs: Parsing Techniques: A Practical Guide, Springer Science & Business Media, ISBN 0387689540, 9780387689548. 2007.
- Aho, Alfred Vaino; Lam, Monica Sin-Ling; Sethi, Ravi; Ullman, Jeffrey David Compilers: Principles, Techniques, and Tools (2 ed.). Boston, Massachusetts, USA: Addison-Wesley. ISBN 0-321-48681-1. OCLC 70775643. 2006.

There are two main approaches to parsing:

- Top-down - in which the parser starts from the root non-terminal and tries to predict and match sub-expressions until reaching the most basic constituents (tokens). This approach is used by LL(k), parser combinators, packrat etc.
- Bottom-up - in which the parser starts from the most basic constituents (tokens) and tries to organize them into larger and larger structures until eventually reaches the root non-terminal. This approach is used by LR(k) variants (SLR, LALR, Canonical LR) and its generalized flavor GLR.

Historically, the most popular flavors of parser algorithms using both approaches were deterministic ones where the parser decides what to do by looking at fixed number of tokens ahead (usually just one) and never backtracks. These approaches were particularly popular due to their linear time complexity and guaranteed unambiguity, i.e. if the input is parsed successfully there can be exacly one representation/interpretation.

Top-down parsers are generally easier to debug and understand while bottom-up parsers are generally more powerful as the parser doesn't need to predict eagerly and commit in advance to what is ahead but can postpone that decision for later when it has collected more information.

LR parsing

LR parser uses a bottom-up approach. It takes tokens from the input and tries to organize them into bigger constructs. The process is based on a deterministic finite-state automata. Finite-state automata are machines that have a concept of a state, input and action. The machine makes a transition to another state based on the current state and the token it sees ahead (input). If for each state and each token ahead there is only one transition possible then we say that the FSA is deterministic (DFSA), if zero or multiple transitions are possible then we are dealing with a non-deterministic FSA.

LR parser uses DFSA. While it can be written manually, it would be a very tedious process and thus in practice we use compiler generators (aka compiler compilers) to automatically produce a program that represents the DFSA (and the rest of the LR parser) for the given grammar. Automata are usually represented by a table used during parsing to decide the action and transition to perform.

Given a sequence of tokens, if machine starts in a start state and end-up in an accepting state, we say that the sequence of tokens (a sentence) belongs to the language recognized by the DFSA. A set of languages which can be recognized by DFSA are called deterministic context-free languages - CFG. NFSA can recognize a full set of CFG. GLR parsing is based on NFSA.

Depending on the algorithm used to produce the FSA table we have different LR variants (SLR, LALR etc.). They only differ by the table they use, the parsing algorithm is the same.

Each state of the parser FSA is related to the grammar symbol and when a parser reaches a particular state we can tell that the last symbol it saw is the symbol related to the current state.

The parser also keeps a history of what it has seen so far on the parse stack. We can think of a content of the stack as a sequence of symbols the parser saw before reaching the current state.

At each step, based on the current state and the token ahead the parser can perform one of the following operations:

- **Shift** - where the parser takes the token ahead, puts it on the stack and transition to the next state according to the table,
- **Reduce** - where the parser takes zero or more symbols from the stack and replaces them by another higher-level symbol. For example, if we have states on the stack that represent symbols sequence `Expression + Expression`, the parser can take these three symbols from the top of the stack and replace them by `Add`. For this to happen, the grammar must have a production `Add: Expression + Expression`. Basically, reduction operation reduce a pattern from the right-hand side to the symbol on the left-hand side in the grammar production. A decision on which reduction to perform is specified by the FSA table.
- **Accept** - if the parser reaches accepting state while consuming all the tokens from the input and reducing the parse stack to the root symbol the parsing succeeded.
- **Error** - If there is no action that can be performed for the current state and the token ahead the parser reports an error.

Let's see this in practice on the Rustemo example.

We have a simple grammar that parses expression with just `+` operation.

```
Expression: Add | Num;
Add: Expression '+' Expression {left};
terminals
Num: /\d+/;
Plus: '+';
```

Note

For a full hands-on tutorial on building an expression language see the calculator tutorial.

If we parse the input `3 + 2 + 1` with the parser compiled in `debug` profile the log trace will have these messages at the position where the parser saw `3 + 2` and sees `+` ahead:

```
Context at 5[1,5]:
3 + 2--> + 1

Skipped ws: 1
Trying recognizers: [STOP, Plus]
    Recognizing STOP -- not recognized
    Recognizing Plus -- recognized
Token ahead: Plus("\"+\"" [1,6-1,7])
Stack: [
    State(0:AUG, 0..0 [0]),
    State(2:Expression, 0..1 [1,0-1,1]),
    State(4:Plus, 2..3 [1,2-1,3]),
    State(1:Num, 4..5 [1,4-1,5]),
]
```

The states on the stack give us information of what the parser saw previously.

The states represented in the log are from bottom to top. Each state has numeric identifier, the grammar symbol and the position in both absolute and line/column-based location.

Tip

State position information are handy as they tell you where in the input text you can find the reduced symbol. E.g. `Num` from the top of the stack above can be found at line 1, column 4, and it extends to line 1, column 5.

The `AUG` state is the start state of the parser.

At the current location, the parser saw an expression, a plus token and a number. The next thing that parser decides to do is to reduce by production `Expression: Num`, thus replacing the top symbol on the stack `Num` with `Expression`.

```
Reduce by production 'Expression: Num', size 1
GOTO 4:Plus -> 5:Expression
Stack: [
    State(0:AUG, 0..0 [0]),
    State(2:Expression, 0..1 [1,0-1,1]),
    State(4:Plus, 2..3 [1,2-1,3]),
    State(5:Expression, 4..5 [1,4-1,5]),
]
Current state: 5:Expression
```

Now, the parser is in the state `Expression` and the LR parsing table instructs the parser to reduce by production `Add: Expression Plus Expression` which takes the last three states/symbols from the stack and replace them with `Add`. Thus, currently the parser has seen an addition:

```
Reduce by production 'Add: Expression Plus Expression', size 3
GOTO 0:AUG -> 3:Add
Stack: [
    State(0:AUG, 0..0 [0]),
    State(3:Add, 0..5 [1,0-1,5]),
]
Current state: 3:Add
```

Another reduction is done by production `Expression: Add` replacing the `Add` state at the top of the stack with `Expression`. Thus, currently the parser has seen an expression:

```
Reduce by production 'Expression: Add', size 1
GOTO 0:AUG -> 2:Expression
Stack: [
    State(0:AUG, 0..0 [0]),
    State(2:Expression, 0..5 [1,0-1,5]),
]
Current state: 2:Expression
```

Since no further reductions are possible the parser shifts the next token to the top of the stack:

```
Shifting to state 4:Plus at location [1,6-1,7] with token Plus("\"+\"" [1,6-1,7])
Context at 7[1,7]:
3 + 2 +--> 1

Skipped ws: 1
Trying recognizers: [Num]
    Recognizing Num -- recognized '1'
Token ahead: Num("\"1\"" [1,8-1,9])
Stack: [
    State(0:AUG, 0..0 [0]),
    State(2:Expression, 0..5 [1,0-1,5]),
    State(4:Plus, 6..7 [1,6-1,7]),
]
Current state: 4:Plus
```

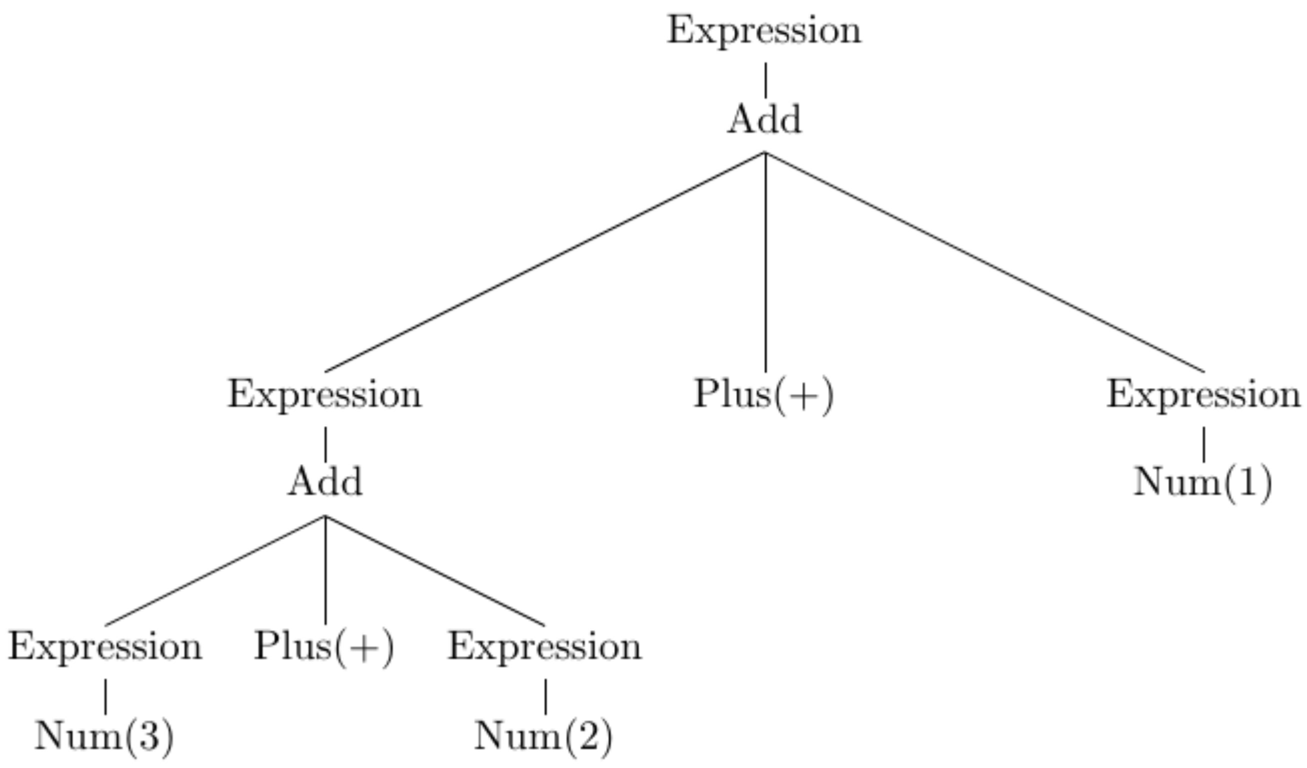
The parser repeats the process until it reduce the whole input to the root grammar symbol - `Expression`. At the end we can see:

```
Reduce by production 'Expression: Add', size 1
GOTO 0:AUG -> 2:Expression
Stack: [
    State(0:AUG, 0..0 [0]),
    State(2:Expression, 0..9 [1,0-1,9]),
]
Current state: 2:Expression
Accept
```

A tree-based interpretation

A usual interpretation of shift/reduce operations is building the parse tree. The content of the parse stack is then interpreted as a sequence of sub-trees where shifted token is a terminal node (a leaf of the tree) and each reduced symbol is a non-terminal node (a non-leaf node).

For the input `3 + 2 + 1` and the grammar above the tree is:



Leaf nodes (e.g. `Plus(+)`, `Num(3)`) are created by shift operations. Non-leaf nodes (e.g. `Add`, `Expression`) are created by reducing their child nodes.

It is now easy to see that the tree has been built bottom-up, starting from leaf terminal and reducing up to the root symbol of the grammar. Thus we say that the LR parsing is bottom-up.

Building outputs

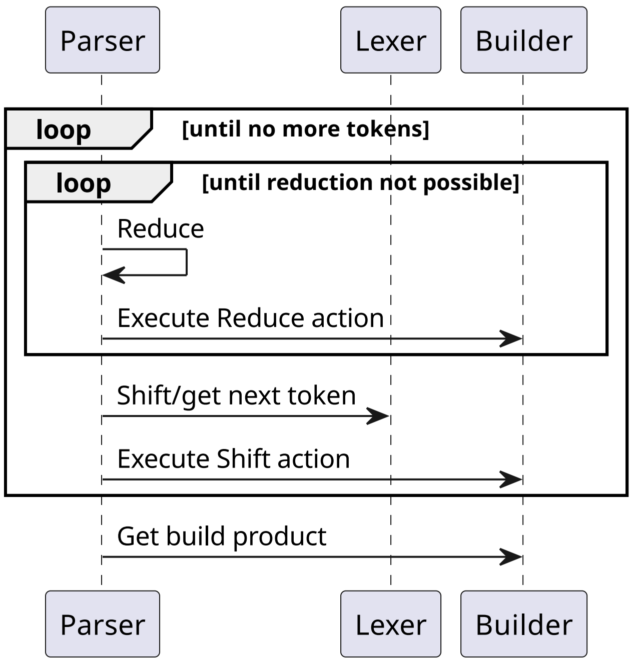
During shift/reduce operation, LR parser usually execute what we call *semantic actions* which job is to build the parsing output. In Rustemo, these actions are passed to a component called *builder* which is in charge of building the final result.

The default builder will build the Abstract-Syntax Tree (AST) with Rust types inferred from the grammar. There is also the generic tree builder. And finally, the user can supply a custom builder. For the details see the section on builders.

Rustemo parsing overview

The diagram bellow depict the high-level overview of the parsing process in Rustemo:

- LR parser works in a loop.
- If there is no lookahead token the parser asks the lexer to recognize the next token.
- The parser do all possible reductions. For each reduction the parser calls the builder.
- The parser performs shift operation, calls the builder and asks the lexer to recognize the next token.
- The process continues until we reach the end of the input and the parser gets to the accept state. Otherwise we report an error.



GLR parsing

GLR is a generalized version of LR which accepts a full set of CFG. If multiple parse actions can be executed at the current parser state the parser will split and investigate each possibility. If some of the path prove wrong it would be discarded (we call these splits - local ambiguities) but if multiple paths lead to the successful parse then all interpretations are valid and instead of the parse tree we get the parse forest. In that case we say that our language is ambiguous.

Due to the fact that automata handled by GLR can be non-deterministic we say that GLR is a form of non-deterministic parsing. See more in the section on resolving LR conflicts.

Rustemo uses a particular implementation of the GLR called Right-Nulled GLR which is a correct and more efficient version of the original GLR.

Configuration

The parser generator configuration is exposed through API calls which can be imported from the root of the `rustemo_compiler` crate:

- `pub fn process_crate_dir()` - recursivelly visit all the grammars starting from the crate root directory and generate the parsers and actions.
- `pub fn process_dir<P: AsRef<Path>>(dir: P)` - recursivelly visit all the grammars starting from the given directory and generate the parsers and actions.
- `pub fn process_grammar<P: AsRef<Path>>(grammar: P)` - generate the parser and actions for the given grammar.
- `Settings::new()` - returns a default `Settings` which can be further configured using chained calls.

These are meant to be called from the `build.rs` script. The most basic usage would be:

```
use std::process::exit;

fn main() {
    let mut settings = rustemo_compiler::Settings::new();
    if std::env::var("CARGO_FEATURE_ARRAYS").is_ok() {
        settings = settings
            .generator_table_type(rustemo_compiler::GeneratorTableType::Arrays);
    }

    if let Err(e) = settings.process_dir() {
        eprintln!("{}", e);
        exit(1);
    }
}
```

> **Note**
>
> Don't forget to add `rustemo-compiler` to the `build-dependencies` section of the `Cargo.toml` file.

In this example we are using the default settings and run the recursive processing of the project dir as determined by the cargo `CARGO_MANIFEST_DIR` environment variable. By default, it will generate both parser and actions in the cargo output dir as determined by `OUT_DIR` env variable.

You can change the default output to be the source tree, i.e. to generate parser/actions next to the grammar file by calling:

```
rustemo_compiler::Settings::new().in_source_tree().process_dir()
```

This will create default settings, change settings to generate the parser in the source tree and then process the current project directory.

It is usually discouraged to modify the source tree from the `build.rs` script but in the case of actions it is usually necessary as the actions are generated and manually maintained. To generate just the actions in the source tree while keeping the parser in the output directory you can do the following:

```
rustemo_compiler::Settings::new().actions_in_source_tree().process_dir()
```

Note

When running rustemo from `build.rs` your crate must have a build dependency to `rustemo-compiler`. If you don't want this than you can always resort to building your parser using rustemo CLI. Just don't forget to manually regenerate the parser and commit changes to your version control system when you change the grammar.

Note

For the full docs for settings provided by Rustemo see the crate docs.

Using generated modules

When the parser/actions is generated in the source tree you use it as any other Rust module but when any of them is generated in the output you can't include them in the module tree as they are generated in the output dir.

To be able to include modules from the output dirs you use `rustemo_mod!` macro:

```
use rustemo::rustemo_mod;
rustemo_mod! {pub(crate) rustemo, "/src/lang"}
```

or

```
use rustemo::rustemo_mod;
rustemo_mod!(generic_tree, "/src/builder/generic_tree");
```

This macro accepts two parameters. The first is a usual syntax used with `mod` (attributes, visibility and the name of the module) while the second parameter is the path to the module/grammar directory from the project root. This second parameter is needed for the macro to be able to calculate the full path in the output directory.

Components

This Chapter describes the main building block of every Rustemo parser, namely lexer, parser, and builder.

Lexers

Lexers are components that break an input sequence into its constituent parts called "tokens" or "lexemes". Rustemo implements context-aware lexing. Given the current parsing context lexers return the next possible tokens from the input stream. This gives a greater recognition strength as the lexing is done just for the tokens expected in a given parser state.

Generally, multiple tokens can match at a given location in which case we say that we have a lexical ambiguity. Rustemo has a built-in mechanism for lexical disambiguation.

To abstract over all possible inputs, Rustemo provides a trait `lexer::Input` which must be implemented by any type that can be used as an input to the parsing process. Types `str` and `[u8]` implement this trait and thus parsing a string or a sequence of bytes is possible out-of-the-box.

String parsing is facilitated by so-called recognizers, basically a string and regex patterns defined for each terminal in the `terminals` section of the grammar. Recognizers are used to configure the default string lexer.

For parsing other types you can provide your custom lexer.

Custom lexers

To create the custom lexer implement trait `rustemo::lexer::Lexer` for your type. Method `next_token` should return a result holding the next token given the current parsing context.

The lexer type should be defined in a file `<grammar name)_lexer.rs` where `<grammar name>` is the base name of the grammar file.

In the tests project a custom_lexer test is given which demonstrates implementation of a custom lexer. The lexer implemented in this test is used to tokenize a sequence of varints which are variable-width integers used in Google's protocol buffers wire format.

So, our lexer has to figure out what is the next varint in the sequence of bytes.

There are more than one approach to handle this. Here is one (see the test for additional information).

We start with the following grammar in a file, say `custom_lexer_2.rustemo`:

```
VarInts: VarInt+;
VarInt: MSBByte* NonMSBByte;

terminals
MSBByte:;
NonMSBByte:;
```

We have defined that the input consists of one or more varint and that each varint contains zero or more `MSBByte` (a byte whose highest bit is set) and exactly one `NonMSBByte` (a byte whose highest bit is not set) at the end.

And in file `custom_lexer_2_lexer.rs` we specify our lexer:

```rust
/// We are parsing a slice of bytes.
pub type Input = [u8];
pub type Ctx<'i> = LRContext<'i, Input, State, TokenKind>;

pub struct MyCustomLexer2();

impl MyCustomLexer2 {
    pub fn new() -> Self {
        MyCustomLexer2()
    }
}

/// In this custom lexer we are not recognizing a full VarInts but only its
/// constituents: MSBByte (if highest bit is set), NonMSBByte (highest bit is
/// not set). How these bytes is organized into VarInts is defined by the
/// grammar and the transformation to a numeric value is done in actions.
impl<'i> Lexer<'i, Ctx<'i>, State, TokenKind> for MyCustomLexer2 {
    type Input = Input;

    fn next_tokens(
        &self,
        context: &mut Ctx<'i>,
        input: &'i Self::Input,
        _token_kinds: Vec<(TokenKind, bool)>,
    ) -> Box<dyn Iterator<Item = Token<'i, Self::Input, TokenKind>> + 'i> {
        let value;
        let kind: TokenKind;
        if context.position() >= input.len() {
            value = &[][..];
            kind = TokenKind::STOP;
        } else {
            value = &input[context.position()..=context.position()];
            if value[0] & 0b1000_0000 != 0 {
                kind = TokenKind::MSBByte;
            } else {
                kind = TokenKind::NonMSBByte;
            };
        }

        Box::new(iter::once(Token {
            kind,
            value,
            location: Location {
                start: Position::Position(context.position()),
                end: Some(Position::Position(context.position())),
            },
        }))
    }
}
```

The job of the lexer in this case is to return `STOP` if at the end of the input or to return a token of `MSBByte` kind containing a single byte at the current position if the highest bit is set or `NonMSBByte` otherwise.

The transformation of a sequence of `(Non)MSBByte` to varint is done in semantic actions given in the file `custom_lexer_2_actions.rs`. The relevant (non-generated) part is this:

```
/// We are doing a conversion in this action. Other actions are generated.
/// msbbyte0 is an option containing first bytes of the VarInt non_msbbyte
/// contains the last byte
pub type VarInt = i128;
pub fn var_int_c1(
    _ctx: &Context,
    msbbyte0: MSBByte0,
    non_msbbyte: NonMSBByte,
) -> VarInt {
    let mut res: i128 = non_msbbyte as i128;
    if let Some(bytes) = msbbyte0 {
        bytes
            .iter()
            .rev()
            .for_each(|b| {
                res <<= 7;
                res |= (b & 0b0111_1111) as i128;
            });
    }
    res
}
```

Next, we configure Rustemo to generate the parser which use custom lexer. We do this either through the API by calling `lexer_type(LexerType::Custom)` (see the `build.rs` file in the tests project), or using `--lexer-type custom` if the parser is generated by the `rcomp` tool.

Finally, we need to configure generated parser with our lexer as follows:

```
    let bytes = std::fs::read(bytes_file).unwrap();
    let result = CustomLexer2Parser::new(MyCustomLexer2::new()).parse(&bytes);
```

Notice the instance of the lexer passed into `new` method during parser construction.

In this example, lexer doesn't need to know what tokens are expected at the current location. But, sometimes we need that information to make a more powerful context aware lexing. To do that you can import `LEXER_DEFINITION` value from the generated parser which implements `LexerDefinition` trait which has `recognizers` method accepting the current LR state (you can find it in the passed in context) and resulting in a iterator of token recognizers which provide information of expected tokens at the location.

For more info see the full test for custom lexers.

Note

Each lexer accepts a vector of tokens expected at a given location. This vector elements are actually pairs of `(token_kind, finish flag)` (e.g. `_token_kinds: Vec<(TokenKind, bool)>` above). Finish flag signifies that if the token is found lexer should not return any other token, i.e. lexing should terminate on a first succesful match where finish flag is true.

Lexical disambiguation

When a lexical ambiguity is encountered during the parsing process Rustemo will use several disambiguation strategies. Some of these strategies can be controlled by the configuration parameters. The job of all these strategies is to reduce a set of possible tokens at the given location.

The following table give and overview of the strategies, their defaults and possibility of control.

Strategy	Default LR	Default GLR	Can be disabled in LR	Can dis. GLR	Impl. In
Priorities	yes	yes	no	no	lexer
Most specific	yes	yes	yes	yes	lexer
Longest match	yes	yes	yes	yes	parser
Gram. order	yes	no	no	yes	parser

Strategies are applied in the following order:

- *Priorities* - expected tokens are sorted by priority. A first match in a priority group will reduce further matches only on that group. You can specify the priority of a terminal inside curly braces. The default priority is 10. For example:

  ```
  terminals
  SomeTerminal: 'some match' {15};
  ```

- *Most specific match* - string matches take precedence over regex matches. String matches are ordered by length. When the first string match succeeds no further matches are tried. This strategy is implemented in `StringLexer`.

- *Longest match* - All possible matches based on previous strategies are found and the longest match is used. There still can be multiple matches with the same length. A further disambiguation will be handled by the next strategy.

- *Grammar order* - Matches are tried in the grammar order. First match wins.

Since LR can't handle ambiguity, grammar order is a final resolution strategy which always resolve to a single token. For GLR this strategy is not enabled by default as we usually want to handle lexical ambiguity by using the GLR machinery.

Parsers

Parsers use tokens from lexer as inputs and recognize syntactic elements. Then, they call a builder to produce the final output.

There are two flavours of parsers supported by Rustemo:

- Deterministic LR
- Non-deterministic GLR, or more precise Right-Nulled GLR

Tip

GLR parsing is more complex as it must handle all possibilities so there is some overhead and LR parsing is generally faster. Thus, use GLR only if you know that you need it or in the early development process when you want to deal with SHIFT/REDUCE conflicts later.

Another benefit of LR parsing is that it is deterministic and non-ambiguous. If the input can be parsed there is only one possible way to do it with LR.

The API for both flavours is similar. You create an instance of the generated parser type and call either `parse` or `parse_file` where the first method accepts the input directly while the second method accepts the path to the file that needs to be parsed.

For example, in the calculator tutorial, we create a new parser instance and call `parse` to parse the input supplied by the user on the stdin:

```rust
fn main() {
    let mut expression = String::new();

    // Read the line from the input
    println!("Expression:");
    io::stdin()
        .read_line(&mut expression)
        .expect("Failed to read line.");

    // Parse the line and get the result.
    let result = CalculatorParser::new().parse(&expression);

    // Print the result using Debug formatter.
    println!("{:#?}", result);
}
```

The parser type `CalculatorParser` is generated by Rustemo from grammar `calculator.rustemo`.

The result of the parsing process is a `Result` value which contains either the result of parsing if successful, in the `Ok` variant, or the error value in `Err` variant.

If deterministic parsing is used the result will be the final output constructed by the configured builder.

For GLR the result will be `Forest` which contains all the possible trees/solution for the given input. For the final output you have to choose the tree and call the builder over it.

To generate GLR parser either set the algorithm using settings API (e.g. from `build.rs` script):

```
rustemo_compiler::Settings::new().parser_algo(ParserAlgo::GLR).process_dir()
```

or call `rcomp` CLI with `--parser-algo glr` over your grammar file.

For example of calling GLR parser see this test:

```
#[test]
fn glr_extract_tree_from_forest() {
    let forest = CalcParser::new().parse("1 + 4 * 9 + 3 * 2 + 7").unwrap();
    output_cmp!(
        "src/glr/forest/forest_tree_first.ast",
        format!("{:#?}", forest.get_first_tree().unwrap())
    );
    output_cmp!(
        "src/glr/forest/forest_tree_17.ast",
        format!("{:#?}", forest.get_tree(17).unwrap())
    );
    output_cmp!(
        "src/glr/forest/forest_tree_last.ast",
        format!("{:#?}", forest.get_tree(41).unwrap())
    );

    // Accessing a tree past the last.
    assert!(forest.get_tree(42).is_none());

    let tree = forest.get_tree(41).unwrap();
    output_cmp!(
        "src/glr/forest/forest_tree_children.ast",
        format!("{:#?}", tree.children()[0].children())
    );
}
```

The most useful API calls for `Forest` are `get_tree` and `get_first_tree`. There is also `solutions` which gives your the number of trees in the forest.

`Forest` supports `into_iter()` and `iter()` so it can be used in the context of a for loop.

```rust
#[test]
fn glr_forest_into_iter() {
    let forest = CalcParser::new().parse("1 + 4 * 9 + 3 * 2 + 7").unwrap();
    let mut forest_get_tree_string = String::new();
    let mut forest_iter_string = String::new();

    for tree_idx in 0..forest.solutions() {
        forest_get_tree_string
            .push_str(&format!("{:#?}", forest.get_tree(tree_idx).unwrap()))
    }

    for tree in forest {
        forest_iter_string.push_str(&format!("{tree:#?}"));
    }
    assert_eq!(forest_get_tree_string, forest_iter_string);
    output_cmp!("src/glr/forest/forest_into_iter.ast", forest_iter_string);
}

#[test]
fn glr_forest_iter() {
    let forest = CalcParser::new().parse("1 + 4 * 9 + 3 * 2 + 7").unwrap();
    let mut forest_get_tree_string = String::new();
    let mut forest_iter_string = String::new();
    let mut forest_iter_ref_string = String::new();

    for tree_idx in 0..forest.solutions() {
        forest_get_tree_string
            .push_str(&format!("{:#?}", forest.get_tree(tree_idx).unwrap()))
    }

    for tree in forest.iter() {
        forest_iter_string.push_str(&format!("{tree:#?}"));
    }

    for tree in &forest {
        forest_iter_ref_string.push_str(&format!("{tree:#?}"));
    }
    assert_eq!(forest_get_tree_string, forest_iter_string);
    assert_eq!(forest_get_tree_string, forest_iter_ref_string);
    output_cmp!("src/glr/forest/forest_iter.ast", forest_iter_string);
}
```

A tree can accept a builder using the `build` method. For an example of calling the default builder over the forest tree see this test:

```rust
#[test]
fn glr_tree_build_default() {
    let forest = CalcParser::new().parse("1 + 4 * 9").unwrap();
    assert_eq!(forest.solutions(), 2);

    let mut builder = calc::DefaultBuilder::new();
    output_cmp!(
        "src/glr/build/tree_build_default_1.ast",
        format!(
            "{:#?}",
            forest.get_first_tree().unwrap().build(&mut builder)
        )
    );
    output_cmp!(
        "src/glr/build/tree_build_default_2.ast",
        format!("{:#?}", forest.get_tree(1).unwrap().build(&mut builder))
    );
}
```

Builders

A builder is a component that is called by the parser during the parsing process to constructs the output.

Currently Rustemo can be configured with three builder types:

- **The default builder**

  When default builder is used, Rustemo will perform type inference for Abstract-Syntax Tree (AST) node types based on the grammar. The builder, AST types, and actions for creating instances of AST nodes will be generated.

- **Generic tree builder**

  This builder builds a tree where each node is of `TreeNode` type.

- **Custom builder**

  Is provided by the user.

Default builder

For default builder Rustemo will generate default AST types and actions following a certain set of rules explained in this section. The generated builder will then call into actions to produce instances of the AST types. The final output of this builder is AST tailored for the given grammar.

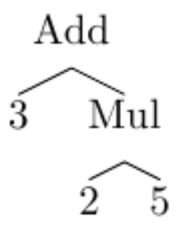
The builder will be generated, together with the parser, inside `<lang>.rs` file where `<lang>` is the name of the grammar. The actions will be generated into `<lang>_actions.rs` file.

> **Note**
>
> There are two approaches where generated files are stored. See the configuration section.

Note

Abstract-Syntax Tree (AST) representation of the input is different from a *Concrete-Syntax Tree* (CST, aka *the Parse Tree*). AST represents the essence of the parsed input without the concrete syntax information.

For example, `3 + 2 * 5` represents an algebric expression where we multiply `2` and `5` and then add product to `3`. AST should be the same no matter what is the concrete syntax used to write down this information.



We could write the same expression in the post-fix (Reverse Polish) notation like `3 2 5 * +`. CST would be different but the AST would be the same.

AST type inference

Based on grammar rules Rustemo will infer and generate AST node types which you can modify afterwards to suit your needs, so you can quickly have a working parser and tune it later.

The inference is done by the following rules:

Each non-terminal grammar rule will be of an `enum` type. The enum variants will be:

1. If only non-content matches are in the production (e.g. just string matches) -> plain variant without inner content
2. A single content match (e.g. regex match) or rule reference without assignment -> variant with referred type as its content.
3. Multiple content matches and/or assignments -> a `struct` type where fields types are of the referred symbols.

In addition, production kinds and assignments LHS names are used for fields/function/type naming. Also, cycle refs are broken using `Box`.

Probably the best way to explain is by using an example. For example, if we have the following grammar:

```
E: left=E '+' right=E {Add, 1, left}
 | left=E '-' right=E {Sub, 1, left}
 | left=E '*' right=E {Mul, 2, left}
 | left=E '/' right=E {Div, 2, left}
 | base=E '^' exp=E {Pow, 3, right}
 | '(' E ')' {Paren}
 | Num {Num};

terminals

Plus: '+';
Sub: '-';
Mul: '*';
Div: '/';
Pow: '^';
LParen: '(';
RParen: ')';
Num: /\d+(\.\d+)?/;
```

we get these generated actions/types:

```rust
use super::calculator04_ambig_lhs::{Context, TokenKind};
/// This file is maintained by rustemo but can be modified manually.
/// All manual changes will be preserved except non-doc comments.
use rustemo::Token as BaseToken;
pub type Input = str;
pub type Ctx<'i> = Context<'i, Input>;
#[allow(dead_code)]
pub type Token<'i> = BaseToken<'i, Input, TokenKind>;
pub type Num = String;
pub fn num(_ctx: &Ctx, token: Token) -> Num {
    token.value.into()
}
#[derive(Debug, Clone)]
pub struct Add {
    pub left: Box<E>,
    pub right: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Sub {
    pub left: Box<E>,
    pub right: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Mul {
    pub left: Box<E>,
    pub right: Box<E>,
}
/// ANCHOR: named-matches
#[derive(Debug, Clone)]
pub struct Div {
    pub left: Box<E>,
    pub right: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Pow {
    pub base: Box<E>,
    pub exp: Box<E>,
}
#[derive(Debug, Clone)]
pub enum E {
    Add(Add),
    Sub(Sub),
    Mul(Mul),
    Div(Div),
    Pow(Pow),
    Paren(Box<E>),
    Num(Num),
}
pub fn e_add(_ctx: &Ctx, left: E, right: E) -> E {
    E::Add(Add {
        left: Box::new(left),
        right: Box::new(right),
    })
}
/// ANCHOR_END: named-matches
```

```
pub fn e_sub(_ctx: &Ctx, left: E, right: E) -> E {
    E::Sub(Sub {
        left: Box::new(left),
        right: Box::new(right),
    })
}
pub fn e_mul(_ctx: &Ctx, left: E, right: E) -> E {
    E::Mul(Mul {
        left: Box::new(left),
        right: Box::new(right),
    })
}
pub fn e_div(_ctx: &Ctx, left: E, right: E) -> E {
    E::Div(Div {
        left: Box::new(left),
        right: Box::new(right),
    })
}
pub fn e_pow(_ctx: &Ctx, base: E, exp: E) -> E {
    E::Pow(Pow {
        base: Box::new(base),
        exp: Box::new(exp),
    })
}
pub fn e_paren(_ctx: &Ctx, e: E) -> E {
    E::Paren(Box::new(e))
}
pub fn e_num(_ctx: &Ctx, num: Num) -> E {
    E::Num(num)
}
```

We see that each grammar rule will have its corresponding type defined. Also, each production and each terminal will have an actions (Rust function) generated. You can change these manually and your manual modifications will be preserved on the next code generation as long as the name is the same.

Tip

On each code generation the existing `<>_actions.rs` file is parsed using syn crate and each type and action that is missing in the existing file is regenerated at the end of the file while the existing items are left untouched. The only items that cannot be preserved are non-doc comments.

This enables you to regenerate types/actions by just deleting them from the actions file and let rustemo run. If you want to regenerate actions/types from scratch just delete the whole file.

Here is an example of generated and manually modified actions for the same grammar above:

```rust
use super::calculator04_ambig_lhs::{Context, TokenKind};
/// This file is maintained by rustemo but can be modified manually.
/// All manual changes will be preserved except non-doc comments.
use rustemo::Token as BaseToken;
pub type Input = str;
pub type Ctx<'i> = Context<'i, Input>;
pub type Token<'i> = BaseToken<'i, Input, TokenKind>;
pub type Num = f32;
pub fn num(_ctx: &Ctx, token: Token) -> Num {
    token.value.parse().unwrap()
}
pub type E = f32;
pub fn e_add(_ctx: &Ctx, left: E, right: E) -> E {
    left + right
}
pub fn e_sub(_ctx: &Ctx, left: E, right: E) -> E {
    left - right
}
pub fn e_mul(_ctx: &Ctx, left: E, right: E) -> E {
    left * right
}
pub fn e_div(_ctx: &Ctx, left: E, right: E) -> E {
    left / right
}
pub fn e_pow(_ctx: &Ctx, base: E, exp: E) -> E {
    f32::powf(base, exp)
}
pub fn e_paren(_ctx: &Ctx, e: E) -> E {
    e
}
pub fn e_num(_ctx: &Ctx, num: Num) -> E {
    num
}
```

In these actions we are doing actual calculations. For the full explanation see the calculator tutorial.

Tip

Lexing context is passed to actions as a first parameter. This can be used to write semantic actions which utilize lexing information like position or surrounding content. For example, layout parser used with string lexer can use this to construct and return a borrowed string slice which span the layout preceding a next valid token. To be able to return a string slice, layout actions need access to the input string and start/end positions.

Generic tree builder

This is a built-in builder that will produce a generic parse tree (a.k.a *Concrete-Syntax-Tree (CST)*).

For example, given the grammar:

```
S: A+ B;
A: 'a' Num;

terminals
B: 'b';
Ta: 'a';
Num: /\d+/;
```

The test:

```
#[test]
fn generic_tree() {
    let result = GenericTreeParser::new().parse("a 42 a 3 b");
    output_cmp!(
        "src/builder/generic_tree/generic_tree.ast",
        format!("{:#?}", result)
    );
}
```

will produce this output:

```
Ok(
    NonTermNode {
        prod: S: A1 B,
        location: [1,0-1,10],
        children: [
            NonTermNode {
                prod: A1: A1 A,
                location: [1,0-1,8],
                children: [
                    NonTermNode {
                        prod: A1: A,
                        location: [1,0-1,4],
                        children: [
                            NonTermNode {
                                prod: A: Ta Num,
                                location: [1,0-1,4],
                                children: [
                                    TermNode {
                                        token: Ta("\"a\"" [1,0-1,1]),
                                        layout: None,
                                    },
                                    TermNode {
                                        token: Num("\"42\"" [1,2-1,4]),
                                        layout: Some(
                                            " ",
                                        ),
                                    },
                                ],
                                layout: None,
                            },
                        ],
                        layout: None,
                    },
                    NonTermNode {
                        prod: A: Ta Num,
                        location: [1,5-1,8],
                        children: [
                            TermNode {
                                token: Ta("\"a\"" [1,5-1,6]),
                                layout: Some(
                                    " ",
                                ),
                            },
                            TermNode {
                                token: Num("\"3\"" [1,7-1,8]),
                                layout: Some(
                                    " ",
                                ),
                            },
                        ],
                        layout: Some(
                            " ",
                        ),
                    },
                ],
                layout: None,
            },
            TermNode {
                token: B("\"b\"" [1,9-1,10]),
                layout: Some(
                    " ",
                ),
            },
        ],
        layout: None,
    },
)
```

We can see that we get all the information from the input. Each node in the tree is a `TermNode` or `NonTermNode` variant of `TreeNode` enum. Each node keeps the layout that precedes it.

For details see the full test.

> **Note**
>
> Generic builder can be configured by `Settings::new().builder_type(BuilderType::Generic)` settings API, exposed through `--builder-type generic` in the `rcomp` CLI.

Custom builders

If you have a specific requirement for the build process you can implement a builder from scratch.

To provide a custom builder you start with a type that implements a `rustemo::Builder` trait and after that implements a concrete parsing algorithm trait. Currently, Rustemo is a (G)LR parser thus you can use `rustemo::LRBuilder` trait.

Let's see how can we do all of this by implementing a builder that does on-the-fly calculation of the arithmetic expression. Start with a type and a base `Builder` trait implementation as each builder needs initialization and should be able to return the final result.

> **Note**
>
> To use a custom builder you should generate the parser with `--builder-type custom` if using rcomp CLI, or calling `rustemo_compiler::Settings::new().builder_type(BuilderType::Custom)` if generating parser from the `build.rs` script.

For example, given the grammar:

```
E: E '+' E {Add, 1, left}
 | E '*' E {Mul, 2, left}
 | Num {Num};

terminals
Plus: '+';
Mul: '*';
Num: /\d+/;
```

in the file `custom_builder.rustemo`, the following builder from file `custom_builder_builder.rs` will perform arithmetic operation on-the-fly (during parsing):

```rust
pub type E = i32;
pub type Context<'i> = LRContext<'i, str, State, TokenKind>;

/// Custom builder that perform arithmetic operations.
pub struct MyCustomBuilder {
    // A stack used to shift numbers and keep intermediate result
    stack: Vec<E>,
}

impl MyCustomBuilder {
    pub fn new() -> Self {
        Self { stack: vec![] }
    }
}

impl Builder for MyCustomBuilder {
    // Result of the build process will be a number
    type Output = E;

    fn get_result(&mut self) -> Self::Output {
        assert!(self.stack.len() == 1);
        self.stack.pop().unwrap()
    }
}
```

Now, implement LR part of the builder. For this we must specify what should be done for `shift/reduce` operations:

```rust
impl<'i> LRBuilder<'i, str, Context<'i>, State, ProdKind, TokenKind>
    for MyCustomBuilder
{
    fn shift_action(
        &mut self,
        _context: &mut Context<'i>,
        token: Token<'i, str, TokenKind>,
    ) {
        if let TokenKind::Num = token.kind {
            self.stack.push(token.value.parse().unwrap())
        }
    }

    fn reduce_action(
        &mut self,
        _context: &mut Context<'i>,
        prod: ProdKind,
        _prod_len: usize,
    ) {
        let res = match prod {
            ProdKind::EAdd => {
                self.stack.pop().unwrap() + self.stack.pop().unwrap()
            }
            ProdKind::EMul => {
                self.stack.pop().unwrap() * self.stack.pop().unwrap()
            }
            ProdKind::ENum => self.stack.pop().unwrap(),
        };
        self.stack.push(res);
    }
}
```

And finally, you call the parser with the instance of custom builder as:

```rust
    let result = CustomBuilderParser::new(MyCustomBuilder::new())
        .parse("2 + 4 * 5 + 20");
```

Tip

You can see the full test here.

Rustemo CLI

Crate `rustemo-compiler` installs a binary `rcomp` which is a CLI to the Rustemo compiler.

```
cargo install rustemo-compiler
```

To get all the option of the `rcomp` you can run `rcomp --help`.

The only mandatory argument is the path to `.rustemo` file from which you want the parser to be generated.

```
rcomp my_grammar.rustemo
```

The default lexer is a string lexer which uses string/regex recognizers from the grammar.

The default builder will call auto-generated actions and create automatically deduced AST.

> **Note**
>
> Instead of calling CLI manually you can setup your project to call the Rustemo compiler from `build.rs` script. You can read more in the configuration section.

Visualizing parser's automata

Besides providing a detailed information about the grammar and conflicts `rcomp` can produce a diagram of the parser's handle finding automaton.

This can be used to investigate conflicts visually for smaller grammars.

E.g. for ambiguous grammar `calc.rustemo`:

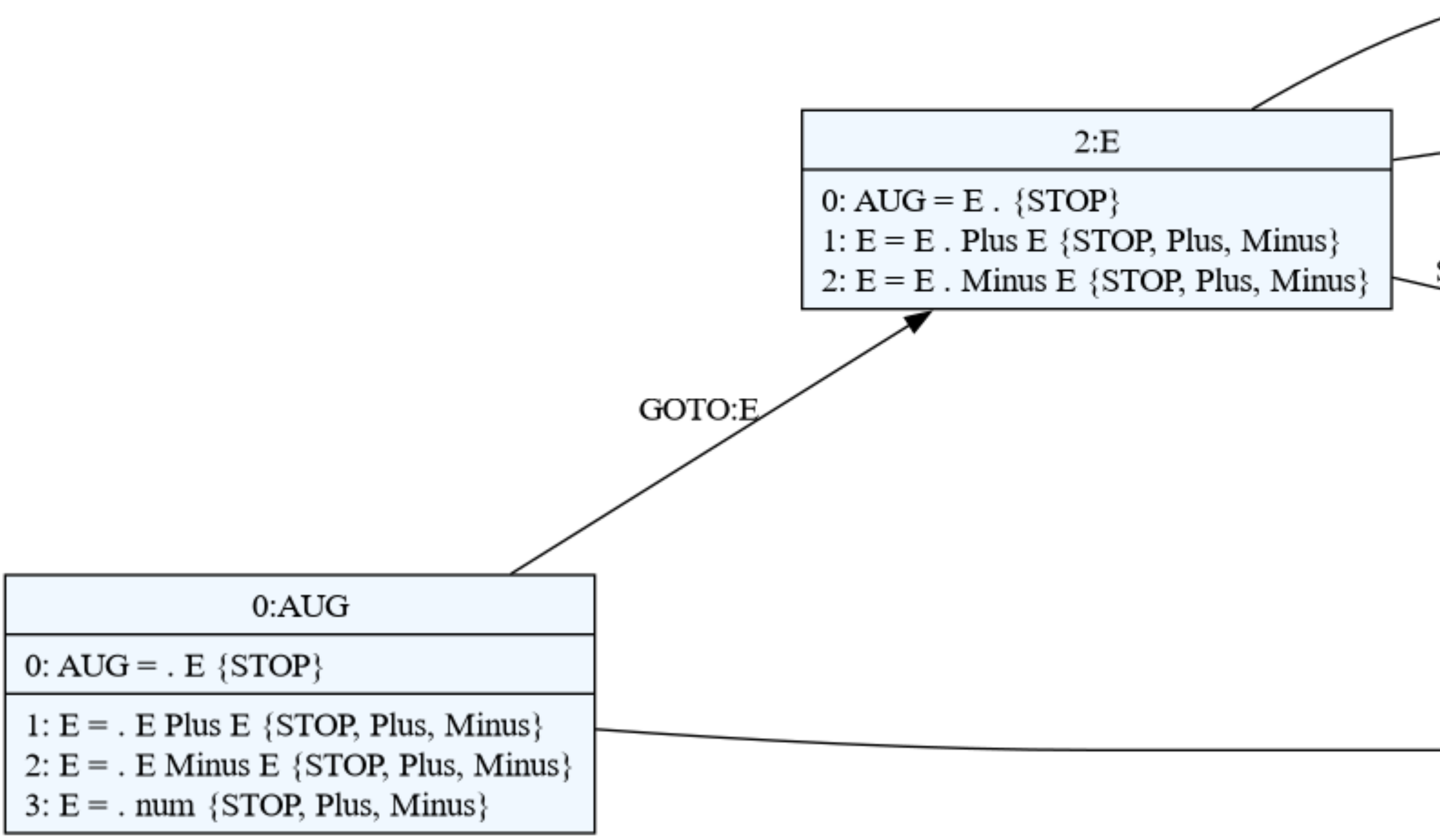
```
E: E '+' E
 | E '-' E
 | num;

terminals
Plus: '+';
Minus: '-';
num: /\d+/;
```

a diagram can be produced with:

```
rcomp --dot calc.rustemo
```

and either transformed to an image with GraphViz dot tool or opened directly using dot viewers like, for example, xdot.

Here is how the diagram for the above grammar looks like. States marked with red are states with conflicts.


2:E
0: AUG = E . {STOP}
1: E = E . Plus E {STOP, Plus, Minus}
2: E = E . Minus E {STOP, Plus, Minus}
GOTO:E
0:AUG
0: AUG = . E {STOP}
1: E = . E Plus E {STOP, Plus, Minus}
2: E = . E Minus E {STOP, Plus, Minus}
3: E = . num {STOP, Plus, Minus}

Handling errors

The error can occur at multiple places during development and usage of the parser. This chapter gives an overview of each class of errors and actions that can be taken to solve them.

The errors you can encounter with Rustemo are:

- Parser development errors:
    - Syntax errors in the grammar
    - Semantic errors in the grammar - invalid grammar (e.g. undefined symbol, infinite loop on a symbol)
    - LR conflicts - non-deterministic grammar (these errors apply only for LR not for GLR)
- Parser usage errors:
    - Syntax errors in the parsed input

When an error happens Rustemo tries to provide a detailed information of the problem. In the sections that follows an overview of each class is given.

Investigating grammar syntax errors

If your grammar is not written according to the Rustemo grammar language rules the Rustemo compiler will report an error with the information about the location and expected tokens at that location.

Here is an example of a grammar syntax error:

```
Error at json.rustemo:[3,7]:
    ...a] "}";
    Member -->JsonString ":" ...
    Expected one of Colon, OBrace.
```

The error report have a file and line/column location. You can also see the context where the error occurred with `-->` mark at the position.

Investigating grammar semantic errors

A grammar may be syntactically correct but still semantically incorrect. For example, you may have a reference to undefined grammar symbol.

Here is an example of a semantic error:

```
Error at json.rustemo:[3,8-3,17]:
    Unexisting symbol 'JsonStrin' in production '13: JsonStrin ":" Value'.
```

The location info contains the span where the symbol is specified in the grammar: from line 3 column 8 to line 3 column 17.

Resolving LR conflicts

> **Note**
>
> Please see the chapter on parsing for the introduction to LR parsing.

LR parsing is based on a deterministic finite-state automata. Because the generated automaton must be deterministic there can be exactly one operation the parser can perform at each state and for each input token.

Not all context-free grammars produce a deterministic automaton. Those that do are called deterministic context-free grammars (DCFL) and they are a proper subset of context-free grammars (CFG).

LR parser generators can produce parsers only for DCFLs.

> **Note**
>
> See this section for a generalized version of LR.

When there is a state in which multiple actions occur we have a conflict. Depending on the conflicting actions we may have:

- *shift-reduce* conflicts - where the parser could either shift the token ahead or reduce the top of the stack by some production,
- *reduce-reduce* conflicts - where the parser could reduce by two or more reductions for the given lookahead token.

In both cases Rustemo compiler produces a detailed information and it is a crucial to understand what is the problem and what you should do to make the grammar deterministic.

When a conflict arise we have a local ambiguity, i.e. Rustemo can't build a state in which the parser will know what to do by just looking at one token ahead. If the parser would know by looking at more tokens ahead than we have a local ambiguity, but if there are situations in which the parser couldn't decide with unlimited lookahead then our language is ambiguous which means that some inputs may have more than one interpretation.

Let's investigate a conflict on a famous *if-then-else* ambiguity problem. For example, if we have this little grammar which describes a language of nested `if` statements:

```
IfStatement: 'if' Condition 'then' Statements
           | 'if' Condition 'then' Statements 'else' Statements;
Statements: Statements Statement | Statement | EMPTY;
Statement: IfStatement;

terminals
If: 'if';
Then: 'then';
Else: 'else';
Condition: 'cond';
```

Then running `rcomp` would reveal that we have 3 shift/reduce conflicts. Let's investigate those conflicts one by one.

```
In State 6:Statements
    IfStatement: If Condition Then Statements .    {STOP, If, Else}
    IfStatement: If Condition Then Statements . Else Statements    {STOP, If, Else}
    Statements: Statements . Statement    {STOP, If, Else}
    Statement: . IfStatement    {STOP, If, Else}
    IfStatement: . If Condition Then Statements    {STOP, If, Else}
    IfStatement: . If Condition Then Statements Else Statements    {STOP, If, Else}
When I saw Statements and see token If ahead I can't decide should I shift or reduce
by production:
IfStatement: If Condition Then Statements
```

This state is reached when the parser has seen `Statements`. The state is described by so called *LR items*. These items are productions with a dot marking the position inside the production where the parser may be. At the same time, since the items are LR(1) (thus one lookahead), we have possible lookahead tokens in curly braces for that production position.

The end of the report (starting with `When I saw...`) gives the explanation. In this state, when there is `If` token as a lookahead, the parser can't determine whether it should shift that token or reduce previously seen `IfStatement`.

To put it differently, the question is if the next `If` statement should be nested inside the body of the previous `If` statement, in which case we should shift, or in the body of some outer `If` statement, in which case we should reduce previous statement and treat the next `If` as the statement on the same nesting level.

The rest two conflicts are similar:

```
In State 6:Statements
    IfStatement: If Condition Then Statements .    {STOP, If, Else}
    IfStatement: If Condition Then Statements . Else Statements    {STOP, If, Else}
    Statements: Statements . Statement    {STOP, If, Else}
    Statement: . IfStatement    {STOP, If, Else}
    IfStatement: . If Condition Then Statements    {STOP, If, Else}
    IfStatement: . If Condition Then Statements Else Statements    {STOP, If, Else}
When I saw Statements and see token Else ahead I can't decide should I shift or
reduce by production:
IfStatement: If Condition Then Statements

In State 10:Statements
    IfStatement: If Condition Then Statements Else Statements .    {STOP, If, Else}
    Statements: Statements . Statement    {STOP, If, Else}
    Statement: . IfStatement    {STOP, If, Else}
    IfStatement: . If Condition Then Statements    {STOP, If, Else}
    IfStatement: . If Condition Then Statements Else Statements    {STOP, If, Else}
When I saw Statements and see token If ahead I can't decide should I shift or reduce
by production:
IfStatement: If Condition Then Statements Else Statements
```

Second is for the same state but different token ahead, `else` in this case. The third is similar to the first.

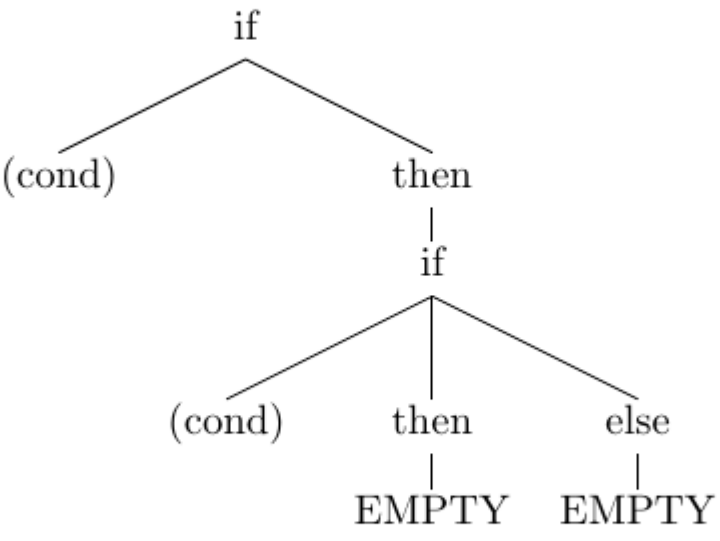
All three conflicts are due to the parser not knowing how to nest statements. An if we think about it, with our language defined as it is currently, there is no way to decide how we should nest statements. For example:

```
if cond then if cond then else
```

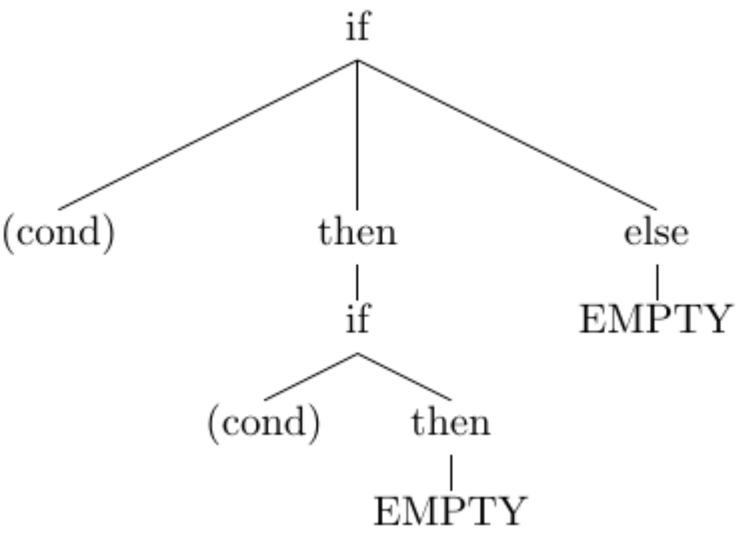
Note

Since the body of `then` and `else` branch may contain EMPTY the above example has empty body in the last `then` and `else` blocks.

The previous example could be interpreted as:



or



Thus, the `else` part can belong to the inner `If` statement as depicted in the first tree, or to the outer statement as shown in the second tree.

In this case, our language is trully ambiguous and larger lookahead won't help.

There are two approaches to handle these conflicts:

- Add additional rules and use disambiguation techniques (priorities and associativities meta-data)
- Change the grammar/language to eliminate ambiguities

When deciding whether to create a shift or reduce operation for a state Rustemo will look into priorities of terminals and productions and choose the one with higher priority. If the priorities are the same Rustemo will look at associativities, and favor the one set on terminal if not default (`None`).

So, in our case we could specify a greedy behavior for all three conflicts. This behavior means that we will favor shift operation thus always choosing to nest under the innermost statement.

```
IfStatement: 'if' Condition 'then' Statements
           | 'if' Condition 'then' Statements 'else' Statements;
Statements: Statements Statement | Statement | EMPTY;
Statement: IfStatement;

terminals
If: 'if' {shift};
Then: 'then';
Else: 'else' {shift};
Condition: 'cond';
```

Now, the grammar will compile, as the only possible interpretation of the above example will be the first tree.

Sometimes it is better and more readable to specify associativity on the production level. See the calculator tutorial for example.

The other way to solve the issue is by changing our language. The confusion is due to not knowing when the body of the `If` statement ends. We could add curly braces to delineate the body of `If` like all C-like languages do, or simply add keyword `end` at the end. Let's do the latter:

```
IfStatement: 'if' Condition 'then' Statements 'end'
           | 'if' Condition 'then' Statements 'else' Statements 'end';
Statements: Statements Statement | Statement | EMPTY;
Statement: IfStatement;

terminals
If: 'if';
Then: 'then';
Else: 'else';
End: 'end';
Condition: 'cond';
```

Now, our language is more flexible as the user can define the nesting by placing `end` keyword at appropriate places.

Global preference for resolving shift-reduce conflicts

One of the standard techniques to resolve shift/reduce conflicts is to prefer shift always which yields a greedy behavior.

This settings is available through `rcomp` CLI or `Settings` type API. By default shift is not preferred. Furthermore, there is a separate control on the preference of shift over empty reductions.

Tip

See the section on disambiguation meta-data in the grammar

Syntax errors in the parsed input

These are errors which you have to handle in your code as the user supplied an invalid input according to your grammar specification.

When you call `parse/parse_file` method with the input, you get a `Result` value where `Ok` variant will hold the result produced by the configured builder, while `Err` variant will hold information about the error.

For example, this can be a result of erroneous input where the result value is converted to a string:

```
Error at input2.calc:[2,9]:
    ...3 + 5
    / 7.54 * -->/ 78 + 3
    ...
    Expected Number.
```

You can see the path of the input file, line/column location of the error, the context where the error location is marked with `-->`, and finally the cause of the error (`Expected Number`).

The parser is called like this:

```
    let mut parser = CalculatorParser::new();
    let result = parser.parse_file(local_file!(file!(), "input2.calc"));
```

Error type contained in `Err` variant is defined as follows:

```
#[derive(Debug)]
pub enum Error {
    Error {
        message: String,
        file: Option<String>,
        location: Option<Location>,
    },
    IOError(std::io::Error),
}
```

As we can see, it either wraps `IOError` or, for Rustemo generated errors, provide `message` , `file` and `location` inside the file.

Handling ambiguities

Todo

1. Lexical ambiguities - when there can be recognized multiple tokens at the current position.
2. Syntactic ambiguities - applicable only to GLR - when multiple interpretation/trees of the input can be constructed.

These are not errors per se so should be moved to some other chapter.

Tutorials

This Chapter provides tutorials which will guide you through the process of creating fully working rustemo parsers.

Calculator

> Note
>
> This tutorial assumes that you have Rustemo CLI properly installed. Refer to the CLI section if you have trouble running `rcomp` command.

In this tutorial we'll create a simple calculator with 4 arithmetic operations: `+`, `-`, `*`, and `/`.

We would like to parse and calculate expressions like this:

```
8 + 4 / 2 - 3.2 * 2
```

> Note
>
> This tutorial is rather lengthy as it covers all aspect of usage workflow, from creating Rust project, Rustemo grammar file to implementing types and actions. Also, this tutorial tries to be a gentle introduction providing all intermediate steps in details. Thus, this should serve as a full introduction and a prerequisite for the following tutorials as it explains the usual workflow in working with Rustemo. Following tutorials will assume that the user is familiar with the usual workflow.

Create the project

We assume that we are building a small demo program which accepts an expression from the user and prints the result or informative error if the expression doesn't conform to the grammar.

So, lets first build a new project:

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

In this project we'll create a `calculator.rustemo`

```
cd calculator/src
touch calculator.rustemo
```

In the rest of the tutorial we assume that grammar is written in the above `calculator.rustemo` file.

The grammar

To parse such expressions we start with a grammar that describe the syntax and lexical structure of our expressions.

We start by a simple idea that each operation is binary, consisting of two operand and an infix operator:

```
<left operand> <operator> <right operand>
```

So, let's write that down in Rustemo grammar notation. First, we may say that our base expression is a single operation:

```
Expression: Operand Operator Operand;
```

Our operands are numbers. Let's define a lexical syntax for operands. We do that in a grammar section that starts with the `terminals` keyword. This section should go after the main part (syntax part) of the grammar. For lexical definitions, we either use plain strings if the content is fixed, or regular expressions, if the content is different between lexemes. See the section on terminals for the explanation.

In this case we have numbers which differs but we can encode their structure using regular expression:

```
terminals
Operand: /\d+(\.\d+)?/;
```

So, our operand should have at least one digit, after which optionally follows a dot and more digits. We could make this more elaborate but let's make it simple for now.

And what is the `operator`? It can be any of the `+`, `-`, `*`, `/` so we can encode that in a regex also:

```
Operator: /\+|-|\*|\//;
```

Symbols `+` and `*` have a special interpretation in regular expressions so we must escape them. Symbol `/` is used to start/end the regex in Rustemo so we must escape that also.

Now, our full grammar is:

```
Expression: Operand Operator Operand;

terminals
Operand: /\d+(\.\d+)?/;
Operator: /\+|-|\*|\//;
```

The problem with our grammar is that the only thing we could ever parse with it is a single operation, like `3 + 4`. If we add more operations our parser, built with this grammar, won't work anymore. We'll see how to extend the parser later but let's now move on.

Generating the parser

Let's run `rcomp` command to generate the parser code from the grammar:

```
rcomp calculator.rustemo
```

If you get no output there were no errors. If you made an error in the grammar you will get a report with the line and column where the error was and what is expected at that location.

For example, if we forget colon after the rule name we get:

```
Error at calculator.rustemo:1:11:
    ...Expression -->Operand Operato...
    Expected one of Colon, OBrace.
Parser(s) not generated.
```

After the parser generator is run successfully you should have files `calculator.rs` and `calculator_actions.rs` generated.

File `calculator.rs` is the parser while `calculator_actions.rs` in the default configuration contains deduced types for the AST (*Abstract Syntax Tree*) together with functions/actions used by the builder during the parsing process to construct the AST.

Regenerating the parser

`calculator.rs` is regenerated whenever you run `rcomp` command.

Actions in file `calculator_actions.rs`, on the other hand, are not fully regenerated. Only missing actions/types will be regenerated. This enables you to provide manual modifications of the actions/types as long as you retain the same name. As we shall soon see, this is a nice feature which provides a quick way to have a working parser with default AST output which can be later tuned as needed.

Adding dependencies

Our generated parser code calls Rustemo code so we must add `rustemo` crate as a dependency:

```
cargo add rustemo
```

Since we are using regular expressions in our grammar we also need `regex` and `once_cell`:

```
cargo add regex --no-default-features --features std,unicode-perl
cargo add once_cell
```

Your `Cargo.toml` should look like this:

```toml
[package]
name = "calculator1"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/
manifest.html

[dependencies]
once_cell = "1"
regex = { version = "1.7.1", default-features = false, features = ["std", "unicode-
perl"] }
colored = "2"
# A relative path to rustemo crate is used here for usage in the rustemo project
tree.
# In your projects you should just specify the version.
rustemo = { version = "0.6.0", path = "../../../../../rustemo" }
```

Running the parser

Now, let's open the `main.rs` file and write this for the header:

```rust
use rustemo::Parser;
use std::io;
// Use the generated parser
use crate::calculator::CalculatorParser;

// Include generated modules
#[rustfmt::skip]
mod calculator;
#[allow(unused)]
#[rustfmt::skip]
mod calculator_actions;
```

and this as the `main` function which will serve as entry point for our program:

```rust
fn main() {
    let mut expression = String::new();

    // Read the line from the input
    println!("Expression:");
    io::stdin()
        .read_line(&mut expression)
        .expect("Failed to read line.");

    // Parse the line and get the result.
    let result = CalculatorParser::new().parse(&expression);

    // Print the result using Debug formatter.
    println!("{:#?}", result);
}
```

So, our initial program will accept the string from the console input, parse it using the parser generated in the `calculator` module and print the result.

Run the program:

```
cargo run
```

You should see a prompt which expect you to enter the expression. If we enter:

```
2 + 3
```

We get debug output from the parser and at the end you should see:

```
Action: Accept
Ok(
    Expression {
        operand_1: "2",
        operator: "+",
        operand_3: "3",
    },
)
```

But if you make a mistake, like:

```
2 + / 3
```

You get:

```
Err(
    Error {
        message: "...2 + -->/ 3\n...\nExpected Operand.",
        file: Some(
            "<str>",
        ),
        location: Some(
            Location {
                start: LineBased(
                    LineBased {
                        line: 1,
                        column: 4,
                    },
                ),
                end: None,
            },
        ),
    },
)
```

Congrats! You have made your first Rustemo parser!

Although, it still have many issues which we'll fix in the rest of the tutorial.

Extending the grammar

Ok, let's see can we parse the example expression from the beginning of the tutorial:

```
8 + 4 / 2 - 3.2 * 2
```

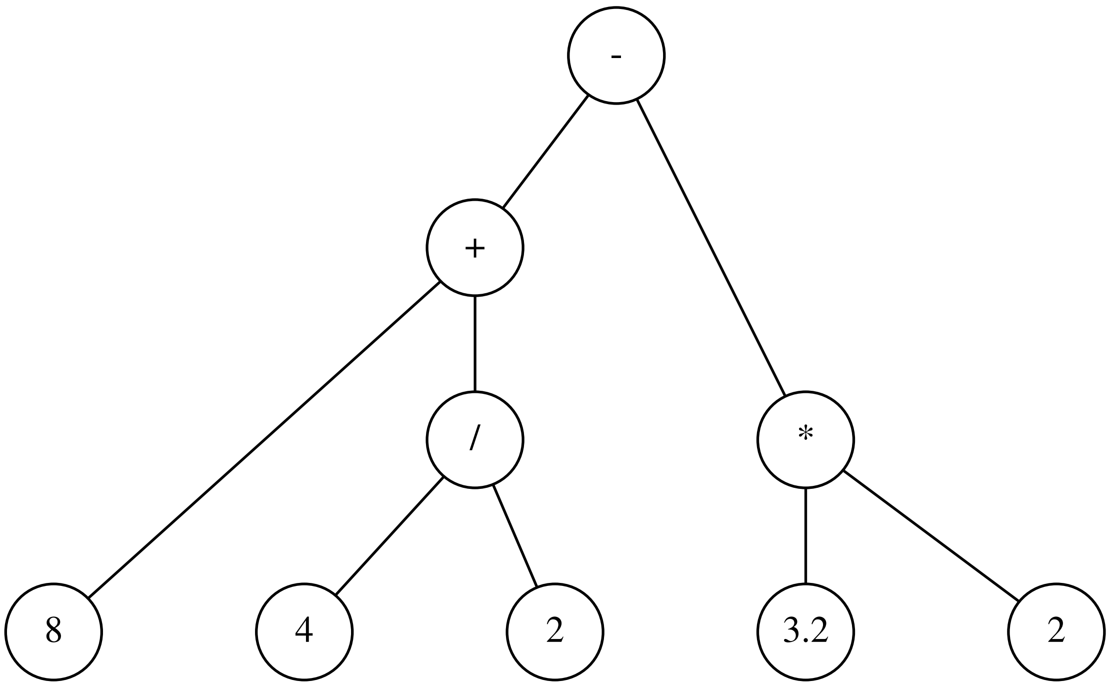
Run the program and give it the expression:

```
$ cargo run
...
Expression:
8 + 4 / 2 - 3.2 * 2
...
Err(
    Error {
        message: "...8 + 4 -->/ 2 - 3.2 * 2\n...\nExpected STOP.",
        file: Some(
            "<str>",
        ),
        location: Some(
            Location {
                start: LineBased(
                    LineBased {
                        line: 1,
                        column: 6,
                    },
                ),
                end: None,
            },
        ),
    },
)
```

As we already discussed above, the grammar we created so far can only parse an expression consisting of a single operation. We can see that the parser reported the error at the occurrence of the second operation.

Let's extend our grammar to allow for expressions of arbitrary length.

First, we shall start from the idea the each expression can be represented as an Abstract Syntax Tree (AST):



These kinds of trees resemble the gist of the underlying expressions. We can observe two things:

1. AST shows the priority of operations. `*` and `/` bind stronger than `+` and `-`. We see that operations with higher priority will be lower in the tree.
2. Each node in the tree is a sub-expression. Numbers are the most basic expressions, each operation is a sub-expression consisting of left sub-tree/sub-expression, operation and right sub-tree/sub-expression.

So, we could write grammar rules as:

```
Add: Expression '+' Expression;
Sub: Expression '-' Expression;
...
```

Where `Add`, `Sub` ... are all expressions, thus:

```
Expression: Add | Sub | Mul | Div;
```

The definition is recursive, e.g. `Add` references `Expression` through its operands and `Expression` references `Add`.

Finally, we can write this in a more compact form as a single grammar rule (contracting `Expression` to `E`):

```
E: E '+' E
 | E '-' E
 | E '*' E
 | E '/' E
 | Number;

terminals
Number: /\d+(\.\d+)?/;
Plus: '+';
Minus: '-';
Mul: '*';
Div: '/';
```

We can read this as:

Expression is:

- Expression plus Expression or
- Expression minus Expression or
- ...
- or a number

Note

We must add operations to the `terminals` section for the sake of giving names to symbols as the names are needed for the generated code. We can use those terminals directly in the syntax part e.g. like:

```
E: E Plus E
```

but I find it more readable to use strings in this case.

Of course, you can use either references or strings, or even combine the two approaches.

Let's run `rcomp` over our new grammar to generate the parser.

```
$ rcomp calculator.rustemo
In State 7:E
E: E Plus E .     {STOP, Plus, Minus, Mul, Div}
E: E . Plus E     {STOP, Plus, Minus, Mul, Div}
...
16 conflict(s). 16 Shift/Reduce and 0 Reduce/Reduce.
Error: Grammar is not deterministic. There are conflicts.
Parser(s) not generated.
```

Woops! What went wrong?

Rustemo is talking about some conflicts and that the grammar is not deterministic. This just means that LR parser cannot be produced with the grammar because LR is a deterministic style of parsing where parser must know exactly what operation to perform in each state based solely on the next token. With this grammar it is not possible.

The report also gives detailed information about problematic spots. The report talks about states, those are LR automaton states as the LR parsing is based on constructing a deterministic push-down automaton (PDA) which is used during parsing.

Tip

To learn more see the section on parsing and conflicts resolution.

In the report we see segments like this:

```
In State 10:E
E: E Div E .    {STOP, Plus, Minus, Mul, Div}
E: E . Plus E    {STOP, Plus, Minus, Mul, Div}
E: E . Minus E    {STOP, Plus, Minus, Mul, Div}
E: E . Mul E    {STOP, Plus, Minus, Mul, Div}
E: E . Div E    {STOP, Plus, Minus, Mul, Div}
When I saw E and see token Minus ahead I can't decide should I shift or reduce by
production:
E: E Div E
```

Each parser state represent a location where parser can be in the parsing process. The location is depicted by the dot in the productions above. So in this state parser saw `E` or `E Div E` before and if there is `Minus` ahead it won't be able to decide if it should construct a `Div` sub-tree making `Div` operation to bind tighter or to consume the `Minus` symbol which follows in anticipation to construct `Minus` sub-tree first.

Obviously, we have an issue with our operation priorities. Indeed, this grammar is ambiguous as, if we forget about priorities, input expressions can yield many possible trees[1]. LR parsers must always produce only one tree as LR parsing is deterministic.

> **Note**
>
> GLR parsing is non-deterministic so it can be used with ambiguous grammars.

This *could* be resolved by transforming our grammar to encode priorities but the process is tedious, the resulting grammar becomes less readable and the resulting trees are far from intuitive to navigate and process.

Luckily, Rustemo has declarative disambiguation mechanisms which makes these issues easy to solve. These disambiguation information are specified inside of curly braces per production or per grammar rule.

The priority is specified by an integer number. The default priority is 10. Productions with higher priority will be the first to be reduced. Think of the reduction as producing a tree node where the node type is determined by the left-hand side of the production while the children are right-hand side.

Ok, knowing this let's extend our grammar:

```
E: E '+' E {1}
 | E '-' E {1}
 | E '*' E {2}
 | E '/' E {2}
 | Number;

terminals
Number: /\d+(\.\d+)?/;
Plus: '+';
Minus: '-';
Mul: '*';
Div: '/';
```

Nice, we now have priorities defined. `+` and `-` operations are of priority `1` while `*` and `/` are of priority `2`. So, in the above conflict, division will be reduced before `Minus` symbol (subtraction) is taken into consideration.

Let's run `rcomp` again:

```
$ rcomp calculator.rustemo
...
8 conflict(s). 8 Shift/Reduce and 0 Reduce/Reduce.
Error: Grammar is not deterministic. There are conflicts.
Parser(s) not generated.
```

Hm... we still have ambiguities but we've cut them in half. Let's see what are those remaining issues:

```
In State 10:E
E: E Div E .    {STOP, Plus, Minus, Mul, Div}
E: E . Plus E    {STOP, Plus, Minus, Mul, Div}
E: E . Minus E    {STOP, Plus, Minus, Mul, Div}
E: E . Mul E    {STOP, Plus, Minus, Mul, Div}
E: E . Div E    {STOP, Plus, Minus, Mul, Div}
When I saw E and see token Mul ahead I can't decide should I shift or reduce by
production:
E: E Div E
```

Ok, now we don't have issues with priorities but we have issues with associativities. When the parser saw division and there is multiplication symbol ahead it doesn't know what to do. Should it favor division or multiplication? They are of the same priority. We know that for all 4 operation parser should favor the operation it has already seen, or to put it simply, it should go from left to right when the priorities are the same[2].

Rustemo also have declarative associativity specification. Associativity is specified by keywords `left` (or `reduce`) and `right` (or `shift`).

Let's fix our grammar:

```
E: E '+' E {1, left}
 | E '-' E {1, left}
 | E '*' E {2, left}
 | E '/' E {2, left}
 | Number;

terminals
Number: /\d+(\.\d+)?/;
Plus: '+';
Minus: '-';
Mul: '*';
Div: '/';
```

And run `rcomp` again.

```
$ rcomp calculator.rustemo
$
```

It seems that everything is fine. Let's check by running our project:

```
$ cargo run
...
Expression:
8 + 4 / 2 - 3.2 * 2
...
Action: Accept
Ok(
    C2(
        EC2 {
            e_1: C1(
                EC1 {
                    e_1: C5(
                        "8",
                    ),
                    e_3: C4(
                        EC4 {
                            e_1: C5(
                                "4",
                            ),
                            e_3: C5(
                                "2",
                            ),
                        },
                    ),
                },
            ),
            e_3: C3(
                EC3 {
                    e_1: C5(
                        "3.2",
                    ),
                    e_3: C5(
                        "2",
                    ),
                },
            ),
        },
    ),
)
```

That's it! We have a working grammar. It wasn't that hard after all, was it? :)

But, the work is not over yet. We got a parser but the AST is not that pretty and how can we evaluate our expressions? If you want to learn how to do that read on.

Improving AST

When we run `rcomp` for our grammar we got two files generated: `calculator.rs` and `calculator_actions.rs`. The first one is the parser that is regenerated from scratch whenever we run `rcomp`. The second contains AST nodes' types and actions used by the parser to construct AST nodes during parsing. This file can be manually modified and the modification will be retained when `rcomp` is run as we shall see in the next section.

At this point we shall focus on the form of AST types that Rustemo generated for us. Open `calculator_actions.rs` file and examine its content. We can see that for each grammar rule production we got struct generated with the name of the grammar rule followed by `Cx` sufix (`C` for Choice and `x` is an ordinal number).

```
#[derive(Debug, Clone)]
pub struct EC1 {
    pub e_1: Box<E>,
    pub e_3: Box<E>,
}
#[derive(Debug, Clone)]
pub struct EC2 {
    pub e_1: Box<E>,
    pub e_3: Box<E>,
}
```

Tip

You can expand and see the whole code in these snippets by using "Show hidden lines" option in the upper right corner of the code box.

And for each grammar rule we have an enum combining the variant structs:

```
#[derive(Debug, Clone)]
pub enum E {
    C1(EC1),
    C2(EC2),
    C3(EC3),
    C4(EC4),
    Number(Number),
}
```

Notice also that recursive types are boxed (e.g. `pub e_1: Box<E>;`) as they should be.

Ok, that is nice but look at those struct names and fields. They don't give us clue about operations, operands etc. It would be pretty hard to use these types.

Let's improve it a bit. First we can specify production kinds which is just a nice name for each production which can be used in the code.

We *could* workaround the issue by making a separate grammar rule for each operation like suggested above:

```
Add: E '+' E;
Sub: E '-' E;
...
E: Add | Sub |...
```

But let's do it with production kinds to see that alternative.

```
E: E '+' E {Add, 1, left}
 | E '-' E {Sub, 1, left}
 | E '*' E {Mul, 2, left}
 | E '/' E {Div, 2, left}
 | Number;

terminals
Number: /\d+(\.\d+)?/;
Plus: '+';
Minus: '-';
Mul: '*';
Div: '/';
```

Notice the additions of `Add` , `Sub` ... in the curly braces.

Regenerate the parser, but first delete actions so they can be regenerated also.

```
$ rm calculator_actions.rs
$ rcomp calculator.rustemo
```

Now, if we open `calculator_actions.rs` we'll see that structs are named after productions kinds. Nice!

```
#[derive(Debug, Clone)]
pub struct Add {
    pub e_1: Box<E>,
    pub e_3: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Sub {
    pub e_1: Box<E>,
    pub e_3: Box<E>,
}
```

But, what about fields. It is certainly not nice to have those generic `e_1, e_3` names. To fix these we can use assignments which is a mechanism to both define AST nodes' field names and specify to the parser what to retain in the AST during parsing. Some parts of the input are syntax noise and should not be kept in the AST.

To change field names add assignments for `left` and `right` operands in each operation:

```
E: left=E '+' right=E {Add, 1, left}
 | left=E '-' right=E {Sub, 1, left}
 | left=E '*' right=E {Mul, 2, left}
 | left=E '/' right=E {Div, 2, left}
 | Number;

terminals
Number: /\d+(\.\d+)?/;
Plus: '+';
Minus: '-';
Mul: '*';
Div: '/';
```

Delete actions and rerun `rcomp` . Now you'll see that the generated structs have nicely named fields.

```
#[derive(Debug, Clone)]
pub struct Add {
    pub left: Box<E>,
    pub right: Box<E>,
}
#[derive(Debug, Clone)]
pub struct Sub {
    pub left: Box<E>,
    pub right: Box<E>,
}
```

Much easier to work with!

In general, it is a good practice to use assignments and production kinds from the start as they represent valuable information (a sort of docs). The AST types generator uses those information, as you have seen.

Let's run our calculator just to make sure it works:

```
$ cargo run
...
Expression:
8 + 4 / 2 - 3.2 * 2
...
Action: Accept
Ok(Sub(Sub { left: Add(Add { left: C5("8"),
right: Div(Div { left: C5("4"), right: C5("2") }) }),
right: Mul(Mul { left: C5("3.2"), right: C5("2") }) }))
```

Everything is fine. Let's move on.

Calculating expressions

The last piece of the puzzle is to make our calculator useful by doing the actual calculation.

When we are parsing input we are doing what we call "semantic analysis" which purpose is to transform the input to some other form which is of use in the domain we are working in.

Usually, the end result of the analysis is some form of AST or more often ASG (Abstract Semantic Graph). In our case, semantic analysis can directly produce the end result of the calculation so that the result of the full process of parsing is a single number which represents the result.

Semantic analysis is done by the actions in `calculator_actions.rs`. The actions are Rust functions which are called during reductions to reduce a production (right-hand side) to the sub-result (left-hand side of the grammar rule). In our case, each reduction should basically be a calculation of a sub-expression result.

As you have seen in the previous section, you need to delete actions if you want them regenerated on each Rustemo run. You can also remove some parts of it and those parts will get regenerated. The logic is that Rustemo will only add missing parts (identified by its name) of the file (missing AST types and action functions) but will not modify already existing. This enables manual modification of parts we want to tune to our likings.

We will now change types and actions to calculate sub-results instead of building AST nodes.

First, we'll start with the `Number` rule which is defined as a `String`:

```
pub type Number = String;
```

We know that our number should be `float`, so change it:

```
pub type Number = f32;
```

Just bellow the `Number` type we see the first action which is called to transform the parsed number to `Number`. `token.value` is a string slice containing the number.

```
pub fn number(_ctx: &Ctx, token: Token) -> Number {
    token.value.into()
}
```

Previously our `Number` was `String` but now it is a float so we should change this action to parse the string and produce a float:

```
pub fn number(_ctx: &Ctx, token: Token) -> Number {
    token.value.parse().unwrap()
}
```

Now, we see that `E` type which represents sub-results is enum.

```
#[derive(Debug, Clone)]
pub enum E {
    Add(Add),
    Sub(Sub),
    Mul(Mul),
    Div(Div),
    Number(Number),
}
```

A sub-result should be float also. So replace the enum with:

```
pub type E = f32;
```

Also, structs for values of variants `Add`, `Sum` ... are not needed anymore so remove them from the file.

Okay, we have fixed generated types, now let's see our expressions' actions. They are of the form:

```
pub fn e_add(_ctx: &Ctx, left: E, right: E) -> E {
    E::Add(Add {
        left: Box::new(left),
        right: Box::new(right),
    })
}
```

So they produce AST node while they should do the calculation. So let's change all of them:

```
pub fn e_add(_ctx: &Ctx, left: E, right: E) -> E {
    left + right
}
pub fn e_sub(_ctx: &Ctx, left: E, right: E) -> E {
    left - right
}
pub fn e_mul(_ctx: &Ctx, left: E, right: E) -> E {
    left * right
}
pub fn e_div(_ctx: &Ctx, left: E, right: E) -> E {
    left / right
}
pub fn e_number(_ctx: &Ctx, number: Number) -> E {
    number
}
```

Tip

Just a reminder that you can see the whole code by using "Show hidden lines".

Very simple, each action is doing its calculation! Notice the last one which is called to reduce `Number` to `E`. It just returns the number as it is already a float produced by the `number` action.

That's it. We have a fully working calculator!

Let's run it just to verify:

```
$ cargo run
...
Expression:
8 + 4 / 2 - 3.2 * 2
...
Action: Accept
Ok(3.6)
```

If you have made it through here, well done!

Now, just to make sure that you have connected all the dots try to solve exercises bellow.

Exercises

1. Extend the language to support power operator (`^`). What are priority and associativity of this operation?

2. Support unary minus operation.

3. Support trigonometric operations in a function call style (e.g. `sin(34) + 2 * cos(12)`)

4. Support variables before an expression. E.g:

   ```
   a = 5
   b = 2 * a
   a + 2 * sin(b)
   ```

   Extend `main.rs` to accept input line-by-line. Whenever the user enter assignment to the variable the program will update its internal state by evaluating RHS of the assignment and keeping variable values in some mapping data structure of your choice. Each defined variable can be used in subsequent expressions. When the user enters just an expression without assignment the parser should evaluate and print the result.

Footnotes

[1] The number of trees (interpretations) of an arithmetic expression can be calculated by the Catalan number sequence. For our example with 4 operations from the beginning the number of trees will be 14.

[2] This doesn't hold for some other operations. See Exercises and think about associativity of the power operation.

This guide is WIP at the moment

Contributing

Contributions are welcome, and they are greatly appreciated!. You can contribute code, documentation, tests, bug reports. Every little bit helps, and credit will always be given. If you plan to make a significant contribution it would be great if you first announce that in the Discussions.

You can contribute in many ways:

Types of Contributions

1. Report Bugs

   Report bugs at https://github.com/igordejanovic/rustemo/issues.

   If you are reporting a bug, please include:

   - Your operating system name and version.
   - Any details about your local setup that might be helpful in troubleshooting.
   - Detailed steps to reproduce the bug.

2. Fix Bugs

   Look through the GitHub issues for bugs. Anything tagged with `bug` and `help wanted` is open to whoever wants to implement it.

3. Implement Features

   Look through the GitHub issues for features. Anything tagged with `enhancement/feature` and `help wanted` is open to whoever wants to implement it.

4. Write Documentation

   Rustemo could always use more documentation, whether as part of the official Rustemo docs, in documentation comments, or even on the web in blog posts, articles, and such.

   Rustemo is using mdbook for the official documentation.

Bootstrapping

This section describes the bootstrap process which is essential to understand in order to contribute to the development of the Rustemo library.

It is usual for compiler compilers to be implemented using themselves. Rustemo is no different. in `rustemo/src/lang` you can find grammar `rustemo.rustemo` which is a description of the rustemo grammar language. This description is then used to generate a parser for rustemo grammar files.

The problem with bootstrapping is a classical chicken and egg problem. To generate the parser you need a working parser. The problem is solved by using a previous version to generate the next.

While the solution seems simple it is not easily achieved from the organizational point of view. E.g., when you change parser generator code you would like to have rustemo parser regenerated with the new code but the current parser version might not be functional at that point.

Thus, rustemo defines a bootstrapping process to help with the development. The idea is to build bootstrapping rustemo binary with the parser code from the git `main` branch and the rest of the code from the current source tree.

If you are not changing the rustemo grammar or the parser code generator you won't need bootstrapping and should proceed as usual with Cargo commands.

But, if you do need to change the rustemo grammar or parser code generator you should install bootstrapping binary with the following command.

```
$ cargo install --path rustemo-compiler --features bootstrap --debug
```

The `--debug` switch is optional but will provide faster build and the built binary will provide better error reports in case of problems.

Note the use of `--features bootstrap`. This command will checkout rustemo parser files (parser and actions) from the git `main` branch, do the build with the rest of the code and install the binary.

You can verify that the bootstrapping binary is used by checking the version:

```
$ rcomp --version
rustemo-compiler 0.1.0-1a45d75ca4-bootstrap
```

Note

It is assumed that the `main` branch contains a working parser.

When the bootstrapping binary is installed you develop as usual and run tests:

```
$ cargo test
```

Whenever you change the rustemo grammar you should regenerate the parser code with `rcomp` binary:

```
rcomp rustemo/src/lang/rustemo.rustemo
```

If bootstrapping binary is used, code generation templates from the working tree when the binary was last built are used. Thus, regenerate bootstrapping binary whenever you change parser code generation templates.

This will also check your grammar for syntax and semantic errors and report diagnostics.

If feature `bootstrap` is not provided during rustemo installation, Cargo proceeds as usual by using current versions of parser files to build the binary.

The architecture

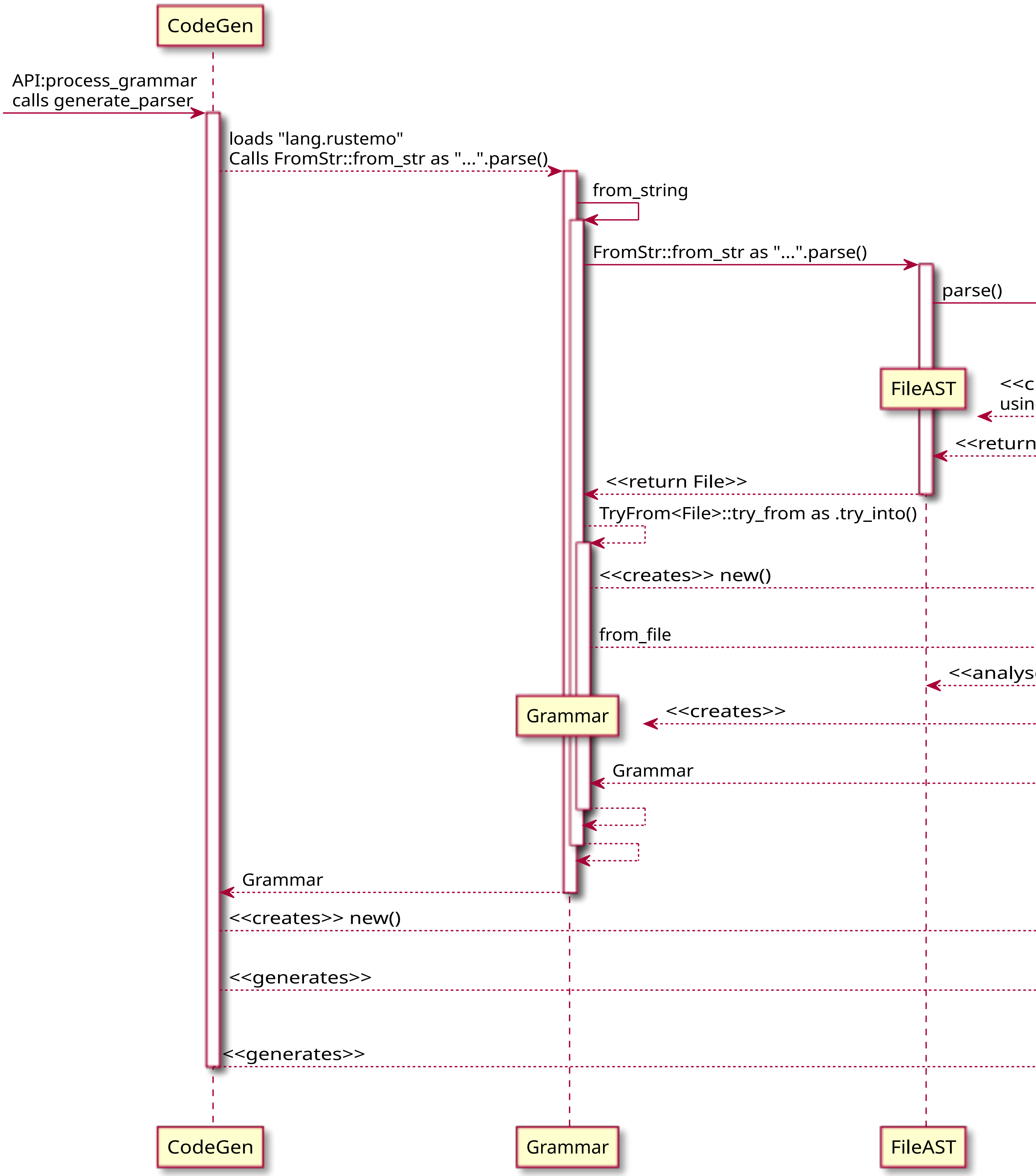
This section describe the main parts of Rustemo and their interplay. The purpose of this section is to give an overview of the process, main modules and their responsibilities.

The parser generation (a.k.a. grammar compilation) process

The following diagram describes the process of transforming of Rustemo grammar to a working parser. The input file is given as `lang.rustemo`. Where `lang` is the name of the language describe by the grammar.

Todo

The diagram is sligtly out-of-date. Needs update.



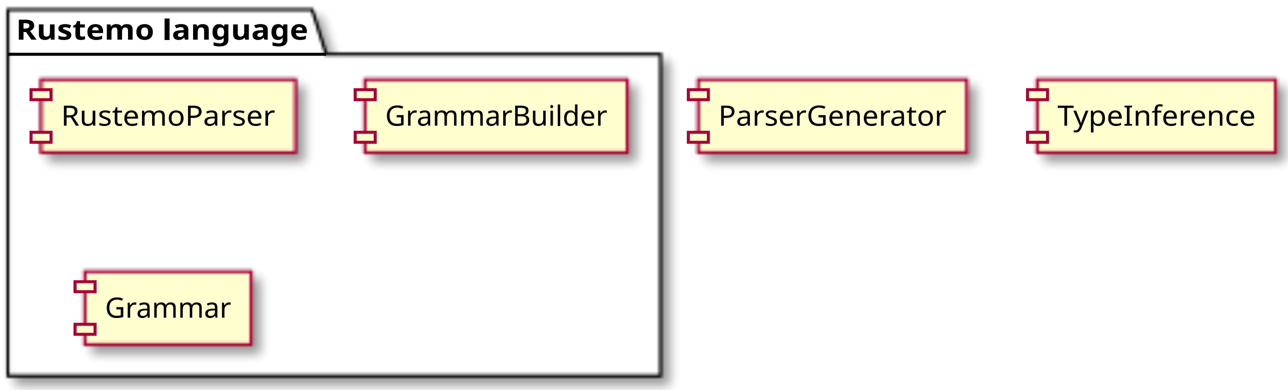
Note

The above grammar assumes the default configuration. It might be slightly different if non-defaults are used. For example, if custom builder is used type inference and generating actions (the last step) is not performed.

Note

Rustemo is bootstrapped using itself, thus, the above diagram describes also the process of bootstrapping. The input grammar is `src/lang/rustemo.rustemo` and the process generates `src/lang/rustemo.rs` which is the parser code and `src/lang/rustemo_actions.rs` which are semantic actions for AST construction. The actions are generated by the AST auto-generation process and manually adapted.

TODO Packages



Installation and setup

To render this document locally in your browser do (this should be done only once to install `mdbook`):

1. Install Rust
2. With cargo install mdbook and used processors

```
cargo install mdbook --git https://github.com/igordejanovic/mdBook.git --branch
merged-prs
cargo install mdbook-admonish
cargo install mdbook-bib
cargo install mdbook-theme
cargo install mdbook-plantuml
cargo install mdbook-graphviz
cargo install mdbook-linkcheck
```

Optional:

1. For pdf output:

```
cargo install mdbook-pdf
```

2. For pdf outline support see this:

```
pip install --user mdbook-pdf-outline
```

In the root of the `docs` directory run `mdbook serve` . The book will be available at http://localhost:3000/

Admonitions

This doc uses `mdbook-admonish` pre-processor. See here.

Including files

Including files ensure that the docs is up-to-date with the content it is referring to.

See this part of the manual.

Cross references

See this on how to reference to other section of the document.

Bibliography

Uses mdbook-bib to reference books, papers etc.

Diagrams

For UML diagrams we use PlantUML, while for general graphs and trees GraphViz is used.

Trees diagrams

For tree diagrams LaTeX (`pdflatex`) with `qtree` package is used to produce PDF and afterwards the PDF file is converted to PNG using `pdftoppm` . See `docs/build-latex-images.sh` script. This script must be called whenever `.tex` files with trees description are changed.