Introduction

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited. Tree-sitter aims to be:

- **General** enough to parse any programming language
- **Fast** enough to parse on every keystroke in a text editor
- **Robust** enough to provide useful results even in the presence of syntax errors
- **Dependency-free** so that the runtime library (which is written in pure C11) can be embedded in any application

Language Bindings

There are bindings that allow Tree-sitter to be used from the following languages:

Official

- C#
- Go
- Haskell
- Java (JDK 22+)
- JavaScript (Node.js)
- JavaScript (Wasm)
- Kotlin
- Python
- Rust
- Swift
- Zig

Third-party

- C# (.NET)
- C++
- Crystal
- D
- Delphi
- ELisp
- Go
- Guile
- Janet
- Java (JDK 8+)
- Java (JDK 11+)
- Julia
- Lua
- Lua
- OCaml
- Odin
- Perl
- Pharo
- PHP
- R
- Ruby

Keep in mind that some of the bindings may be incomplete or out of date.

Parsers

The following parsers can be found in the upstream organization:

- Agda
- Bash
- C
- C++
- C#
- CSS
- ERB / EJS
- Go
- Haskell

- HTML
- Java
- JavaScript
- JSDoc
- JSON
- Julia
- OCaml
- PHP
- Python
- Regex
- Ruby
- Rust
- Scala
- TypeScript
- Verilog

A list of known parsers can be found in the wiki.

Talks on Tree-sitter

- Strange Loop 2018
- FOSDEM 2018
- GitHub Universe 2017

Underlying Research

The design of Tree-sitter was greatly influenced by the following research papers:

- Practical Algorithms for Incremental Software Development Environments
- Context Aware Scanning for Parsing Extensible Languages
- Efficient and Flexible Incremental Parsing
- Incremental Analysis of Real Programming Languages
- Error Detection and Recovery in LR Parsers
- Error Recovery for LR Parsers

Using Parsers

This guide covers the fundamental concepts of using Tree-sitter, which is applicable across all programming languages. Although we'll explore some C-specific details that are valuable for direct C API usage or creating new language bindings, the core concepts remain the same.

Tree-sitter's parsing functionality is implemented through its C API, with all functions documented in the tree_sitter/api.h header file, but if you're working in another language, you can use one of the following bindings found here, each providing idiomatic access to Tree-sitter's functionality. Of these bindings, the official ones have their own API doc hosted online at the following pages:

- Go
- Java
- JavaScript (Node.js)
- Kotlin
- Python
- Rust
- Zig

Getting Started

Building the Library

To build the library on a POSIX system, just run `make` in the Tree-sitter directory. This will create a static library called `libtree-sitter.a` as well as dynamic libraries.

Alternatively, you can incorporate the library in a larger project's build system by adding one source file to the build. This source file needs two directories to be in the include path when compiled:

source file:

- `tree-sitter/lib/src/lib.c`

include directories:

- `tree-sitter/lib/src`
- `tree-sitter/lib/include`

The Basic Objects

There are four main types of objects involved when using Tree-sitter: languages, parsers, syntax trees, and syntax nodes. In C, these are called `TSLanguage`, `TSParser`, `TSTree`, and `TSNode`.

- A `TSLanguage` is an opaque object that defines how to parse a particular programming language. The code for each `TSLanguage` is generated by Tree-sitter. Many languages are already available in separate git repositories within the Tree-sitter GitHub organization and the Tree-sitter grammars GitHub organization. See the next section for how to create new languages.
- A `TSParser` is a stateful object that can be assigned a `TSLanguage` and used to produce a `TSTree` based on some source code.
- A `TSTree` represents the syntax tree of an entire source code file. It contains `TSNode` instances that indicate the structure of the source code. It can also be edited and used to produce a new `TSTree` in the event that the source code changes.
- A `TSNode` represents a single node in the syntax tree. It tracks its start and end positions in the source code, as well as its relation to other nodes like its parent, siblings and children.

An Example Program

Here's an example of a simple C program that uses the Tree-sitter JSON parser.

```c
// Filename - test-json-parser.c

#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <tree_sitter/api.h>

// Declare the `tree_sitter_json` function, which is
// implemented by the `tree-sitter-json` library.
const TSLanguage *tree_sitter_json(void);

int main() {
  // Create a parser.
  TSParser *parser = ts_parser_new();

  // Set the parser's language (JSON in this case).
  ts_parser_set_language(parser, tree_sitter_json());

  // Build a syntax tree based on source code stored in a string.
  const char *source_code = "[1, null]";
  TSTree *tree = ts_parser_parse_string(
    parser,
    NULL,
    source_code,
    strlen(source_code)
  );

  // Get the root node of the syntax tree.
  TSNode root_node = ts_tree_root_node(tree);

  // Get some child nodes.
  TSNode array_node = ts_node_named_child(root_node, 0);
  TSNode number_node = ts_node_named_child(array_node, 0);

  // Check that the nodes have the expected types.
  assert(strcmp(ts_node_type(root_node), "document") == 0);
  assert(strcmp(ts_node_type(array_node), "array") == 0);
  assert(strcmp(ts_node_type(number_node), "number") == 0);

  // Check that the nodes have the expected child counts.
  assert(ts_node_child_count(root_node) == 1);
  assert(ts_node_child_count(array_node) == 5);
  assert(ts_node_named_child_count(array_node) == 2);
  assert(ts_node_child_count(number_node) == 0);

  // Print the syntax tree as an S-expression.
  char *string = ts_node_string(root_node);
  printf("Syntax tree: %s\n", string);

  // Free all of the heap-allocated memory.
  free(string);
  ts_tree_delete(tree);
  ts_parser_delete(parser);
  return 0;
}
```

This program requires three components to build:

1. The Tree-sitter C API from `tree-sitter/api.h` (requiring `tree-sitter/lib/include` in our include path)
2. The Tree-sitter library (`libtree-sitter.a`)
3. The JSON grammar's source code, which we compile directly into the binary

```
clang                                 \
  -I tree-sitter/lib/include          \
  test-json-parser.c                  \
  tree-sitter-json/src/parser.c       \
  tree-sitter/libtree-sitter.a        \
  -o test-json-parser
./test-json-parser
```

When using dynamic linking, you'll need to ensure the shared library is discoverable through `LD_LIBRARY_PATH` or your system's equivalent environment variable. Here's how to compile with dynamic linking:

```
clang                                 \
  -I tree-sitter/lib/include          \
  test-json-parser.c                  \
  tree-sitter-json/src/parser.c       \
  -ltree-sitter                       \
  -o test-json-parser
./test-json-parser
```

Basic Parsing

Providing the Code

In the example on the previous page, we parsed source code stored in a simple string using the `ts_parser_parse_string` function:

```
TSTree *ts_parser_parse_string(
  TSParser *self,
  const TSTree *old_tree,
  const char *string,
  uint32_t length
);
```

You may want to parse source code that's stored in a custom data structure, like a piece table or a rope. In this case, you can use the more general `ts_parser_parse` function:

```
TSTree *ts_parser_parse(
  TSParser *self,
  const TSTree *old_tree,
  TSInput input
);
```

The `TSInput` structure lets you provide your own function for reading a chunk of text at a given byte offset and row/column position. The function can return text encoded in either UTF-8 or UTF-16. This interface allows you to efficiently parse text that is stored in your own data structure.

```
typedef struct {
  void *payload;
  const char *(*read)(
    void *payload,
    uint32_t byte_offset,
    TSPoint position,
    uint32_t *bytes_read
  );
  TSInputEncoding encoding;
  TSDecodeFunction decode;
} TSInput;
```

If you want to decode text that is not encoded in UTF-8 or UTF-16, you can set the `decode` field of the input to your function that will decode text. The signature of the `TSDecodeFunction` is as follows:

```
typedef uint32_t (*TSDecodeFunction)(
  const uint8_t *string,
  uint32_t length,
  int32_t *code_point
);
```

> **Attention**
>
> The `TSInputEncoding` must be set to `TSInputEncodingCustom` for the `decode` function to be called.

The `string` argument is a pointer to the text to decode, which comes from the `read` function, and the `length` argument is the length of the `string`. The `code_point` argument is a pointer to an integer that represents the decoded code point, and should be written to in your `decode` callback. The function should return the number of bytes decoded.

Syntax Nodes

Tree-sitter provides a DOM-style interface for inspecting syntax trees. A syntax node's *type* is a string that indicates which grammar rule the node represents.

```
const char *ts_node_type(TSNode);
```

Syntax nodes store their position in the source code both in raw bytes and row/column coordinates. In a point, rows and columns are zero-based. The `row` field represents the number of newlines before a given position, while `column` represents the number of bytes between the position and beginning of the line.

```
uint32_t ts_node_start_byte(TSNode);
uint32_t ts_node_end_byte(TSNode);
typedef struct {
  uint32_t row;
  uint32_t column;
} TSPoint;
TSPoint ts_node_start_point(TSNode);
TSPoint ts_node_end_point(TSNode);
```

> **Note**
>
> A *newline* is considered to be a single line feed (`\n`) character.

Retrieving Nodes

Every tree has a *root node*:

```
TSNode ts_tree_root_node(const TSTree *);
```

Once you have a node, you can access the node's children:

```
uint32_t ts_node_child_count(TSNode);
TSNode ts_node_child(TSNode, uint32_t);
```

You can also access its siblings and parent:

```
TSNode ts_node_next_sibling(TSNode);
TSNode ts_node_prev_sibling(TSNode);
TSNode ts_node_parent(TSNode);
```

These methods may all return a *null node* to indicate, for example, that a node does not *have* a next sibling. You can check if a node is null:

```
bool ts_node_is_null(TSNode);
```

Named vs Anonymous Nodes

Tree-sitter produces *concrete* syntax trees — trees that contain nodes for every individual token in the source code, including things like commas and parentheses. This is important for use-cases that deal with individual tokens, like syntax highlighting. But some types of code analysis are easier to perform using an *abstract* syntax tree — a tree in which the less important details have been removed. Tree-sitter's trees support these use cases by making a distinction between *named* and *anonymous* nodes.

Consider a grammar rule like this:

```
if_statement: $ => seq("if", "(", $._expression, ")", $._statement);
```

A syntax node representing an `if_statement` in this language would have 5 children: the condition expression, the body statement, as well as the `if`, `(`, and `)` tokens. The expression and the statement would be marked as *named* nodes, because they have been given explicit names in the grammar. But the `if`, `(`, and `)` nodes would *not* be named nodes, because they are represented in the grammar as simple strings.

You can check whether any given node is named:

```
bool ts_node_is_named(TSNode);
```

When traversing the tree, you can also choose to skip over anonymous nodes by using the `_named_` variants of all of the methods described above:

```
TSNode ts_node_named_child(TSNode, uint32_t);
uint32_t ts_node_named_child_count(TSNode);
TSNode ts_node_next_named_sibling(TSNode);
TSNode ts_node_prev_named_sibling(TSNode);
```

If you use this group of methods, the syntax tree functions much like an abstract syntax tree.

Node Field Names

To make syntax nodes easier to analyze, many grammars assign unique *field names* to particular child nodes. In the creating parsers section, it's explained how to do this in your own grammars. If a syntax node has fields, you can access its children using their field name:

```
TSNode ts_node_child_by_field_name(
  TSNode self,
  const char *field_name,
  uint32_t field_name_length
);
```

Fields also have numeric ids that you can use, if you want to avoid repeated string comparisons. You can convert between strings and ids using the `TSLanguage`:

```
uint32_t ts_language_field_count(const TSLanguage *);
const char *ts_language_field_name_for_id(const TSLanguage *, TSFieldId);
TSFieldId ts_language_field_id_for_name(const TSLanguage *, const char *, uint32_t);
```

The field ids can be used in place of the name:

```
TSNode ts_node_child_by_field_id(TSNode, TSFieldId);
```

Advanced Parsing

Editing

In applications like text editors, you often need to re-parse a file after its source code has changed. Tree-sitter is designed to support this use case efficiently. There are two steps required. First, you must *edit* the syntax tree, which adjusts the ranges of its nodes so that they stay in sync with the code.

```
typedef struct {
  uint32_t start_byte;
  uint32_t old_end_byte;
  uint32_t new_end_byte;
  TSPoint start_point;
  TSPoint old_end_point;
  TSPoint new_end_point;
} TSInputEdit;

void ts_tree_edit(TSTree *, const TSInputEdit *);
```

Then, you can call `ts_parser_parse` again, passing in the old tree. This will create a new tree that internally shares structure with the old tree.

When you edit a syntax tree, the positions of its nodes will change. If you have stored any `TSNode` instances outside of the `TSTree`, you must update their positions separately, using the same `TSInputEdit` value, in order to update their cached positions.

```
void ts_node_edit(TSNode *, const TSInputEdit *);
```

This `ts_node_edit` function is *only* needed in the case where you have retrieved `TSNode` instances *before* editing the tree, and then *after* editing the tree, you want to continue to use those specific node instances. Often, you'll just want to re-fetch nodes from the edited tree, in which case `ts_node_edit` is not needed.

Multi-language Documents

Sometimes, different parts of a file may be written in different languages. For example, templating languages like EJS and ERB allow you to generate HTML by writing a mixture of HTML and another language like JavaScript or Ruby.

Tree-sitter handles these types of documents by allowing you to create a syntax tree based on the text in certain *ranges* of a file.

```
typedef struct {
  TSPoint start_point;
  TSPoint end_point;
  uint32_t start_byte;
  uint32_t end_byte;
} TSRange;

void ts_parser_set_included_ranges(
  TSParser *self,
  const TSRange *ranges,
  uint32_t range_count
);
```

For example, consider this ERB document:

```
<ul>
  <% people.each do |person| %>
    <li><%= person.name %></li>
  <% end %>
</ul>
```

Conceptually, it can be represented by three syntax trees with overlapping ranges: an ERB syntax tree, a Ruby syntax tree, and an HTML syntax tree. You could generate these syntax trees with the following code:

```c
#include <string.h>
#include <tree_sitter/api.h>

// These functions are each implemented in their own repo.
const TSLanguage *tree_sitter_embedded_template(void);
const TSLanguage *tree_sitter_html(void);
const TSLanguage *tree_sitter_ruby(void);

int main(int argc, const char **argv) {
  const char *text = argv[1];
  unsigned len = strlen(text);

  // Parse the entire text as ERB.
  TSParser *parser = ts_parser_new();
  ts_parser_set_language(parser, tree_sitter_embedded_template());
  TSTree *erb_tree = ts_parser_parse_string(parser, NULL, text, len);
  TSNode erb_root_node = ts_tree_root_node(erb_tree);

  // In the ERB syntax tree, find the ranges of the `content` nodes,
  // which represent the underlying HTML, and the `code` nodes, which
  // represent the interpolated Ruby.
  TSRange html_ranges[10];
  TSRange ruby_ranges[10];
  unsigned html_range_count = 0;
  unsigned ruby_range_count = 0;
  unsigned child_count = ts_node_child_count(erb_root_node);

  for (unsigned i = 0; i < child_count; i++) {
    TSNode node = ts_node_child(erb_root_node, i);
    if (strcmp(ts_node_type(node), "content") == 0) {
      html_ranges[html_range_count++] = (TSRange) {
        ts_node_start_point(node),
        ts_node_end_point(node),
        ts_node_start_byte(node),
        ts_node_end_byte(node),
      };
    } else {
      TSNode code_node = ts_node_named_child(node, 0);
      ruby_ranges[ruby_range_count++] = (TSRange) {
        ts_node_start_point(code_node),
        ts_node_end_point(code_node),
        ts_node_start_byte(code_node),
        ts_node_end_byte(code_node),
      };
    }
  }

  // Use the HTML ranges to parse the HTML.
  ts_parser_set_language(parser, tree_sitter_html());
  ts_parser_set_included_ranges(parser, html_ranges, html_range_count);
  TSTree *html_tree = ts_parser_parse_string(parser, NULL, text, len);
  TSNode html_root_node = ts_tree_root_node(html_tree);

  // Use the Ruby ranges to parse the Ruby.
  ts_parser_set_language(parser, tree_sitter_ruby());
  ts_parser_set_included_ranges(parser, ruby_ranges, ruby_range_count);
  TSTree *ruby_tree = ts_parser_parse_string(parser, NULL, text, len);
  TSNode ruby_root_node = ts_tree_root_node(ruby_tree);

  // Print all three trees.
  char *erb_sexp = ts_node_string(erb_root_node);
  char *html_sexp = ts_node_string(html_root_node);
  char *ruby_sexp = ts_node_string(ruby_root_node);
  printf("ERB: %s\n", erb_sexp);
  printf("HTML: %s\n", html_sexp);
  printf("Ruby: %s\n", ruby_sexp);
  return 0;
}
```

This API allows for great flexibility in how languages can be composed. Tree-sitter is not responsible for mediating the interactions between languages. Instead, you are free to do that using arbitrary application-specific logic.

Concurrency

Tree-sitter supports multi-threaded use cases by making syntax trees very cheap to copy.

```c
TSTree *ts_tree_copy(const TSTree *);
```

Internally, copying a syntax tree just entails incrementing an atomic reference count. Conceptually, it provides you a new tree which you can freely query, edit, reparse, or delete on a new thread while continuing to use the original tree on a different thread.

> **Danger**
>
> Individual `TSTree` instances are *not* thread safe; you must copy a tree if you want to use it on multiple threads simultaneously.

Walking Trees with Tree Cursors

You can access every node in a syntax tree using the `TSNode` APIs described earlier, but if you need to access a large number of nodes, the fastest way to do so is with a *tree cursor*. A cursor is a stateful object that allows you to walk a syntax tree with maximum efficiency.

> **Note**
>
> The given input node is considered the root of the cursor, and the cursor cannot walk outside this node. Going to the parent or any sibling of the root node will always return `false`.
>
> This has no unexpected effects if the given input node is the actual `root` node of the tree, but is something to keep in mind when using cursors constructed with a node that is not the `root` node.

You can initialize a cursor from any node:

```
TSTreeCursor ts_tree_cursor_new(TSNode);
```

You can move the cursor around the tree:

```
bool ts_tree_cursor_goto_first_child(TSTreeCursor *);
bool ts_tree_cursor_goto_next_sibling(TSTreeCursor *);
bool ts_tree_cursor_goto_parent(TSTreeCursor *);
```

These methods return `true` if the cursor successfully moved and `false` if there was no node to move to.

You can always retrieve the cursor's current node, as well as the field name that is associated with the current node.

```
TSNode ts_tree_cursor_current_node(const TSTreeCursor *);
const char *ts_tree_cursor_current_field_name(const TSTreeCursor *);
TSFieldId ts_tree_cursor_current_field_id(const TSTreeCursor *);
```

Pattern Matching with Queries

Code analysis often requires finding specific patterns in source code. Tree-sitter provides a simple pattern-matching language for this purpose, similar to what's used in its unit test system. This allows you to express and search for code structures without writing complex parsing logic.

Query Syntax

A *query* consists of one or more *patterns*, where each pattern is an S-expression that matches a certain set of nodes in a syntax tree. The expression to match a given node consists of a pair of parentheses containing two things: the node's type, and optionally, a series of other S-expressions that match the node's children. For example, this pattern would match any `binary_expression` node whose children are both `number_literal` nodes:

```
(binary_expression (number_literal) (number_literal))
```

Children can also be omitted. For example, this would match any `binary_expression` where at least *one* of child is a `string_literal` node:

```
(binary_expression (string_literal))
```

Fields

In general, it's a good idea to make patterns more specific by specifying field names associated with child nodes. You do this by prefixing a child pattern with a field name followed by a colon. For example, this pattern would match an `assignment_expression` node where the `left` child is a `member_expression` whose `object` is a `call_expression`.

```
(assignment_expression
  left: (member_expression
    object: (call_expression)))
```

Negated Fields

You can also constrain a pattern so that it only matches nodes that *lack* a certain field. To do this, add a field name prefixed by a `!` within the parent pattern. For example, this pattern would match a class declaration with no type parameters:

```
(class_declaration
  name: (identifier) @class_name
  !type_parameters)
```

Anonymous Nodes

The parenthesized syntax for writing nodes only applies to named nodes. To match specific anonymous nodes, you write their name between double quotes. For example, this pattern would match any `binary_expression` where the operator is `!=` and the right side is `null`:

```
(binary_expression
  operator: "!="
  right: (null))
```

Special Nodes

The Wildcard Node

A wildcard node is represented with an underscore (`_`), it matches any node. This is similar to `.` in regular expressions. There are two types, `(_)` will match any named node, and `_` will match any named or anonymous node.

For example, this pattern would match any node inside a call:

```
(call (_) @call.inner)
```

The `ERROR` Node

When the parser encounters text it does not recognize, it represents this node as `(ERROR)` in the syntax tree. These error nodes can be queried just like normal nodes:

```
(ERROR) @error-node
```

The `MISSING` Node

If the parser is able to recover from erroneous text by inserting a missing token and then reducing, it will insert that missing node in the final tree so long as that tree has the lowest error cost. These missing nodes appear as seemingly normal nodes in the tree, but they are zero tokens wide, and are internally represented as a property of the actual terminal node that was inserted, instead of being

its own kind of node, like the `ERROR` node. These special missing nodes can be queried using `(MISSING)`:

```
(MISSING) @missing-node
```

This is useful when attempting to detect all syntax errors in a given parse tree, since these missing node are not captured by `(ERROR)` queries. Specific missing node types can also be queried:

```
(MISSING identifier) @missing-identifier
(MISSING ";") @missing-semicolon
```

Supertype Nodes

Some node types are marked as *supertypes* in a grammar. A supertype is a node type that contains multiple subtypes. For example, in the JavaScript grammar example, `expression` is a supertype that can represent any kind of expression, such as a `binary_expression`, `call_expression`, or `identifier`. You can use supertypes in queries to match any of their subtypes, rather than having to list out each subtype individually. For example, this pattern would match any kind of expression, even though it's not a visible node in the syntax tree:

```
(expression) @any-expression
```

To query specific subtypes of a supertype, you can use the syntax `supertype/subtype`. For example, this pattern would match a `binary_expression` only if it is a child of `expression`:

```
(expression/binary_expression) @binary-expression
```

This also applies to anonymous nodes. For example, this pattern would match `"()"` only if it is a child of `expression`:

```
(expression/"()") @empty-expression
```

Operators

Capturing Nodes

When matching patterns, you may want to process specific nodes within the pattern. Captures allow you to associate names with specific nodes in a pattern, so that you can later refer to those nodes by those names. Capture names are written *after* the nodes that they refer to, and start with an `@` character.

For example, this pattern would match any assignment of a `function` to an `identifier`, and it would associate the name `the-function-name` with the identifier:

```
(assignment_expression
  left: (identifier) @the-function-name
  right: (function))
```

And this pattern would match all method definitions, associating the name `the-method-name` with the method name, `the-class-name` with the containing class name:

```
(class_declaration
  name: (identifier) @the-class-name
  body: (class_body
    (method_definition
      name: (property_identifier) @the-method-name)))
```

Quantification Operators

You can match a repeating sequence of sibling nodes using the postfix `+` and `*` *repetition* operators, which work analogously to the `+` and `*` operators in regular expressions. The `+` operator matches *one or more* repetitions of a pattern, and the `*` operator matches *zero or more*.

For example, this pattern would match a sequence of one or more comments:

```
(comment)+
```

This pattern would match a class declaration, capturing all of the decorators if any were present:

```
(class_declaration
  (decorator)* @the-decorator
  name: (identifier) @the-name)
```

You can also mark a node as optional using the `?` operator. For example, this pattern would match all function calls, capturing a string argument if one was present:

```
(call_expression
  function: (identifier) @the-function
  arguments: (arguments (string)? @the-string-arg))
```

Grouping Sibling Nodes

You can also use parentheses for grouping a sequence of *sibling* nodes. For example, this pattern would match a comment followed by a function declaration:

```
(
  (comment)
  (function_declaration)
)
```

Any of the quantification operators mentioned above (`+`, `*`, and `?`) can also be applied to groups. For example, this pattern would match a comma-separated series of numbers:

```
(
  (number)
  ("," (number))*
)
```

Alternations

An alternation is written as a pair of square brackets (`[]`) containing a list of alternative patterns. This is similar to *character classes* from regular expressions (`[abc]` matches either a, b, or c).

For example, this pattern would match a call to either a variable or an object property. In the case of a variable, capture it as `@function`, and in the case of a property, capture it as `@method`:

```
(call_expression
  function: [
    (identifier) @function
    (member_expression
      property: (property_identifier) @method)
  ])
```

This pattern would match a set of possible keyword tokens, capturing them as `@keyword`:

```
[
  "break"
  "delete"
  "else"
  "for"
  "function"
  "if"
  "return"
  "try"
  "while"
] @keyword
```

Anchors

The anchor operator, `.`, is used to constrain the ways in which child patterns are matched. It has different behaviors depending on where it's placed inside a query.

When `.` is placed before the *first* child within a parent pattern, the child will only match when it is the first named node in the parent. For example, the below pattern matches a given `array` node at most once, assigning the `@the-element` capture to the first `identifier` node in the parent `array`:

```
(array . (identifier) @the-element)
```

Without this anchor, the pattern would match once for every identifier in the array, with `@the-element` bound to each matched identifier.

Similarly, an anchor placed after a pattern's *last* child will cause that child pattern to only match nodes that are the last named child of their parent. The below pattern matches only nodes that are the last named child within a `block`.

```
(block (_) @last-expression .)
```

Finally, an anchor *between* two child patterns will cause the patterns to only match nodes that are immediate siblings. The pattern below, given a long dotted name like `a.b.c.d`, will only match pairs of consecutive identifiers: `a, b`, `b, c`, and `c, d`.

```
(dotted_name
  (identifier) @prev-id
  .
  (identifier) @next-id)
```

Without the anchor, non-consecutive pairs like `a, c` and `b, d` would also be matched.

The restrictions placed on a pattern by an anchor operator ignore anonymous nodes.

Predicates

You can also specify arbitrary metadata and conditions associated with a pattern by adding *predicate* S-expressions anywhere within your pattern. Predicate S-expressions start with a *predicate name* beginning with a `#` character, and ending with a `?` character. After that, they can contain an arbitrary number of `@`-prefixed capture names or strings.

Tree-sitter's CLI supports the following predicates by default:

The `eq?` predicate

This family of predicates allows you to match against a single capture or string value.

The first argument to this predicate must be a capture, but the second can be either a capture to compare the two captures' text, or a string to compare first capture's text against.

The base predicate is `#eq?`, but its complement, `#not-eq?`, can be used to *not* match a value. Additionally, you can prefix either of these with `any-` to match if *any* of the nodes match the predicate. This is only useful when dealing with quantified captures, as by default a quantified capture will only match if *all* the captured nodes match the predicate.

Thus, there are four predicates in total:

- `#eq?`
- `#not-eq?`
- `#any-eq?`
- `#any-not-eq?`

Consider the following example targeting C:

```
((identifier) @variable.builtin
  (#eq? @variable.builtin "self"))
```

This pattern would match any identifier that is `self`.

Now consider the following example:

```
(
  (pair
    key: (property_identifier) @key-name
    value: (identifier) @value-name)
  (#eq? @key-name @value-name)
)
```

This pattern would match key-value pairs where the `value` is an identifier with the same text as the key (meaning they are the same):

As mentioned earlier, the `any-` prefix is meant for use with quantified captures. Here's an example finding an empty comment within a group of comments:

```
((comment)+ @comment.empty
  (#any-eq? @comment.empty "//"))
```

The `match?` predicate

These predicates are similar to the `eq?` predicates, but they use regular expressions to match against the capture's text instead of string comparisons.

The first argument must be a capture, and the second must be a string containing a regular expression.

Like the `eq?` predicate family, we can tack on `not-` to the beginning of the predicate to negate the match, and `any-` to match if *any* of the nodes in a quantified capture match the predicate.

This pattern matches identifiers written in `SCREAMING_SNAKE_CASE`.

```
((identifier) @constant
  (#match? @constant "^[A-Z][A-Z_]+"))
```

This query identifies documentation comments in C that begin with three forward slashes (`///`).

```
((comment)+ @comment.documentation
  (#match? @comment.documentation "^///\\s+.*"))
```

This query finds C code embedded in Go comments that appear just before a "C" import statement. These are known as `Cgo` comments and are used to inject C code into Go programs.

```
((comment)+ @injection.content
  .
  (import_declaration
    (import_spec path: (interpreted_string_literal) @_import_c))
  (#eq? @_import_c "\"C\"")
  (#match? @injection.content "^//"))
```

The `any-of?` predicate

The `any-of?` predicate allows you to match a capture against multiple strings, and will match if the capture's text is equal to any of the strings.

The query below will match any of the builtin variables in JavaScript.

```
((identifier) @variable.builtin
  (#any-of? @variable.builtin
        "arguments"
        "module"
        "console"
        "window"
        "document"))
```

The `is?` predicate

The `is?` predicate allows you to assert that a capture has a given property. This isn't widely used, but the CLI uses it to determine whether a given node is a local variable or not, for example:

```
((identifier) @variable.builtin
  (#match? @variable.builtin "^(arguments|module|console|window|document)$")
  (#is-not? local))
```

This pattern would match any builtin variable that is not a local variable, because the `#is-not? local` predicate is used.

Directives

Similar to predicates, directives are a way to associate arbitrary metadata with a pattern. The only difference between predicates and directives is that directives end in a `!` character instead of `?` character.

Tree-sitter's CLI supports the following directives by default:

The `set!` directive

This directive allows you to associate key-value pairs with a pattern. The key and value can be any arbitrary text that you see fit.

```
((comment) @injection.content
  (#match? @injection.content "/[*\/][!*\/]<?[^a-zA-Z]")
  (#set! injection.language "doxygen"))
```

This pattern would match any comment that contains a Doxygen-style comment, and then sets the `injection.language` key to `"doxygen"`. Programmatically, when iterating the captures of this pattern, you can access this property to then parse the comment with the Doxygen parser.

The `#select-adjacent!` directive

The `#select-adjacent!` directive allows you to filter the text associated with a capture so that only nodes adjacent to another capture are preserved. It takes two arguments, both of which are capture names.

The `#strip!` directive

The `#strip!` directive allows you to remove text from a capture. It takes two arguments: the first is the capture to strip text from, and the second is a regular expression to match against the text. Any text matched by the regular expression will be removed from the text associated with the capture.

For an example on the `#select-adjacent!` and `#strip!` directives, view the code navigation documentation.

Recap

To recap about the predicates and directives Tree-Sitter's bindings support:

- `#eq?` checks for a direct match against a capture or string
- `#match?` checks for a match against a regular expression
- `#any-of?` checks for a match against a list of strings
- `#is?` checks for a property on a capture
- Adding `not-` to the beginning of these predicates will negate the match
- By default, a quantified capture will only match if *all* the nodes match the predicate

- Adding `any-` before the `eq` or `match` predicates will instead match if any of the nodes match the predicate
- `#set!` associates key-value pairs with a pattern
- `#select-adjacent!` filters the text associated with a capture so that only nodes adjacent to another capture are preserved
- `#strip!` removes text from a capture

> **Info**
>
> Predicates and directives are not handled directly by the Tree-sitter C library. They are just exposed in a structured form so that higher-level code can perform the filtering. However, higher-level bindings to Tree-sitter like the Rust Crate or the WebAssembly binding do implement a few common predicates like those explained above. In the future, more "standard" predicates and directives may be added.

The Query API

Create a query by specifying a string containing one or more patterns:

```
TSQuery *ts_query_new(
  const TSLanguage *language,
  const char *source,
  uint32_t source_len,
  uint32_t *error_offset,
  TSQueryError *error_type
);
```

If there is an error in the query, then the `error_offset` argument will be set to the byte offset of the error, and the `error_type` argument will be set to a value that indicates the type of error:

```
typedef enum {
  TSQueryErrorNone = 0,
  TSQueryErrorSyntax,
  TSQueryErrorNodeType,
  TSQueryErrorField,
  TSQueryErrorCapture,
} TSQueryError;
```

The `TSQuery` value is immutable and can be safely shared between threads. To execute the query, create a `TSQueryCursor`, which carries the state needed for processing the queries. The query cursor should not be shared between threads, but can be reused for many query executions.

```
TSQueryCursor *ts_query_cursor_new(void);
```

You can then execute the query on a given syntax node:

```
void ts_query_cursor_exec(TSQueryCursor *, const TSQuery *, TSNode);
```

You can then iterate over the matches:

```
typedef struct {
  TSNode node;
  uint32_t index;
} TSQueryCapture;

typedef struct {
  uint32_t id;
  uint16_t pattern_index;
  uint16_t capture_count;
  const TSQueryCapture *captures;
} TSQueryMatch;

bool ts_query_cursor_next_match(TSQueryCursor *, TSQueryMatch *match);
```

This function will return `false` when there are no more matches. Otherwise, it will populate the `match` with data about which pattern matched and which nodes were captured.

Static Node Types

In languages with static typing, it can be helpful for syntax trees to provide specific type information about individual syntax nodes. Tree-sitter makes this information available via a generated file called `node-types.json`. This *node types* file provides structured data about every possible syntax node in a grammar.

You can use this data to generate type declarations in statically-typed programming languages.

The node types file contains an array of objects, each of which describes a particular type of syntax node using the following entries:

Basic Info

Every object in this array has these two entries:

- `"type"` — A string that indicates, which grammar rule the node represents. This corresponds to the `ts_node_type` function described here.
- `"named"` — A boolean that indicates whether this kind of node corresponds to a rule name in the grammar or just a string literal. See here for more info.

Examples:

```
{
  "type": "string_literal",
  "named": true
}
{
  "type": "+",
  "named": false
}
```

Together, these two fields constitute a unique identifier for a node type; no two top-level objects in the `node-types.json` should have the same values for both `"type"` and `"named"`.

Internal Nodes

Many syntax nodes can have *children*. The node type object describes the possible children that a node can have using the following entries:

- `"fields"` — An object that describes the possible fields that the node can have. The keys of this object are field names, and the values are *child type* objects, described below.
- `"children"` — Another *child type* object that describes all the node's possible *named* children *without* fields.

A *child type* object describes a set of child nodes using the following entries:

- `"required"` — A boolean indicating whether there is always *at least one* node in this set.
- `"multiple"` — A boolean indicating whether there can be *multiple* nodes in this set.
- `"types"` - An array of objects that represent the possible types of nodes in this set. Each object has two keys: `"type"` and `"named"`, whose meanings are described above.

Example with fields:

```
{
  "type": "method_definition",
  "named": true,
  "fields": {
    "body": {
      "multiple": false,
      "required": true,
      "types": [{ "type": "statement_block", "named": true }]
    },
    "decorator": {
      "multiple": true,
      "required": false,
      "types": [{ "type": "decorator", "named": true }]
    },
    "name": {
      "multiple": false,
      "required": true,
      "types": [
        { "type": "computed_property_name", "named": true },
        { "type": "property_identifier", "named": true }
      ]
    },
    "parameters": {
      "multiple": false,
      "required": true,
      "types": [{ "type": "formal_parameters", "named": true }]
    }
  }
}
```

Example with children:

```
{
  "type": "array",
  "named": true,
  "fields": {},
  "children": {
    "multiple": true,
    "required": false,
    "types": [
      { "type": "_expression", "named": true },
      { "type": "spread_element", "named": true }
    ]
  }
}
```

Supertype Nodes

In Tree-sitter grammars, there are usually certain rules that represent abstract *categories* of syntax nodes (e.g. "expression", "type", "declaration"). In the `grammar.js` file, these are often written as hidden rules whose definition is a simple `choice` where each member is just a single symbol.

Normally, hidden rules are not mentioned in the node types file, since they don't appear in the syntax tree. But if you add a hidden rule to the grammar's `supertypes` list, then it *will* show up in the node types file, with the following special entry:

- `"subtypes"` — An array of objects that specify the *types* of nodes that this 'supertype' node can wrap.

Example:

```
{
  "type": "_declaration",
  "named": true,
  "subtypes": [
    { "type": "class_declaration", "named": true },
    { "type": "function_declaration", "named": true },
    { "type": "generator_function_declaration", "named": true },
    { "type": "lexical_declaration", "named": true },
    { "type": "variable_declaration", "named": true }
  ]
}
```

Supertype nodes will also appear elsewhere in the node types file, as children of other node types, in a way that corresponds with how the supertype rule was used in the grammar. This can make the node types much shorter and easier to read, because a single supertype will take the place of multiple subtypes.

Example:

```
{
  "type": "export_statement",
  "named": true,
  "fields": {
    "declaration": {
      "multiple": false,
      "required": false,
      "types": [{ "type": "_declaration", "named": true }]
    },
    "source": {
      "multiple": false,
      "required": false,
      "types": [{ "type": "string", "named": true }]
    }
  }
}
```

ABI versions

Parsers generated with tree-sitter have an associated ABI version, which establishes hard compatibility boundaries between the generated parser and the tree-sitter library.

A given version of the tree-sitter library is only able to load parsers generated with supported ABI versions:

tree-sitter version	Min parser ABI version	Max parser ABI version
0.14	9	9
>=0.15.0, <=0.15.7	9	10
>=0.15.8, <=0.16	9	11
0.17, 0.18	9	12
>=0.19, <=0.20.2	13	13
>=0.20.3, <=0.24	13	14
>=0.25	13	15

By default, the tree-sitter CLI will generate parsers using the latest available ABI for that version, but an older ABI (supported by the CLI) can be selected by passing the `--abi` option to the `generate` command.

Note that the ABI version range supported by the CLI can be smaller than for the library: When a new ABI version is released, older versions will be phased out over a deprecation period, which starts with no longer being able to generate parsers with the oldest ABI version.

Creating parsers

Developing Tree-sitter grammars can have a difficult learning curve, but once you get the hang of it, it can be fun and even zen-like. This document will help you to get started and to develop a useful mental model.

Getting Started

Dependencies

To develop a Tree-sitter parser, there are two dependencies that you need to install:

- **A JavaScript runtime** — Tree-sitter grammars are written in JavaScript, and Tree-sitter uses a JavaScript runtime (the default being Node.js) to interpret JavaScript files. It requires this runtime command (default: `node`) to be in one of the directories in your `PATH`.
- **A C Compiler** — Tree-sitter creates parsers that are written in C. To run and test these parsers with the `tree-sitter parse` or `tree-sitter test` commands, you must have a C/C++ compiler installed. Tree-sitter will try to look for these compilers in the standard places for each platform.

Installation

To create a Tree-sitter parser, you need to use the `tree-sitter` CLI. You can install the CLI in a few different ways:

- Build the `tree-sitter-cli` Rust crate from source using `cargo`, the Rust package manager. This works on any platform. See the contributing docs for more information.
- Install the `tree-sitter-cli` Rust crate from crates.io using `cargo`. You can do so by running the following command: `cargo install tree-sitter-cli --locked`
- Install the `tree-sitter-cli` Node.js module using `npm`, the Node package manager. This approach is fast, but it only works on certain platforms, because it relies on pre-built binaries.
- Download a binary for your platform from the latest GitHub release, and put it into a directory on your `PATH`.

Project Setup

The preferred convention is to name the parser repository "tree-sitter-" followed by the name of the language, in lowercase.

```
mkdir tree-sitter-${LOWER_PARSER_NAME}
cd tree-sitter-${LOWER_PARSER_NAME}
```

> **Note**
>
> The `LOWER_` prefix here means the "lowercase" name of the language.

Init

Once you've installed the `tree-sitter` CLI tool, you can start setting up your project, which will allow your parser to be used from multiple languages.

```
# This will prompt you for input
tree-sitter init
```

The `init` command will create a bunch of files in the project. There should be a file called `grammar.js` with the following contents:

```
/**
 * @file PARSER_DESCRIPTION
 * @author PARSER_AUTHOR_NAME PARSER_AUTHOR_EMAIL
 * @license PARSER_LICENSE
 */

/// <reference types="tree-sitter-cli/dsl" />
// @ts-check

export default grammar({
  name: 'LOWER_PARSER_NAME',

  rules: {
    // TODO: add the actual grammar rules
    source_file: $ => 'hello'
  }
});
```

> **Info**
>
> The placeholders shown above would be replaced with the corresponding data you provided in the `init` sub-command's prompts.

To learn more about this command, check the reference page.

Generate

Next, run the following command:

```
tree-sitter generate
```

This will generate the C code required to parse this trivial language.

You can test this parser by creating a source file with the contents "hello" and parsing it:

```
echo 'hello' > example-file
tree-sitter parse example-file
```

Alternatively, in Windows PowerShell:

```
"hello" | Out-File example-file -Encoding utf8
tree-sitter parse example-file
```

This should print the following:

```
(source_file [0, 0] - [1, 0])
```

You now have a working parser.

Finally, look back at the triple-slash and `@ts-check` comments in `grammar.js`; these tell your editor to provide documentation and type information as you edit your grammar. For these to work, you must download Tree-sitter's TypeScript API from npm into a `node_modules` directory in your project:

```
npm install # or your package manager of choice
```

To learn more about this command, check the reference page.

The Grammar DSL

The following is a complete list of built-in functions you can use in your `grammar.js` to define rules. Use-cases for some of these functions will be explained in more detail in later sections.

- **Symbols (the `$` object)** — Every grammar rule is written as a JavaScript function that takes a parameter conventionally called `$`. The syntax `$.identifier` is how you refer to another grammar symbol within a rule. Names starting with `$.MISSING` or `$.UNEXPECTED` should be avoided as they have special meaning for the `tree-sitter test` command.

- **String and Regex literals** — The terminal symbols in a grammar are described using JavaScript strings and regular expressions. Of course during parsing, Tree-sitter does not actually use JavaScript's regex engine to evaluate these regexes; it generates its own regex-matching logic based on the Rust regex syntax as part of each parser. Regex literals are just used as a convenient way of writing regular expressions in your grammar. You can use Rust regular expressions in your grammar DSL through the `RustRegex` class. Simply pass your regex pattern as a string:

  ```
  new RustRegex('(?i)[a-z_][a-z0-9_]*') // matches a simple identifier
  ```

  Unlike JavaScript's builtin `RegExp` class, which takes a pattern and flags as separate arguments, `RustRegex` only accepts a single pattern string. While it doesn't support separate flags, you can use inline flags within the pattern itself. For more details about Rust's regex syntax and capabilities, check out the Rust regex documentation.

  > **Note**
  >
  > Only a subset of the Regex engine is actually supported. This is due to certain features like lookahead and lookaround assertions not feasible to use in an LR(1) grammar, as well as certain flags being unnecessary for tree-sitter. However, plenty of features are supported by default:
  >
  > - Character classes
  > - Character ranges
  > - Character sets
  > - Quantifiers
  > - Alternation
  > - Grouping
  > - Unicode character escapes
  > - Unicode property escapes

- **Sequences : `seq(rule1, rule2, ...)`** — This function creates a rule that matches any number of other rules, one after another. It is analogous to simply writing multiple symbols next to each other in EBNF notation.

- **Alternatives : `choice(rule1, rule2, ...)`** — This function creates a rule that matches *one* of a set of possible rules. The order of the arguments does not matter. This is analogous to the `|` (pipe) operator in EBNF notation.

- **Repetitions : `repeat(rule)`** — This function creates a rule that matches *zero-or-more* occurrences of a given rule. It is analogous to the `{x}` (curly brace) syntax in EBNF notation.

- **Repetitions : `repeat1(rule)`** — This function creates a rule that matches *one-or-more* occurrences of a given rule. The previous `repeat` rule is implemented in `repeat1` but is included because it is very commonly used.

- **Options : `optional(rule)`** — This function creates a rule that matches *zero or one* occurrence of a given rule. It is analogous to the `[x]` (square bracket) syntax in EBNF notation.

- **Precedence : `prec(number, rule)`** — This function marks the given rule with a numerical precedence, which will be used to resolve *LR(1) Conflicts* at parser-generation time. When two rules overlap in a way that represents either a true ambiguity or a *local* ambiguity given one token of lookahead, Tree-sitter will try to resolve the conflict by matching the rule with the higher precedence. The default precedence of all rules is zero. This works similarly to the precedence directives in Yacc grammars.

  This function can also be used to assign lexical precedence to a given token, but it must be wrapped in a `token` call, such as `token(prec(1, 'foo'))`. This reads as "the token `foo` has a lexical precedence of 1". The purpose of lexical precedence is to solve the issue where multiple tokens can match the same set of characters, but one token should be preferred over the other. See Lexical Precedence vs Parse Precedence for a more detailed explanation.

- **Left Associativity : `prec.left([number], rule)`** — This function marks the given rule as left-associative (and optionally applies a numerical precedence). When an LR(1) conflict arises in which all the rules have the same numerical precedence, Tree-sitter will consult the rules' associativity. If there is a left-associative rule, Tree-sitter will prefer matching a rule that ends *earlier*. This works similarly to associativity directives in Yacc grammars.

- **Right Associativity : `prec.right([number], rule)`** — This function is like `prec.left`, but it instructs Tree-sitter to prefer matching a rule that ends *later*.

- **Dynamic Precedence : `prec.dynamic(number, rule)`** — This function is similar to `prec`, but the given numerical precedence is applied at *runtime* instead of at parser generation time. This is only necessary when handling a conflict dynamically using the `conflicts` field in the

grammar, and when there is a genuine *ambiguity*: multiple rules correctly match a given piece of code. In that event, Tree-sitter compares the total dynamic precedence associated with each rule, and selects the one with the highest total. This is similar to dynamic precedence directives in Bison grammars.

- **Tokens :** `token(rule)` — This function marks the given rule as producing only a single token. Tree-sitter's default is to treat each String or RegExp literal in the grammar as a separate token. Each token is matched separately by the lexer and returned as its own leaf node in the tree. The `token` function allows you to express a complex rule using the functions described above (rather than as a single regular expression) but still have Tree-sitter treat it as a single token. The token function will only accept terminal rules, so `token($.foo)` will not work. You can think of it as a shortcut for squashing complex rules of strings or regexes down to a single token.

- **Immediate Tokens :** `token.immediate(rule)` — Usually, whitespace (and any other extras, such as comments) is optional before each token. This function means that the token will only match if there is no whitespace.

- **Aliases :** `alias(rule, name)` — This function causes the given rule to *appear* with an alternative name in the syntax tree. If `name` is a *symbol*, as in `alias($.foo, $.bar)`, then the aliased rule will *appear* as a named node called `bar`. And if `name` is a *string literal*, as in `alias($.foo, 'bar')`, then the aliased rule will appear as an anonymous node, as if the rule had been written as the simple string.

- **Field Names :** `field(name, rule)` — This function assigns a *field name* to the child node(s) matched by the given rule. In the resulting syntax tree, you can then use that field name to access specific children.

- **Reserved Keywords :** `reserved(wordset, rule)` — This function will override the global reserved word set with the one passed into the `wordset` parameter. This is useful for contextual keywords, such as `if` in JavaScript, which cannot be used as a variable name in most contexts, but can be used as a property name.

In addition to the `name` and `rules` fields, grammars have a few other optional public fields that influence the behavior of the parser. Each of these fields is a function that accepts the grammar object (`$`) as its only parameter, like the grammar rules themselves. These fields are:

- `extras` — an array of tokens that may appear *anywhere* in the language. This is often used for whitespace and comments. The default value of `extras` is to accept whitespace. To control whitespace explicitly, specify `extras: $ => []` in your grammar. See the section on using extras for more details.

- `inline` — an array of rule names that should be automatically *removed* from the grammar by replacing all of their usages with a copy of their definition. This is useful for rules that are used in multiple places but for which you *don't* want to create syntax tree nodes at runtime.

- `conflicts` — an array of arrays of rule names. Each inner array represents a set of rules that's involved in an *LR(1) conflict* that is *intended to exist* in the grammar. When these conflicts occur at runtime, Tree-sitter will use the GLR algorithm to explore all the possible interpretations. If *multiple* parses end up succeeding, Tree-sitter will pick the subtree whose corresponding rule has the highest total *dynamic precedence*.

- `externals` — an array of token names which can be returned by an *external scanner*. External scanners allow you to write custom C code which runs during the lexing process to handle lexical rules (e.g. Python's indentation tokens) that cannot be described by regular expressions.

- `precedences` — an array of arrays of strings, where each array of strings defines named precedence levels in descending order. These names can be used in the `prec` functions to define precedence relative only to other names in the array, rather than globally. Can only be used with parse precedence, not lexical precedence.

- `word` — the name of a token that will match keywords to the keyword extraction optimization.

- `supertypes` — an array of rule names which should be considered to be 'supertypes' in the generated *node types* file. Supertype rules are automatically hidden from the parse tree, regardless of whether their names start with an underscore. The main use case for supertypes is to group together multiple different kinds of nodes under a single abstract category, such as "expression" or "declaration". See the section on `using supertypes` for more details.

- `reserved` — similar in structure to the main `rules` property, an object of reserved word sets associated with an array of reserved rules. The reserved rule in the array must be a terminal token meaning it must be a string, regex, token, or terminal rule. The reserved rule must also exist and be used in the grammar, specifying arbitrary tokens will not work. The *first* reserved word set in the object is the global word set, meaning it applies to every rule in every parse state. However, certain keywords are contextual, depending on the rule. For example, in JavaScript, keywords are typically not allowed as ordinary variables, however, they *can* be used as a property name. In this situation, the `reserved` function would be used, and the word set to pass in would be the name of the word set that is declared in the `reserved` object that corresponds to an empty array, signifying *no* keywords are reserved.

Writing the Grammar

Writing a grammar requires creativity. There are an infinite number of CFGs (context-free grammars) that can be used to describe any given language. To produce a good Tree-sitter parser, you need to create a grammar with two important properties:

1. **An intuitive structure** — Tree-sitter's output is a concrete syntax tree; each node in the tree corresponds directly to a terminal or non-terminal symbol in the grammar. So to produce an easy-to-analyze tree, there should be a direct correspondence between the symbols in your grammar and the recognizable constructs in the language. This might seem obvious, but it is very different from the way that context-free grammars are often written in contexts like language specifications or Yacc/Bison parsers.

2. **A close adherence to LR(1)** — Tree-sitter is based on the GLR parsing algorithm. This means that while it can handle any context-free grammar, it works most efficiently with a class of context-free grammars called LR(1) Grammars. In this respect, Tree-sitter's grammars are similar to (but less restrictive than) Yacc and Bison grammars, but *different* from ANTLR grammars, Parsing Expression Grammars, or the ambiguous grammars commonly used in language specifications.

It's unlikely that you'll be able to satisfy these two properties just by translating an existing context-free grammar directly into Tree-sitter's grammar format. There are a few kinds of adjustments that are often required. The following sections will explain these adjustments in more depth.

The First Few Rules

It's usually a good idea to find a formal specification for the language you're trying to parse. This specification will most likely contain a context-free grammar. As you read through the rules of this CFG, you will probably discover a complex and cyclic graph of relationships. It might be unclear how you should navigate this graph as you define your grammar.

Although languages have very different constructs, their constructs can often be categorized in to similar groups like *Declarations*, *Definitions*, *Statements*, *Expressions*, *Types* and *Patterns*. In writing your grammar, a good first step is to create just enough structure to include all of these basic *groups* of symbols. For a language like Go, you might start with something like this:

```
{
  // ...

  rules: {
    source_file: $ => repeat($._definition),

    _definition: $ => choice(
      $.function_definition
      // TODO: other kinds of definitions
    ),

    function_definition: $ => seq(
      'func',
      $.identifier,
      $.parameter_list,
      $._type,
      $.block
    ),

    parameter_list: $ => seq(
      '(',
       // TODO: parameters
      ')'
    ),

    _type: $ => choice(
      'bool'
      // TODO: other kinds of types
    ),

    block: $ => seq(
      '{',
      repeat($._statement),
      '}'
    ),

    _statement: $ => choice(
      $.return_statement
      // TODO: other kinds of statements
    ),

    return_statement: $ => seq(
      'return',
      $.expression,
      ';'
    ),

    expression: $ => choice(
      $.identifier,
      $.number
      // TODO: other kinds of expressions
    ),

    identifier: $ => /[a-z]+/,

    number: $ => /\d+/
  }
}
```

One important fact to know up front is that the start rule for the grammar is the first property in the `rules` object. In the example above, that would correspond to `source_file`, but it can be named anything.

Some details of this grammar will be explained in more depth later on, but if you focus on the `TODO` comments, you can see that the overall strategy is *breadth-first*. Notably, this initial skeleton does not need to directly match an exact subset of the context-free grammar in the language specification. It just needs to touch on the major groupings of rules in as simple and obvious a way as possible.

With this structure in place, you can now freely decide what part of the grammar to flesh out next. For example, you might decide to start with *types*. One-by-one, you could define the rules for writing basic types and composing them into more complex types:

```
{
  // ...

  _type: $ => choice(
    $.primitive_type,
    $.array_type,
    $.pointer_type
  ),

  primitive_type: $ => choice(
    'bool',
    'int'
  ),

  array_type: $ => seq(
    '[',
    ']',
    $._type
  ),

  pointer_type: $ => seq(
    '*',
    $._type
  )
}
```

After developing the *type* sublanguage a bit further, you might decide to switch to working on *statements* or *expressions* instead. It's often useful to check your progress by trying to parse some real code using `tree-sitter parse`.

And remember to add tests for each rule in your `test/corpus` folder!

Structuring Rules Well

Imagine that you were just starting work on the Tree-sitter JavaScript parser. Naively, you might try to directly mirror the structure of the ECMAScript Language Spec. To illustrate the problem with this approach, consider the following line of code:

```
return x + y;
```

According to the specification, this line is a `ReturnStatement`, the fragment `x + y` is an `AdditiveExpression`, and `x` and `y` are both `IdentifierReferences`. The relationship between these constructs is captured by a complex series of production rules:

```
ReturnStatement          ->  'return' Expression
Expression               ->  AssignmentExpression
AssignmentExpression     ->  ConditionalExpression
ConditionalExpression    ->  LogicalORExpression
LogicalORExpression      ->  LogicalANDExpression
LogicalANDExpression     ->  BitwiseORExpression
BitwiseORExpression      ->  BitwiseXORExpression
BitwiseXORExpression     ->  BitwiseANDExpression
BitwiseANDExpression     ->  EqualityExpression
EqualityExpression       ->  RelationalExpression
RelationalExpression     ->  ShiftExpression
ShiftExpression          ->  AdditiveExpression
AdditiveExpression       ->  MultiplicativeExpression
MultiplicativeExpression ->  ExponentiationExpression
ExponentiationExpression ->  UnaryExpression
UnaryExpression          ->  UpdateExpression
UpdateExpression         ->  LeftHandSideExpression
LeftHandSideExpression   ->  NewExpression
NewExpression            ->  MemberExpression
MemberExpression         ->  PrimaryExpression
PrimaryExpression        ->  IdentifierReference
```

The language spec encodes the twenty different precedence levels of JavaScript expressions using twenty levels of indirection between `IdentifierReference` and `Expression`. If we were to create a concrete syntax tree representing this statement according to the language spec, it would have twenty levels of nesting, and it would contain nodes with names like `BitwiseXORExpression`, which are unrelated to the actual code.

Standard Rule Names

Tree-sitter places no restrictions on how to name the rules of your grammar. It can be helpful, however, to follow certain conventions used by many other established grammars in the ecosystem. Some of these well-established patterns are listed below:

- `source_file` : Represents an entire source file, this rule is commonly used as the root node for a grammar,
- `expression` / `statement` : Used to represent statements and expressions for a given language. Commonly defined as a choice between several more specific sub-expression/sub-statement rules.
- `block` : Used as the parent node for block scopes, with its children representing the block's contents.
- `type` : Represents the types of a language such as `int` , `char` , and `void` .
- `identifier` : Used for constructs like variable names, function arguments, and object fields; this rule is commonly used as the `word` token in grammars.
- `string` : Used to represent `"string literals"` .
- `comment` : Used to represent comments, this rule is commonly used as an `extra` .

Using Precedence

To produce a readable syntax tree, we'd like to model JavaScript expressions using a much flatter structure like this:

```
{
  // ...

  expression: $ => choice(
    $.identifier,
    $.unary_expression,
    $.binary_expression,
    // ...
  ),

  unary_expression: $ => choice(
    seq('-', $.expression),
    seq('!', $.expression),
    // ...
  ),

  binary_expression: $ => choice(
    seq($.expression, '*', $.expression),
    seq($.expression, '+', $.expression),
    // ...
  ),
}
```

Of course, this flat structure is highly ambiguous. If we try to generate a parser, Tree-sitter gives us an error message:

```
Error: Unresolved conflict for symbol sequence:

  '-'  _expression  •  '*'  …

Possible interpretations:

  1:  '-'  (binary_expression  _expression  •  '*'  _expression)
  2:  (unary_expression  '-'  _expression)  •  '*'  …

Possible resolutions:

  1:  Specify a higher precedence in `binary_expression` than in the other rules.
  2:  Specify a higher precedence in `unary_expression` than in the other rules.
  3:  Specify a left or right associativity in `unary_expression`
  4:  Add a conflict for these rules: `binary_expression` `unary_expression`
```

> **Hint**
>
> The • character in the error message indicates where exactly during parsing the conflict occurs, or in other words, where the parser is encountering ambiguity.

For an expression like `-a * b` , it's not clear whether the `-` operator applies to the `a * b` or just to the `a` . This is where the `prec` function described in the previous page comes into play. By wrapping a rule with `prec` , we can indicate that certain sequence of symbols should *bind to each other more tightly* than others. For example, the `'-', $.expression` sequence in `unary_expression` should bind more tightly than the `$.expression, '+', $.expression` sequence in `binary_expression` :

```
{
  // ...

  unary_expression: $ =>
    prec(
      2,
      choice(
        seq("-", $.expression),
        seq("!", $.expression),
        // ...
      ),
    );
}
```

Using Associativity

Applying a higher precedence in `unary_expression` fixes that conflict, but there is still another conflict:

```
Error: Unresolved conflict for symbol sequence:

  _expression  '*'  _expression  •  '*'  …

Possible interpretations:

  1:  _expression  '*'  (binary_expression  _expression  •  '*'  _expression)
  2:  (binary_expression  _expression  '*'  _expression)  •  '*'  …

Possible resolutions:

  1:  Specify a left or right associativity in `binary_expression`
  2:  Add a conflict for these rules: `binary_expression`
```

For an expression like `a * b * c`, it's not clear whether we mean `a * (b * c)` or `(a * b) * c`. This is where `prec.left` and `prec.right` come into use. We want to select the second interpretation, so we use `prec.left`.

```
{
  // ...

  binary_expression: $ => choice(
    prec.left(2, seq($.expression, '*', $.expression)),
    prec.left(1, seq($.expression, '+', $.expression)),
    // ...
  ),
}
```

Using Conflicts

Sometimes, conflicts are actually desirable. In our JavaScript grammar, expressions and patterns can create intentional ambiguity. A construct like `[x, y]` could be legitimately parsed as both an array literal (like in `let a = [x, y]`) or as a destructuring pattern (like in `let [x, y] = arr`).

```
export default grammar({
  name: "javascript",

  rules: {
    expression: $ => choice(
      $.identifier,
      $.array,
      $.pattern,
    ),

    array: $ => seq(
      "[",
      optional(seq(
        $.expression, repeat(seq(",", $.expression))
      )),
      "]"
    ),

    array_pattern: $ => seq(
      "[",
      optional(seq(
        $.pattern, repeat(seq(",", $.pattern))
      )),
      "]"
    ),

    pattern: $ => choice(
      $.identifier,
      $.array_pattern,
    ),
  },
})
```

In such cases, we want the parser to explore both possibilities by explicitly declaring this ambiguity:

```
{
  name: "javascript",

  conflicts: $ => [
    [$.array, $.array_pattern],
  ],

  rules: {
    // ...
  },
}
```

> **Note**
>
> The example is a bit contrived for the purpose of illustrating the usage of conflicts. The actual JavaScript grammar isn't structured like that, but this conflict is actually present in the Tree-sitter JavaScript grammar.

Hiding Rules

You may have noticed in the above examples that some grammar rule name like `_expression` and `_type` began with an underscore. Starting a rule's name with an underscore causes the rule to be *hidden* in the syntax tree. This is useful for rules like `_expression` in the grammars above, which always just wrap a single child node. If these nodes were not hidden, they would add substantial depth and noise to the syntax tree without making it any easier to understand.

Using Fields

Often, it's easier to analyze a syntax node if you can refer to its children by *name* instead of by their position in an ordered list. Tree-sitter grammars support this using the `field` function. This function allows you to assign unique names to some or all of a node's children:

```
function_definition: $ =>
  seq(
    "func",
    field("name", $.identifier),
    field("parameters", $.parameter_list),
    field("return_type", $._type),
    field("body", $.block),
  );
```

Adding fields like this allows you to retrieve nodes using the field APIs.

Using Extras

Extras are tokens that can appear anywhere in the grammar, without being explicitly mentioned in a rule. This is useful for things like whitespace and comments, which can appear between any two tokens in most programming languages. To define an extra, you can use the `extras` function:

```
module.exports = grammar({
  name: "my_language",

  extras: ($) => [
    /\s/, // whitespace
    $.comment,
  ],

  rules: {
    comment: ($) =>
      token(
        choice(seq("//", /.*/), seq("/*", /[^*]*\*+([^/*][^*]*\*+)*/, "/")),
      ),
  },
});
```

> ⚠ **Warning**
>
> When adding more complicated tokens to `extras`, it's preferable to associate the pattern with a rule. This way, you avoid the lexer inlining this pattern in a bunch of spots, which can dramatically reduce the parser size.

For example, instead of defining the `comment` token inline in `extras`:

```
// ✗ Less preferable

const comment = token(
  choice(seq("//", /.*/), seq("/*", /[^*]*\*+([^/*][^*]*\*+)*/, "/")),
);

module.exports = grammar({
  name: "my_language",
  extras: ($) => [
    /\s/, // whitespace
    comment,
  ],
  rules: {
    // ...
  },
});
```

We can define it as a rule and then reference it in `extras`:

```
// ✅ More preferable

module.exports = grammar({
  name: "my_language",

  extras: ($) => [
    /\s/, // whitespace
    $.comment,
  ],

  rules: {
    // ...

    comment: ($) =>
      token(
        choice(seq("//", /.*/), seq("/*", /[^*]*\*+([^/*][^*]*\*+)*/, "/")),
      ),
  },
});
```

> **Note**
>
> Tree-sitter intentionally simplifies the whitespace character class, `\s`, to `[ \t\n\r]` as a performance optimization. This is because typically users do not require the full Unicode definition of whitespace.

Using Supertypes

Some rules in your grammar will represent abstract categories of syntax nodes, such as "expression", "type", or "declaration". These rules are often defined as simple choices between several other rules. For example, in the JavaScript grammar, the `_expression` rule is defined as a choice between many different kinds of expressions:

```
expression: $ => choice(
  $.identifier,
  $.unary_expression,
  $.binary_expression,
  $.call_expression,
  $.member_expression,
  // ...
),
```

By default, Tree-sitter will generate a visible node type for each of these abstract category rules, which can lead to unnecessarily deep and complex syntax trees. To avoid this, you can add these abstract category rules to the grammar's `supertypes` definition. Tree-sitter will then treat these rules as *supertypes*, and will not generate visible node types for them in the syntax tree.

```
module.exports = grammar({
  name: "javascript",

  supertypes: $ => [
    $.expression,
  ],

  rules: {
    expression: $ => choice(
      $.identifier,
      // ...
    ),

    // ...
  },
});
```

Although supertype rules are hidden from the syntax tree, they can still be used in queries. See the chapter on Query Syntax for more information.

Lexical Analysis

Tree-sitter's parsing process is divided into two phases: parsing (which is described above) and lexing — the process of grouping individual characters into the language's fundamental *tokens*. There are a few important things to know about how Tree-sitter's lexing works.

Conflicting Tokens

Grammars often contain multiple tokens that can match the same characters. For example, a grammar might contain the tokens (`"if"` and `/[a-z]+/`). Tree-sitter differentiates between these conflicting tokens in a few ways.

1. **Context-aware Lexing** — Tree-sitter performs lexing on-demand, during the parsing process. At any given position in a source document, the lexer only tries to recognize tokens that are *valid* at that position in the document.

2. **Lexical Precedence** — When the precedence functions described in the previous page are used *within* the `token` function, the given explicit precedence values serve as instructions to

the lexer. If there are two valid tokens that match the characters at a given position in the document, Tree-sitter will select the one with the higher precedence.

3. **Match Length** — If multiple valid tokens with the same precedence match the characters at a given position in a document, Tree-sitter will select the token that matches the longest sequence of characters.

4. **Match Specificity** — If there are two valid tokens with the same precedence, and they both match the same number of characters, Tree-sitter will prefer a token that is specified in the grammar as a `String` over a token specified as a `RegExp`.

5. **Rule Order** — If none of the above criteria can be used to select one token over another, Tree-sitter will prefer the token that appears earlier in the grammar.

If there is an external scanner it may have an additional impact over regular tokens defined in the grammar.

Lexical Precedence vs. Parse Precedence

One common mistake involves not distinguishing *lexical precedence* from *parse precedence*. Parse precedence determines which rule is chosen to interpret a given sequence of tokens. *Lexical precedence* determines which token is chosen to interpret at a given position of text, and it is a lower-level operation that is done first. The above list fully captures Tree-sitter's lexical precedence rules, and you will probably refer back to this section of the documentation more often than any other. Most of the time when you really get stuck, you're dealing with a lexical precedence problem. Pay particular attention to the difference in meaning between using `prec` inside the `token` function versus outside it. The *lexical precedence* syntax, as mentioned in the previous page, is `token(prec(N, ...))`.

Keywords

Many languages have a set of *keyword* tokens (e.g. `if`, `for`, `return`), as well as a more general token (e.g. `identifier`) that matches any word, including many of the keyword strings. For example, JavaScript has a keyword `instanceof`, which is used as a binary operator, like this:

```
if (a instanceof Something) b();
```

The following, however, is not valid JavaScript:

```
if (a instanceofSomething) b();
```

A keyword like `instanceof` cannot be followed immediately by another letter, because then it would be tokenized as an `identifier`, **even though an identifier is not valid at that position**. Because Tree-sitter uses context-aware lexing, as described above, it would not normally impose this restriction. By default, Tree-sitter would recognize `instanceofSomething` as two separate tokens: the `instanceof` keyword followed by an `identifier`.

Keyword Extraction

Fortunately, Tree-sitter has a feature that allows you to fix this, so that you can match the behavior of other standard parsers: the `word` token. If you specify a `word` token in your grammar, Tree-sitter will find the set of *keyword* tokens that match strings also matched by the `word` token. Then, during lexing, instead of matching each of these keywords individually, Tree-sitter will match the keywords via a two-step process where it *first* matches the `word` token.

For example, suppose we added `identifier` as the `word` token in our JavaScript grammar:

```
grammar({
  name: "javascript",

  word: $ => $.identifier,

  rules: {
    expression: $ =>
      choice(
        $.identifier,
        $.unary_expression,
        $.binary_expression,
        // ...
      ),

    binary_expression: $ =>
      choice(
        prec.left(1, seq($.expression, "instanceof", $.expression)),
        // ...
      ),

    unary_expression: $ =>
      choice(
        prec.left(2, seq("typeof", $.expression)),
        // ...
      ),

    identifier: $ => /[a-z_]+/,
  },
});
```

Tree-sitter would identify `typeof` and `instanceof` as keywords. Then, when parsing the invalid code above, rather than scanning for the `instanceof` token individually, it would scan for an `identifier` first, and find `instanceofSomething`. It would then correctly recognize the code as invalid.

Aside from improving error detection, keyword extraction also has performance benefits. It allows Tree-sitter to generate a smaller, simpler lexing function, which means that **the parser will compile much more quickly**.

> **Note**
>
> The word token must be a unique token that is not reused by another rule. If you want to have a word token used in a rule that's called something else, you should just alias the word token instead, like how the Rust grammar does it here

External Scanners

Many languages have some tokens whose structure is impossible or inconvenient to describe with a regular expression. Some examples:

- Indent and dedent tokens in Python
- Heredocs in Bash and Ruby
- Percent strings in Ruby

Tree-sitter allows you to handle these kinds of tokens using *external scanners*. An external scanner is a set of C functions that you, the grammar author, can write by hand to add custom logic for recognizing certain tokens.

To use an external scanner, there are a few steps. First, add an `externals` section to your grammar. This section should list the names of all of your external tokens. These names can then be used elsewhere in your grammar.

```
grammar({
  name: "my_language",

  externals: $ => [$.indent, $.dedent, $.newline],

  // ...
});
```

Then, add another C source file to your project. Its path must be src/scanner.c for the CLI to recognize it.

In this new source file, define an `enum` type containing the names of all of your external tokens. The ordering of this enum must match the order in your grammar's `externals` array; the actual names do not matter.

```
#include "tree_sitter/parser.h"
#include "tree_sitter/alloc.h"
#include "tree_sitter/array.h"

enum TokenType {
  INDENT,
  DEDENT,
  NEWLINE
}
```

Finally, you must define five functions with specific names, based on your language's name and five actions: *create*, *destroy*, *serialize*, *deserialize*, and *scan*.

Create

```
void * tree_sitter_my_language_external_scanner_create() {
  // ...
}
```

This function should create your scanner object. It will only be called once anytime your language is set on a parser. Often, you will want to allocate memory on the heap and return a pointer to it. If your external scanner doesn't need to maintain any state, it's ok to return `NULL`.

Destroy

```
void tree_sitter_my_language_external_scanner_destroy(void *payload) {
  // ...
}
```

This function should free any memory used by your scanner. It is called once when a parser is deleted or assigned a different language. It receives as an argument the same pointer that was returned from the *create* function. If your *create* function didn't allocate any memory, this function can be a no-op.

Serialize

```
unsigned tree_sitter_my_language_external_scanner_serialize(
  void *payload,
  char *buffer
) {
  // ...
}
```

This function should copy the complete state of your scanner into a given byte buffer, and return the number of bytes written. The function is called every time the external scanner successfully recognizes a token. It receives a pointer to your scanner and a pointer to a buffer. The maximum number of bytes that you can write is given by the `TREE_SITTER_SERIALIZATION_BUFFER_SIZE` constant, defined in the `tree_sitter/parser.h` header file.

The data that this function writes will ultimately be stored in the syntax tree so that the scanner can be restored to the right state when handling edits or ambiguities. For your parser to work correctly, the `serialize` function must store its entire state, and `deserialize` must restore the entire state. For good performance, you should design your scanner so that its state can be serialized as quickly and compactly as possible.

Deserialize

```
void tree_sitter_my_language_external_scanner_deserialize(
  void *payload,
  const char *buffer,
  unsigned length
) {
  // ...
}
```

This function should *restore* the state of your scanner based the bytes that were previously written by the `serialize` function. It is called with a pointer to your scanner, a pointer to the buffer of bytes, and the number of bytes that should be read. It is good practice to explicitly erase your scanner state variables at the start of this function, before restoring their values from the byte buffer.

Scan

Typically, one will

- Call `lexer->advance` several times, if the characters are valid for the token being lexed.
- Optionally, call `lexer->mark_end` to mark the end of the token, and "peek ahead" to check if the next character (or set of characters) invalidates the token.
- Set `lexer->result_symbol` to the token type.
- Return `true` from the scanning function, indicating that a token was successfully lexed.

Tree-sitter will then push resulting node to the parse stack, and the input position will remain where it reached at the point `lexer->mark_end` was called.

```
bool tree_sitter_my_language_external_scanner_scan(
  void *payload,
  TSLexer *lexer,
  const bool *valid_symbols
) {
  // ...
}
```

The second parameter to this function is the lexer, of type `TSLexer`. The `TSLexer` struct has the following fields:

- **`int32_t lookahead`** — The current next character in the input stream, represented as a 32-bit unicode code point.
- **`TSSymbol result_symbol`** — The symbol that was recognized. Your scan function should *assign* to this field one of the values from the `TokenType` enum, described above.
- **`void (*advance)(TSLexer *, bool skip)`** — A function for advancing to the next character. If you pass `true` for the second argument, the current character will be treated as whitespace; whitespace won't be included in the text range associated with tokens emitted by the external scanner.
- **`void (*mark_end)(TSLexer *)`** — A function for marking the end of the recognized token. This allows matching tokens that require multiple characters of lookahead. By default, (if you don't call `mark_end`), any character that you moved past using the `advance` function will be included in the size of the token. But once you call `mark_end`, then any later calls to `advance` will *not* increase the size of the returned token. You can call `mark_end` multiple times to increase the size of the token.
- **`uint32_t (*get_column)(TSLexer *)`** — A function for querying the current column position of the lexer. It returns the number of codepoints since the start of the current line. The codepoint position is recalculated on every call to this function by reading from the start of the line.
- **`bool (*is_at_included_range_start)(const TSLexer *)`** — A function for checking whether the parser has just skipped some characters in the document. When parsing an embedded document using the `ts_parser_set_included_ranges` function (described in the multi-language document section), the scanner may want to apply some special behavior when moving to a disjoint part of the document. For example, in EJS documents, the JavaScript parser uses this function to enable inserting automatic semicolon tokens in between the code directives, delimited by `<%` and `%>`.
- **`bool (*eof)(const TSLexer *)`** — A function for determining whether the lexer is at the end of the file. The value of `lookahead` will be `0` at the end of a file, but this function should be used instead of checking for that value because the `0` or "NUL" value is also a valid character that could be present in the file being parsed.

The third argument to the `scan` function is an array of booleans that indicates which of external tokens are expected by the parser. You should only look for a given token if it is valid according to this array. At the same time, you cannot backtrack, so you may need to combine certain pieces of logic.

```
if (valid_symbols[INDENT] || valid_symbols[DEDENT]) {

  // ... logic that is common to both `INDENT` and `DEDENT`

  if (valid_symbols[INDENT]) {

    // ... logic that is specific to `INDENT`

    lexer->result_symbol = INDENT;
    return true;
  }
}
```

External Scanner Helpers

Allocator

Instead of using libc's `malloc`, `calloc`, `realloc`, and `free`, you should use the versions prefixed with `ts_` from `tree_sitter/alloc.h`. These macros can allow a potential consumer to override the default allocator with their own implementation, but by default will use the libc functions.

As a consumer of the tree-sitter core library as well as any parser libraries that might use allocations, you can enable overriding the default allocator and have it use the same one as the library allocator, of which you can set with `ts_set_allocator`. To enable this overriding in scanners, you must compile them with the `TREE_SITTER_REUSE_ALLOCATOR` macro defined, and tree-sitter the library must be linked into your final app dynamically, since it needs to resolve the internal functions at runtime. If you are compiling an executable binary that uses the core library, but want to load parsers dynamically at runtime, then you will have to use a special linker flag on Unix. For non-Darwin systems, that would be `--dynamic-list` and for Darwin systems, that would be `-exported_symbols_list`. The CLI does exactly this, so you can use it as a reference (check out `cli/build.rs`).

For example, assuming you wanted to allocate 100 bytes for your scanner, you'd do so like the following example:

```
#include "tree_sitter/parser.h"
#include "tree_sitter/alloc.h"

// ...

void* tree_sitter_my_language_external_scanner_create() {
  return ts_calloc(100, 1); // or ts_malloc(100)
}

// ...
```

Arrays

If you need to use array-like types in your scanner, such as tracking a stack of indentations or tags, you should use the array macros from `tree_sitter/array.h`.

There are quite a few of them provided for you, but here's how you could get started tracking some . Check out the header itself for more detailed documentation.

> ⚠ **Attention**
>
> Do not use any of the array functions or macros that are prefixed with an underscore and have comments saying that it is not what you are looking for. These are internal functions used as helpers by other macros that are public. They are not meant to be used directly, nor are they what you want.

```c
#include "tree_sitter/parser.h"
#include "tree_sitter/array.h"

enum TokenType {
  INDENT,
  DEDENT,
  NEWLINE,
  STRING,
}

// Create the array in your create function

void* tree_sitter_my_language_external_scanner_create() {
  return ts_calloc(1, sizeof(Array(int)));

  // or if you want to zero out the memory yourself

  Array(int) *stack = ts_malloc(sizeof(Array(int)));
  array_init(&stack);
  return stack;
}

bool tree_sitter_my_language_external_scanner_scan(
  void *payload,
  TSLexer *lexer,
  const bool *valid_symbols
) {
  Array(int) *stack = payload;
  if (valid_symbols[INDENT]) {
    array_push(stack, lexer->get_column(lexer));
    lexer->result_symbol = INDENT;
    return true;
  }
  if (valid_symbols[DEDENT]) {
    array_pop(stack); // this returns the popped element by value, but we don't need
it
    lexer->result_symbol = DEDENT;
    return true;
  }

  // we can also use an array on the stack to keep track of a string

  Array(char) next_string = array_new();

  if (valid_symbols[STRING] && lexer->lookahead == '"') {
    lexer->advance(lexer, false);
    while (lexer->lookahead != '"' && lexer->lookahead != '\n' && !lexer->eof(lexer))
{
      array_push(&next_string, lexer->lookahead);
      lexer->advance(lexer, false);
    }

    // assume we have some arbitrary constraint of not having more than 100 characters
in a string
    if (lexer->lookahead == '"' && next_string.size <= 100) {
      lexer->advance(lexer, false);
      lexer->result_symbol = STRING;
      return true;
    }
  }

  return false;
}
```

Other External Scanner Details

External scanners have priority over Tree-sitter's normal lexing process. When a token listed in the externals array is valid at a given position, the external scanner is called first. This makes external scanners a powerful way to override Tree-sitter's default lexing behavior, especially for cases that can't be handled with regular lexical rules, parsing, or dynamic precedence.

During error recovery, Tree-sitter's first step is to call the external scanner's scan function with all tokens marked as valid. Your scanner should detect and handle this case appropriately. One simple approach is to add an unused "sentinel" token at the end of your externals array:

```javascript
{
  name: "my_language",

  externals: $ => [$.token1, $.token2, $.error_sentinel]

  // ...
}
```

You can then check if this sentinel token is marked valid to determine if Tree-sitter is in error recovery mode.

If you would rather not handle the error recovery case explicitly, the easiest way to "opt-out" and let tree-sitter's internal lexer handle it is to return `false` from your scan function when `valid_symbols` contains the error sentinel.

```c
bool tree_sitter_my_language_external_scanner_scan(
  void *payload,
  TSLexer *lexer,
  const bool *valid_symbols
) {
  if (valid_symbols[ERROR_SENTINEL]) {
    return false;
  }
  // ...
}
```

When you include literal keywords in the externals array, for example:

```javascript
externals: $ => ['if', 'then', 'else']
```

those keywords will be tokenized by the external scanner whenever they appear in the grammar.

This is equivalent to declaring named tokens and aliasing them:

```javascript
{
  name: "my_language",

  externals: $ => [$.if_keyword, $.then_keyword, $.else_keyword],

  rules: {

    // then using it in a rule like so:
    if_statement: $ => seq(alias($.if_keyword, 'if'), ...),

    // ...
  }
}
```

The tokenization process for external keywords works in two stages:

1. The external scanner attempts to recognize the token first
2. If the scanner returns true and sets a token, that token is used
3. If the scanner returns false, Tree-sitter falls back to its internal lexer

However, when you use rule references (like `$.if_keyword`) in the externals array without defining the corresponding rules in the grammar, Tree-sitter cannot fall back to its internal lexer. In this case, the external scanner is solely responsible for recognizing these tokens.

> **Danger**
>
> - External scanners can easily create infinite loops
> - Be extremely careful when emitting zero-width tokens
> - Always use the `eof` function when looping through characters

Writing Tests

For each rule that you add to the grammar, you should first create a *test* that describes how the syntax trees should look when parsing that rule. These tests are written using specially-formatted text files in the `test/corpus/` directory within your parser's root folder.

For example, you might have a file called `test/corpus/statements.txt` that contains a series of entries like this:

```
==================
Return statements
==================

func x() int {
  return 1;
}

---

(source_file
  (function_definition
    (identifier)
    (parameter_list)
    (primitive_type)
    (block
      (return_statement (number)))))
```

- The **name** of each test is written between two lines containing only `=` (equal sign) characters.
- Then the **input source code** is written, followed by a line containing three or more `-` (dash) characters.
- Then, the **expected output syntax tree** is written as an S-expression. The exact placement of whitespace in the S-expression doesn't matter, but ideally the syntax tree should be legible.

> **Tip**
>
> The S-expression does not show syntax nodes like `func`, `(` and `;`, which are expressed as strings and regexes in the grammar. It only shows the *named* nodes, as described in this section of the page on parser usage.

The expected output section can also *optionally* show the *field names* associated with each child node. To include field names in your tests, you write a node's field name followed by a colon, before the node itself in the S-expression:

```
(source_file
  (function_definition
    name: (identifier)
    parameters: (parameter_list)
    result: (primitive_type)
    body: (block
      (return_statement (number)))))
```

- If your language's syntax conflicts with the `===` and `---` test separators, you can optionally add an arbitrary identical suffix (in the below example, `|||`) to disambiguate them:

```
==================|||
Basic module
==================|||

---- MODULE Test ----
increment(n) == n + 1
====

---|||

(source_file
  (module (identifier)
    (operator (identifier)
      (parameter_list (identifier))
      (plus (identifier_ref) (number)))))
```

These tests are important. They serve as the parser's API documentation, and they can be run every time you change the grammar to verify that everything still parses correctly.

By default, the `tree-sitter test` command runs all the tests in your `test/corpus/` folder. To run a particular test, you can use the `-i` flag:

```
tree-sitter test -i 'Return statements'
```

The recommendation is to be comprehensive in adding tests. If it's a visible node, add it to a test file in your `test/corpus` directory. It's typically a good idea to test all the permutations of each language construct. This increases test coverage, but doubly acquaints readers with a way to examine expected outputs and understand the "edges" of a language.

> **Tip**
>
> After modifying the grammar, you can run `tree-sitter test -u` to update all syntax trees in corpus files with current parser output.

Attributes

Tests can be annotated with a few `attributes`. Attributes must be put in the header, below the test name, and start with a `:`. A couple of attributes also take in a parameter, which require the use of parenthesis.

> **Tip**
>
> If you'd like to supply in multiple parameters, e.g. to run tests on multiple platforms or to test multiple languages, you can repeat the attribute on a new line.

The following attributes are available:

- `:cst` - This attribute specifies that the expected output should be in the form of a CST instead of the normal S-expression. This CST matches the format given by `parse --cst`.
- `:error` — This attribute will assert that the parse tree contains an error. It's useful to just validate that a certain input is invalid without displaying the whole parse tree, as such you should omit the parse tree below the `---` line.
- `:fail-fast` — This attribute will stop the testing of additional cases if the test marked with this attribute fails.
- `:language(LANG)` — This attribute will run the tests using the parser for the specified language. This is useful for multi-parser repos, such as XML and DTD, or Typescript and TSX. The default parser used will always be the first entry in the `grammars` field in the `tree-sitter.json` config file, so having a way to pick a second or even third parser is useful.
- `:platform(PLATFORM)` — This attribute specifies the platform on which the test should run. It is useful to test platform-specific behavior (e.g. Windows newlines are different from Unix). This attribute must match up with Rust's `std::env::consts::OS`.
- `:skip` — This attribute will skip the test when running `tree-sitter test`. This is useful when you want to temporarily disable running a test without deleting it.

Examples using attributes:

```
========================
Test that will be skipped
:skip
========================

int main() {}

------------------------

==================================
Test that will run on Linux or macOS

:platform(linux)
:platform(macos)
==================================

int main() {}

----------------------------------

================================================================================
Test that expects an error, and will fail fast if there's no parse error
:fail-fast
:error
================================================================================

int main ( {}

--------------------------------------------------------------------------------

================================================
Test that will parse with both Typescript and TSX
:language(typescript)
:language(tsx)
================================================

console.log('Hello, world!');

------------------------------------------------
```

Automatic Compilation

You might notice that the first time you run `tree-sitter test` after regenerating your parser, it takes some extra time. This is because Tree-sitter automatically compiles your C code into a dynamically-loadable library. It recompiles your parser as-needed whenever you update it by re-running `tree-sitter generate`, or whenever the external scanner file is changed.

Publishing your grammar

Once you feel that your parser is in a stable working state for consumers to use, you can publish it to various registries. It's strongly recommended to publish grammars to GitHub, crates.io (Rust), npm (JavaScript), and PyPI (Python) to make it easier for others to find and use your grammar.

If your grammar is hosted on GitHub, you can make use of our reusable workflows to handle the publishing process for you. This action will automatically handle regenerating and publishing your grammar in CI, so long as you have the required tokens setup for the various registries. For an example of this workflow in action, see the Python grammar's GitHub

From start to finish

To release a new grammar (or publish your first version), these are the steps you should follow:

1. Bump your version to the desired version with `tree-sitter version`. For example, if you're releasing version `1.0.0` of your grammar, you'd run `tree-sitter version 1.0.0`.
2. Commit the changes with `git commit -am "Release 1.0.0" (or however you like)` (ensure that your working directory is clean).
3. Tag the commit with `git tag -- v1.0.0`.
4. Push the commit and tag with `git push --tags origin main` (assuming you're on the `main` branch, and `origin` is your remote).
5. (optional) If you've set up the GitHub workflows for your grammar, the release will be automatically published to GitHub, crates.io, npm, and PyPI.

Adhering to Semantic Versioning

When releasing new versions of your grammar, it's important to adhere to Semantic Versioning. This ensures that consumers can predictably update their dependencies and that their existing tree-sitter integrations (queries, tree traversal code, node type checks) will continue to work as expected when upgrading.

1. Increment the major version when you make incompatible changes to the grammar's node types or structure
2. Increment the minor version when you add new node types or patterns while maintaining backward compatibility
3. Increment the patch version when you fix bugs without changing the grammar's structure

For grammars in version 0.y.z (zero version), the usual semantic versioning rules are technically relaxed. However, if your grammar already has users, it's recommended to treat version changes more conservatively:

- Treat patch version (`z`) changes as if they were minor version changes
- Treat minor version (`y`) changes as if they were major version changes

This helps maintain stability for existing users during the pre-1.0 phase. By following these versioning guidelines, you ensure that downstream users can safely upgrade without their existing queries breaking.

Syntax Highlighting

Syntax highlighting is a very common feature in applications that deal with code. Tree-sitter has built-in support for syntax highlighting via the `tree-sitter-highlight` library, which is now used on GitHub.com for highlighting code written in several languages. You can also perform syntax highlighting at the command line using the `tree-sitter highlight` command.

This document explains how the Tree-sitter syntax highlighting system works, using the command line interface. If you are using `tree-sitter-highlight` library (either from C or from Rust), all of these concepts are still applicable, but the configuration data is provided using in-memory objects, rather than files.

Overview

All the files needed to highlight a given language are normally included in the same git repository as the Tree-sitter grammar for that language (for example, `tree-sitter-javascript`, `tree-sitter-ruby`). To run syntax highlighting from the command-line, three types of files are needed:

1. Per-user configuration in `~/.config/tree-sitter/config.json` (see the init-config page for more info).
2. Language configuration in grammar repositories' `tree-sitter.json` files (see the init page for more info).
3. Tree queries in the grammars repositories' `queries` folders.

For an example of the language-specific files, see the `tree-sitter.json` file and `queries` directory in the `tree-sitter-ruby` repository. The following sections describe the behavior of each file.

Language Configuration

The `tree-sitter.json` file is used by the Tree-sitter CLI. Within this file, the CLI looks for data nested under the top-level `"grammars"` key. This key is expected to contain an array of objects with the following keys:

Basics

These keys specify basic information about the parser:

- `scope` (required) — A string like `"source.js"` that identifies the language. We strive to match the scope names used by popular TextMate grammars and by the Linguist library.
- `path` (optional) — A relative path from the directory containing `tree-sitter.json` to another directory containing the `src/` folder, which contains the actual generated parser. The default value is `"."` (so that `src/` is in the same folder as `tree-sitter.json`), and this very rarely needs to be overridden.
- `external-files` (optional) — A list of relative paths from the root dir of a parser to files that should be checked for modifications during recompilation. This is useful during development to have changes to other files besides scanner.c be picked up by the cli.

Language Detection

These keys help to decide whether the language applies to a given file:

- `file-types` — An array of filename suffix strings. The grammar will be used for files whose names end with one of these suffixes. Note that the suffix may match an *entire* filename.
- `first-line-regex` — A regex pattern that will be tested against the first line of a file to determine whether this language applies to the file. If present, this regex will be used for any file whose language does not match any grammar's `file-types`.
- `content-regex` — A regex pattern that will be tested against the contents of the file to break ties in cases where multiple grammars matched the file using the above two criteria. If the regex matches, this grammar will be preferred over another grammar with no `content-regex`. If the regex does not match, a grammar with no `content-regex` will be preferred over this one.
- `injection-regex` — A regex pattern that will be tested against a *language name* ito determine whether this language should be used for a potential *language injection* site. Language injection is described in more detail in a later section.

Query Paths

These keys specify relative paths from the directory containing `tree-sitter.json` to the files that control syntax highlighting:

- `highlights` — Path to a *highlight query*. Default: `queries/highlights.scm`
- `locals` — Path to a *local variable query*. Default: `queries/locals.scm`.
- `injections` — Path to an *injection query*. Default: `queries/injections.scm`.

The behaviors of these three files are described in the next section.

Queries

Tree-sitter's syntax highlighting system is based on *tree queries*, which are a general system for pattern-matching on Tree-sitter's syntax trees. See this section of the documentation for more information about tree queries.

Syntax highlighting is controlled by *three* different types of query files that are usually included in the `queries` folder. The default names for the query files use the `.scm` file. We chose this extension because it commonly used for files written in Scheme, a popular dialect of Lisp, and these query files use a Lisp-like syntax.

Highlights

The most important query is called the highlights query. The highlights query uses *captures* to assign arbitrary *highlight names* to different nodes in the tree. Each highlight name can then be mapped to a color (as described in the init-config command). Commonly used highlight names include `keyword`, `function`, `type`, `property`, and `string`. Names can also be dot-separated like `function.builtin`.

Example Go Snippet

For example, consider the following Go code:

```go
func increment(a int) int {
    return a + 1
}
```

With this syntax tree:

```
(source_file
  (function_declaration
    name: (identifier)
    parameters: (parameter_list
      (parameter_declaration
        name: (identifier)
        type: (type_identifier)))
    result: (type_identifier)
    body: (block
      (return_statement
        (expression_list
          (binary_expression
            left: (identifier)
            right: (int_literal)))))))
```

Example Query

Suppose we wanted to render this code with the following colors:

- keywords `func` and `return` in purple
- function `increment` in blue
- type `int` in green
- number `5` brown

We can assign each of these categories a *highlight name* using a query like this:

```
; highlights.scm

"func" @keyword
"return" @keyword
(type_identifier) @type
(int_literal) @number
(function_declaration name: (identifier) @function)
```

Then, in our config file, we could map each of these highlight names to a color:

```
{
  "theme": {
    "keyword": "purple",
    "function": "blue",
    "type": "green",
    "number": "brown"
  }
}
```

Highlights Result

Running `tree-sitter highlight` on this Go file would produce output like this:

Output ›

Local Variables

Good syntax highlighting helps the reader to quickly distinguish between the different types of *entities* in their code. Ideally, if a given entity appears in *multiple* places, it should be colored the

same in each place. The Tree-sitter syntax highlighting system can help you to achieve this by keeping track of local scopes and variables.

The *local variables* query is different from the highlights query in that, while the highlights query uses *arbitrary* capture names, which can then be mapped to colors, the locals variable query uses a fixed set of capture names, each of which has a special meaning.

The capture names are as follows:

- `@local.scope` — indicates that a syntax node introduces a new local scope.
- `@local.definition` — indicates that a syntax node contains the *name* of a definition within the current local scope.
- `@local.reference` — indicates that a syntax node contains the *name*, which *may* refer to an earlier definition within some enclosing scope.

Additionally, to ignore certain nodes from being tagged, you can use the `@ignore` capture. This is useful if you want to exclude a subset of nodes from being tagged. When writing a query leveraging this, you should ensure this pattern comes before any other patterns that would be used for tagging, for example:

```
(expression (identifier) @ignore)

(identifier) @local.reference
```

When highlighting a file, Tree-sitter will keep track of the set of scopes that contains any given position, and the set of definitions within each scope. When processing a syntax node that is captured as a `local.reference`, Tree-sitter will try to find a definition for a name that matches the node's text. If it finds a match, Tree-sitter will ensure that the *reference*, and the *definition* are colored the same.

The information produced by this query can also be *used* by the highlights query. You can *disable* a pattern for nodes, which have been identified as local variables by adding the predicate `(#is-not? local)` to the pattern. This is used in the example below:

Example Ruby Snippet

Consider this Ruby code:

```ruby
def process_list(list)
  context = current_context
  list.map do |item|
    process_item(item, context)
  end
end

item = 5
list = [item]
```

With this syntax tree:

```
(program
  (method
    name: (identifier)
    parameters: (method_parameters
      (identifier))
    (assignment
      left: (identifier)
      right: (identifier))
    (method_call
      method: (call
        receiver: (identifier)
        method: (identifier))
      block: (do_block
        (block_parameters
          (identifier))
        (method_call
          method: (identifier)
          arguments: (argument_list
            (identifier)
            (identifier))))))
  (assignment
    left: (identifier)
    right: (integer))
  (assignment
    left: (identifier)
    right: (array
      (identifier))))
```

There are several types of names within this method:

- `process_list` is a method.
- Within this method, `list` is a formal parameter
- `context` is a local variable.
- `current_context` is *not* a local variable, so it must be a method.
- Within the `do` block, `item` is a formal parameter
- Later on, `item` and `list` are both local variables (not formal parameters).

Example Queries

Let's write some queries that let us clearly distinguish between these types of names. First, set up the highlighting query, as described in the previous section. We'll assign distinct colors to method calls, method definitions, and formal parameters:

```
; highlights.scm

(call method: (identifier) @function.method)
(method_call method: (identifier) @function.method)

(method name: (identifier) @function.method)

(method_parameters (identifier) @variable.parameter)
(block_parameters (identifier) @variable.parameter)

((identifier) @function.method
 (#is-not? local))
```

Then, we'll set up a local variable query to keep track of the variables and scopes. Here, we're indicating that methods and blocks create local *scopes*, parameters and assignments create *definitions*, and other identifiers should be considered *references*:

```
; locals.scm

(method) @local.scope
(do_block) @local.scope

(method_parameters (identifier) @local.definition)
(block_parameters (identifier) @local.definition)

(assignment left:(identifier) @local.definition)

(identifier) @local.reference
```

Locals Result

Running `tree-sitter highlight` on this ruby file would produce output like this:

Output ›

Language Injection

Some source files contain code written in multiple different languages. Examples include:

- HTML files, which can contain JavaScript inside `<script>` tags and CSS inside `<style>` tags
- ERB files, which contain Ruby inside `<% %>` tags, and HTML outside those tags
- PHP files, which can contain HTML between the `<php` tags
- JavaScript files, which contain regular expression syntax within regex literals
- Ruby, which can contain snippets of code inside heredoc literals, where the heredoc delimiter often indicates the language

All of these examples can be modeled in terms a *parent* syntax tree and one or more *injected* syntax trees, which reside *inside* of certain nodes in the parent tree. The language injection query allows you to specify these "injections" using the following captures:

- `@injection.content` — indicates that the captured node should have its contents re-parsed using another language.
- `@injection.language` — indicates that the captured node's text may contain the *name* of a language that should be used to re-parse the `@injection.content`.

The language injection behavior can also be configured by some properties associated with patterns:

- `injection.language` — can be used to hard-code the name of a specific language.
- `injection.combined` — indicates that *all* the matching nodes in the tree should have their content parsed as *one* nested document.
- `injection.include-children` — indicates that the `@injection.content` node's *entire* text should be re-parsed, including the text of its child nodes. By default, child nodes' text will be *excluded* from the injected document.
- `injection.self` — indicates that the `@injection.content` node should be parsed using the same language as the node itself. This is useful for cases where the node's language is not known until runtime (e.g. via inheriting another language)
- `injection.parent` indicates that the `@injection.content` node should be parsed using the same language as the node's parent language. This is only meant for injections that need to refer back to the parent language to parse the node's text inside the injected language.

Examples

Consider this ruby code:

```
system <<-BASH.strip!
  abc --def | ghi > jkl
BASH
```

With this syntax tree:

```
(program
  (method_call
    method: (identifier)
    arguments: (argument_list
      (call
        receiver: (heredoc_beginning)
        method: (identifier))))
  (heredoc_body
    (heredoc_end)))
```

The following query would specify that the contents of the heredoc should be parsed using a language named "BASH" (because that is the text of the `heredoc_end` node):

```
(heredoc_body
  (heredoc_end) @injection.language) @injection.content
```

You can also force the language using the `#set!` predicate. For example, this will force the language to be always `ruby`.

```
((heredoc_body) @injection.content
 (#set! injection.language "ruby"))
```

Unit Testing

Tree-sitter has a built-in way to verify the results of syntax highlighting. The interface is based on Sublime Text's system for testing highlighting.

Tests are written as normal source code files that contain specially-formatted *comments* that make assertions about the surrounding syntax highlighting. These files are stored in the `test/highlight` directory in a grammar repository.

Here is an example of a syntax highlighting test for JavaScript:

```
var abc = function(d) {
  // <- keyword
  //          ^ keyword
  //              ^ variable.parameter
  // ^ function

  if (a) {
  // <- keyword
  // ^ punctuation.bracket

    foo(`foo ${bar}`);
    // <- function
    //    ^ string
    //          ^ variable
  }

  baz();
  // <- !variable
};
```

> **From the Sublime text docs**
>
> The two types of tests are:
>
> **Caret**: ^ this will test the following selector against the scope on the most recent non-test line. It will test it at the same column the ^ is in. Consecutive ^s will test each column against the selector.
>
> **Arrow**: <- this will test the following selector against the scope on the most recent non-test line. It will test it at the same column as the comment character is in.

> **Note**
>
> An exclamation mark (`!`) can be used to negate a selector. For example, `!keyword` will match any scope that is not the `keyword` class.

Code Navigation Systems

Tree-sitter can be used in conjunction with its query language as a part of code navigation systems. An example of such a system can be seen in the `tree-sitter tags` command, which emits a textual dump of the interesting syntactic nodes in its file argument. A notable application of this is GitHub's support for search-based code navigation. This document exists to describe how to integrate with such systems, and how to extend this functionality to any language with a Tree-sitter grammar.

Tagging and captures

Tagging is the act of identifying the entities that can be named in a program. We use Tree-sitter queries to find those entities. Having found them, you use a syntax capture to label the entity and its name.

The essence of a given tag lies in two pieces of data: the *role* of the entity that is matched (i.e. whether it is a definition or a reference) and the *kind* of that entity, which describes how the entity is used (i.e. whether it's a class definition, function call, variable reference, and so on). Our convention is to use a syntax capture following the `@role.kind` capture name format, and another inner capture, always called `@name`, that pulls out the name of a given identifier.

You may optionally include a capture named `@doc` to bind a docstring. For convenience purposes, the tagging system provides two built-in functions, `#select-adjacent!` and `#strip!` that are convenient for removing comment syntax from a docstring. `#strip!` takes a capture as its first argument and a regular expression as its second, expressed as a quoted string. Any text patterns matched by the regular expression will be removed from the text associated with the passed capture. `#select-adjacent!`, when passed two capture names, filters the text associated with the first capture so that only nodes adjacent to the second capture are preserved. This can be useful when writing queries that would otherwise include too much information in matched comments.

Examples

This query recognizes Python function definitions and captures their declared name. The `function_definition` syntax node is defined in the Python Tree-sitter grammar.

```
(function_definition
  name: (identifier) @name) @definition.function
```

A more sophisticated query can be found in the JavaScript Tree-sitter repository:

```
(assignment_expression
  left: [
    (identifier) @name
    (member_expression
      property: (property_identifier) @name)
  ]
  right: [(arrow_function) (function)]
) @definition.function
```

An even more sophisticated query is in the Ruby Tree-sitter repository, which uses built-in functions to strip the Ruby comment character (`#`) from the docstrings associated with a class or singleton-class declaration, then selects only the docstrings adjacent to the node matched as `@definition.class`.

```
(
  (comment)* @doc
  .
  [
    (class
      name: [
        (constant) @name
        (scope_resolution
          name: (_) @name)
      ]) @definition.class
    (singleton_class
      value: [
        (constant) @name
        (scope_resolution
          name: (_) @name)
      ]) @definition.class
  ]
  (#strip! @doc "^#\\s*")
  (#select-adjacent! @doc @definition.class)
)
```

The below table describes a standard vocabulary for kinds and roles during the tagging process. New applications may extend (or only recognize a subset of) these capture names, but it is desirable to standardize on the names below.

Category	Tag
Class definitions	`@definition.class`
Function definitions	`@definition.function`
Interface definitions	`@definition.interface`
Method definitions	`@definition.method`
Module definitions	`@definition.module`

Category	Tag
Function/method calls	`@reference.call`
Class reference	`@reference.class`
Interface implementation	`@reference.implementation`

Command-line invocation

You can use the `tree-sitter tags` command to test out a tags query file, passing as arguments one or more files to tag. We can run this tool from within the Tree-sitter Ruby repository, over code in a file called `test.rb`:

```ruby
module Foo
  class Bar
    # won't be included

    # is adjacent, will be
    def baz
    end
  end
end
```

Invoking `tree-sitter tags test.rb` produces the following console output, representing matched entities' name, role, location, first line, and docstring:

```
    test.rb
        Foo              | module       def (0, 7) - (0, 10) `module Foo`
        Bar              | class        def (1, 8) - (1, 11) `class Bar`
        baz              | method       def (2, 8) - (2, 11) `def baz`  "is adjacent,
will be"
```

It is expected that tag queries for a given language are located at `queries/tags.scm` in that language's repository.

Unit Testing

Tags queries may be tested with `tree-sitter test`. Files under `test/tags/` are checked using the same comment system as highlights queries. For example, the above Ruby tags can be tested with these comments:

```ruby
module Foo
  #     ^ definition.module
  class Bar
    #    ^ definition.class

    def baz
      #  ^ definition.method
    end
  end
end
```

Implementation

Tree-sitter consists of two components: a C library (`libtree-sitter`), and a command-line tool (the `tree-sitter` CLI).

The library, `libtree-sitter` , is used in combination with the parsers generated by the CLI, to produce syntax trees from source code and keep the syntax trees up-to-date as the source code changes. `libtree-sitter` is designed to be embedded in applications. It is written in plain C. Its interface is specified in the header file `tree_sitter/api.h` .

The CLI is used to generate a parser for a language by supplying a context-free grammar describing the language. The CLI is a build tool; it is no longer needed once a parser has been generated. It is written in Rust, and is available on crates.io, npm, and as a pre-built binary on GitHub.

The CLI

The `tree-sitter` CLI's most important feature is the `generate` command. This subcommand reads in a context-free grammar from a file called `grammar.js` and outputs a parser as a C file called `parser.c` . The source files in the `cli/src` directory all play a role in producing the code in `parser.c` . This section will describe some key parts of this process.

Parsing a Grammar

First, Tree-sitter must evaluate the JavaScript code in `grammar.js` and convert the grammar to a JSON format. It does this by shelling out to `node` . The format of the grammars is formally specified by the JSON schema in grammar.schema.json. The parsing is implemented in parse_grammar.rs.

Grammar Rules

A Tree-sitter grammar is composed of a set of *rules* — objects that describe how syntax nodes can be composed of other syntax nodes. There are several types of rules: symbols, strings, regexes, sequences, choices, repetitions, and a few others. Internally, these are all represented using an enum called `Rule` .

Preparing a Grammar

Once a grammar has been parsed, it must be transformed in several ways before it can be used to generate a parser. Each transformation is implemented by a separate file in the `prepare_grammar` directory, and the transformations are ultimately composed together in `prepare_grammar/mod.rs` .

At the end of these transformations, the initial grammar is split into two grammars: a *syntax grammar* and a *lexical grammar*. The syntax grammar describes how the language's *non-terminal symbols* are constructed from other grammar symbols, and the lexical grammar describes how the grammar's *terminal symbols* (strings and regexes) can be composed of individual characters.

Building Parse Tables

The Runtime

WIP

Contributing

Code of Conduct

Contributors to Tree-sitter should abide by the Contributor Covenant.

Developing Tree-sitter

Prerequisites

To make changes to Tree-sitter, you should have:

1. A C compiler, for compiling the core library and the generated parsers.
2. A Rust toolchain, for compiling the Rust bindings, the highlighting library, and the CLI.
3. Node.js and NPM, for generating parsers from `grammar.js` files.
4. Either Emscripten, Docker, or podman for compiling the library to Wasm.

Building

Clone the repository:

```
git clone https://github.com/tree-sitter/tree-sitter
cd tree-sitter
```

Optionally, build the Wasm library. If you skip this step, then the `tree-sitter playground` command will require an internet connection. If you have Emscripten installed, this will use your `emcc` compiler. Otherwise, it will use Docker or Podman:

```
cd lib/binding_web
npm install # or your JS package manager of choice
npm run build
```

Build the Rust libraries and the CLI:

```
cargo build --release
```

This will create the `tree-sitter` CLI executable in the `target/release` folder.

If you want to automatically install the `tree-sitter` CLI in your system, you can run:

```
cargo install --path crates/cli
```

If you're going to be in a fast iteration cycle and would like the CLI to build faster, you can use the `release-dev` profile:

```
cargo build --profile release-dev
# or
cargo install --path crates/cli --profile release-dev
```

Testing

Before you can run the tests, you need to fetch some upstream grammars that are used for testing:

```
cargo xtask fetch-fixtures
```

To test any changes you've made to the CLI, you can regenerate these parsers using your current CLI code:

```
cargo xtask generate-fixtures
```

Then you can run the tests:

```
cargo xtask test
```

Similarly, to test the Wasm binding, you need to compile these parsers to Wasm:

```
cargo xtask generate-fixtures --wasm
cargo xtask test-wasm
```

Wasm Stdlib

The tree-sitter Wasm stdlib can be built via xtask:

```
cargo xtask build-wasm-stdlib
```

This command looks for the Wasi SDK indicated by the `TREE_SITTER_WASI_SDK_PATH` environment variable. If you don't have the binary, it can be downloaded from wasi-sdk's releases page. Similarly, this command also looks for the `wasm-opt` tool from binaryen indicated by the `TREE_SITTER_BINARYEN_PATH` environment variable. `wasm-opt` and the rest of the binaryen tool suite can be downloaded from the project's releases page. Note that any changes to `crates/language/wasm/**` requires rebuilding the tree-sitter Wasm stdlib via `cargo xtask build-wasm-stdlib`.

Debugging

The test script has a number of useful flags. You can list them all by running `cargo xtask test -h`. Here are some of the main flags:

If you want to run a specific unit test, pass its name (or part of its name) as an argument:

```
cargo xtask test test_does_something
```

You can run the tests under the debugger (either `lldb` or `gdb`) using the `-g` flag:

```
cargo xtask test -g test_does_something
```

Part of the Tree-sitter test suite involves parsing the *corpus* tests for several languages and performing randomized edits to each example in the corpus. If you just want to run the tests for a particular *language*, you can pass the `-l` flag. Additionally, if you want to run a particular *example* from the corpus, you can pass the `-e` flag:

```
cargo xtask test -l javascript -e Arrays
```

If you are using `lldb` to debug the C library, tree-sitter provides custom pretty printers for several of its types. You can enable these helpers by importing them:

```
(lldb) command script import /path/to/tree-
sitter/lib/lldb_pretty_printers/tree_sitter_types.py
```

Published Packages

The main `tree-sitter/tree-sitter` repository contains the source code for several packages that are published to package registries for different languages:

- Rust crates on crates.io:
  - `tree-sitter` — A Rust binding to the core library
  - `tree-sitter-highlight` — The syntax-highlighting library
  - `tree-sitter-cli` — The command-line tool
- JavaScript modules on npmjs.com:
  - `web-tree-sitter` — A Wasm-based JavaScript binding to the core library
  - `tree-sitter-cli` — The command-line tool

There are also several other dependent repositories that contain other published packages:

- `tree-sitter/node-tree-sitter` — Node.js bindings to the core library, published as `tree-sitter` on npmjs.com
- `tree-sitter/py-tree-sitter` — Python bindings to the core library, published as `tree-sitter` on PyPI.org.
- `tree-sitter/go-tree-sitter` — Go bindings to the core library, published as `tree_sitter` on pkg.go.dev.

Developing Documentation

Our current static site generator for documentation is `mdBook`, with a little bit of custom JavaScript to handle the playground page. Most of the documentation is written in Markdown, including this file! You can find these files at `docs/src`. If you'd like to submit a PR to improve the documentation, navigate to the page you'd like to edit and hit the edit icon at the top right of the page.

Prerequisites for Local Development

> **Note**
>
> We're assuming you have `cargo` installed, the Rust package manager.

To run and iterate on the docs locally, the `mdbook` CLI tool is required, which can be installed with

```
cargo install mdbook
```

You might have noticed we have some fancy admonitions sprinkled throughout the documentation, like the note above. These are created using `mdbook-admonish`, a preprocessor for `mdBook`. As such,

this is also a requirement for developing the documentation locally. To install it, run:

```
cargo install mdbook-admonish
```

Once you've installed it, you can begin using admonitions in your markdown files. See the reference for more information.

Spinning it up

Now that you've installed the prerequisites, you can run the following command to start a local server:

```
cd docs
mdbook serve --open
```

`mdbook` has a live-reload feature, so any changes you make to the markdown files will be reflected in the browser after a short delay. Once you've made a change that you're happy with, you can submit a PR with your changes.

Improving the Playground

The playground page is a little more complicated, but if you know some basic JavaScript and CSS you should be able to make changes. The playground code can be found in `docs/src/assets/js/playground.js`, and its corresponding css at `docs/src/assets/css/playground.css`. The editor of choice we use for the playground is CodeMirror, and the tree-sitter module is fetched from here. This, along with the Wasm module and Wasm parsers, live in the .github.io repo.

Syntax Tree Playground

CLI Overview

The `tree-sitter` command-line interface is used to create, manage, test, and build tree-sitter parsers. It is controlled by

- a personal `tree-sitter/config.json` config file generated by `tree-sitter init-config`
- a parser `tree-sitter.json` config file generated by `tree-sitter init`.

tree-sitter init-config

This command initializes a configuration file for the Tree-sitter CLI.

```
tree-sitter init-config
```

These directories are created in the "default" location for your platform:

- On Unix, `$XDG_CONFIG_HOME/tree-sitter` or `$HOME/.config/tree-sitter`
- On Windows, `%APPDATA%\tree-sitter` or `$HOME\AppData\Roaming\tree-sitter`

> **Info**
>
> The CLI will work if there's no config file present, falling back on default values for each configuration option.

When you run the `init-config` command, it will print out the location of the file that it creates so that you can easily find and modify it.

The configuration file is a JSON file that contains the following fields:

parser-directories

The `tree-sitter highlight` command takes one or more file paths, and tries to automatically determine, which language should be used to highlight those files. To do this, it needs to know *where* to look for Tree-sitter grammars on your filesystem. You can control this using the `"parser-directories"` key in your configuration file:

```
{
  "parser-directories": [
    "/Users/my-name/code",
    "~/other-code",
    "$HOME/another-code"
  ]
}
```

Any folder within one of these *parser directories* whose name begins with `tree-sitter-` will be treated as a Tree-sitter grammar repository.

theme

The Tree-sitter highlighting system works by annotating ranges of source code with logical "highlight names" like `function.method`, `type.builtin`, `keyword`, etc. To decide what *color* should be used for rendering each highlight, a *theme* is needed.

In your config file, the `"theme"` value is an object whose keys are dot-separated highlight names like `function.builtin` or `keyword`, and whose values are JSON expressions that represent text styling parameters.

Highlight Names

A theme can contain multiple keys that share a common subsequence. Examples:

- `variable` and `variable.parameter`
- `function`, `function.builtin`, and `function.method`

For a given highlight produced, styling will be determined based on the **longest matching theme key**. For example, the highlight `function.builtin.static` would match the key `function.builtin` rather than `function`.

Styling Values

Styling values can be any of the following:

- Integers from 0 to 255, representing ANSI terminal color ids.
- Strings like `"#e45649"` representing hexadecimal RGB colors.
- Strings naming basic ANSI colors like `"red"`, `"black"`, `"purple"`, or `"cyan"`.
- Objects with the following keys:
  - `color` — An integer or string as described above.
  - `underline` — A boolean indicating whether the text should be underlined.
  - `italic` — A boolean indicating whether the text should be italicized.
  - `bold` — A boolean indicating whether the text should be bold-face.

An example theme can be seen below:

```
{
  "function": 26,
  "operator": {
    "bold": true,
    "color": 239
  },
  "variable.builtin": {
    "bold": true
  },
  "variable.parameter": {
    "underline": true
  },
  "type.builtin": {
    "color": 23,
    "bold": true
  },
  "keyword": 56,
  "type": 23,
  "number": {
    "bold": true,
    "color": 94
  },
  "constant": 94,
  "attribute": {
    "color": 124,
    "italic": true
  },
  "comment": {
    "color": 245,
    "italic": true
  },
  "constant.builtin": {
    "color": 94,
    "bold": true
  },
}
```

parse-theme

The `tree-sitter parse` command will output a pretty-printed CST when the `-c/--cst` option is used. You can control what colors are used for various parts of the tree in your configuration file.

> **Note**
>
> Omitting a field will cause the relevant text to be rendered with its default color.

An example parse theme can be seen below:

```
{
  "parse-theme": {
    // The color of node kinds
    "node-kind": [20, 20, 20],
    // The color of text associated with a node
    "node-text": [255, 255, 255],
    // The color of node fields
    "field": [42, 42, 42],
    // The color of the range information for unnamed nodes
    "row-color": [255, 255, 255],
    // The color of the range information for named nodes
    "row-color-named": [255, 130, 0],
    // The color of extra nodes
    "extra": [255, 0, 255],
    // The color of ERROR nodes
    "error": [255, 0, 0],
    // The color of MISSING nodes and their associated text
    "missing": [153, 75, 0],
    // The color of newline characters
    "line-feed": [150, 150, 150],
    // The color of backtick characters
    "backtick": [0, 200, 0],
    // The color of literals
    "literal": [0, 0, 200],
  }
}
```

`tree-sitter init`

The `init` command is your starting point for creating a new grammar. When you run it, it sets up a repository with all the essential files and structure needed for grammar development. Since the command includes git-related files by default, we recommend using git for version control of your grammar.

```
tree-sitter init [OPTIONS] # Aliases: i
```

Generated files

Required files

The following required files are always created if missing:

- `tree-sitter.json` - The main configuration file that determines how `tree-sitter` interacts with the grammar. If missing, the `init` command will prompt the user for the required fields. See below for the full documentation of the structure of this file.
- `package.json` - The `npm` manifest for the parser. This file is required for some `tree-sitter` subcommands, and if the grammar has dependencies (e.g., another published base grammar that this grammar extends).
- `grammar.js` - An empty template for the main grammar file; see the section on creating parsers.

Language bindings

Language bindings are files that allow your parser to be directly used by projects written in the respective language. The following bindings are created if enabled in `tree-sitter.json`:

C/C++

- `Makefile` — This file tells `make` how to compile your language.
- `CMakeLists.txt` — This file tells `cmake` how to compile your language.
- `bindings/c/tree_sitter/tree-sitter-language.h` — This file provides the C interface of your language.
- `bindings/c/tree-sitter-language.pc` — This file provides pkg-config metadata about your language's C library.

Go

- `go.mod` — This file is the manifest of the Go module.
- `bindings/go/binding.go` — This file wraps your language in a Go module.
- `bindings/go/binding_test.go` — This file contains a test for the Go package.

Node

- `binding.gyp` — This file tells Node.js how to compile your language.
- `bindings/node/binding.cc` — This file wraps your language in a JavaScript module for Node.js.
- `bindings/node/index.js` — This is the file that Node.js initially loads when using your language.
- `bindings/node/index.d.ts` — This file provides type hints for your parser when used in TypeScript.
- `bindings/node/binding_test.js` — This file contains a test for the Node.js package.

Java

- `pom.xml` - This file is the manifest of the Maven package.
- `bindings/java/main/namespace/language/TreeSitterLanguage.java` - This file wraps your language in a Java class.
- `bindings/java/test/TreeSitterLanguageTest.java` - This file contains a test for the Java package.

Python

- `pyproject.toml` — This file is the manifest of the Python package.
- `setup.py` — This file tells Python how to compile your language.
- `bindings/python/tree_sitter_language/binding.c` — This file wraps your language in a Python module.
- `bindings/python/tree_sitter_language/__init__.py` — This file tells Python how to load your language.
- `bindings/python/tree_sitter_language/__init__.pyi` — This file provides type hints for your parser when used in Python.
- `bindings/python/tree_sitter_language/py.typed` — This file provides type hints for your parser when used in Python.
- `bindings/python/tests/test_binding.py` — This file contains a test for the Python package.

Rust

- `Cargo.toml` — This file is the manifest of the Rust package.
- `bindings/rust/build.rs` — This file tells Rust how to compile your language.
- `bindings/rust/lib.rs` — This file wraps your language in a Rust crate when used in Rust.

Swift

- `Package.swift` — This file tells Swift how to compile your language.
- `bindings/swift/TreeSitterLanguage/language.h` — This file wraps your language in a Swift module when used in Swift.
- `bindings/swift/TreeSitterLanguageTests/TreeSitterLanguageTests.swift` — This file contains a test for the Swift package.

Zig

- `build.zig` - This file tells Zig how to compile your language.
- `build.zig.zon` - This file is the manifest of the Zig package.
- `bindings/zig/root.zig` - This file wraps your language in a Zig module.
- `bindings/zig/test.zig` - This file contains a test for the Zig package.

Additional files

In addition, the following files are created that aim to improve the development experience:

- `.editorconfig` — This file tells your editor how to format your code. More information about this file can be found here.
- `.gitattributes` — This file tells Git how to handle line endings and tells GitHub which files are generated.
- `.gitignore` — This file tells Git which files to ignore when committing changes.

Structure of `tree-sitter.json`

The `grammars` field

This field is an array of objects, though you typically only need one object in this array unless your repo has multiple grammars (for example, `Typescript` and `TSX`), e.g.,

```
{
  "tree-sitter": [
    {
      "scope": "source.ruby",
      "file-types": [
        "rb",
        "gemspec",
        "Gemfile",
        "Rakefile"
      ],
      "first-line-regex": "#!.*\\bruby$"
    }
  ]
}
```

Basic fields

These keys specify basic information about the parser:

- `scope` (required) — A string like `"source.js"` that identifies the language. We strive to match the scope names used by popular TextMate grammars and by the Linguist library.
- `path` — A relative path from the directory containing `tree-sitter.json` to another directory containing the `src/` folder, which contains the actual generated parser. The default value is `"."` (so that `src/` is in the same folder as `tree-sitter.json`), and this very rarely needs to be overridden.
- `external-files` — A list of relative paths from the root dir of a parser to files that should be checked for modifications during recompilation. This is useful during development to have changes to other files besides scanner.c be picked up by the cli.

Language detection

These keys help to decide whether the language applies to a given file:

- `file-types` — An array of filename suffix strings (not including the dot). The grammar will be used for files whose names end with one of these suffixes. Note that the suffix may match an *entire* filename.
- `first-line-regex` — A regex pattern that will be tested against the first line of a file to determine whether this language applies to the file. If present, this regex will be used for any file whose language does not match any grammar's `file-types`.
- `content-regex` — A regex pattern that will be tested against the contents of the file to break ties in cases where multiple grammars matched the file using the above two criteria. If the

regex matches, this grammar will be preferred over another grammar with no `content-regex`. If the regex does not match, a grammar with no `content-regex` will be preferred over this one.

- `injection-regex` — A regex pattern that will be tested against a *language name* to determine whether this language should be used for a potential *language injection* site. Language injection is described in more detail in the relevant section.

Query paths

These keys specify relative paths from the directory containing `tree-sitter.json` to the files that control syntax highlighting:

- `highlights` — Path to a *highlight query*. Default: `queries/highlights.scm`
- `locals` — Path to a *local variable query*. Default: `queries/locals.scm`.
- `injections` — Path to an *injection query*. Default: `queries/injections.scm`.
- `tags` — Path to a *tag query*. Default: `queries/tags.scm`.

The `metadata` field

This field contains information that tree-sitter will use to populate relevant bindings' files, especially their versions. Typically, this will all be set up when you run `tree-sitter init`, but you are welcome to update it as you see fit.

- `version` (required) — The current version of your grammar, which should follow semver
- `license` — The license of your grammar, which should be a valid SPDX license
- `description` — The brief description of your grammar
- `authors` (required) — An array of objects that contain a `name` field, and optionally an `email` and `url` field. Each field is a string
- `links` — An object that contains a `repository` field, and optionally a `funding` field. Each field is a string
- `namespace` — The namespace for the `Java` and `Kotlin` bindings, defaults to `io.github.tree-sitter` if not provided

The `bindings` field

This field controls what bindings are generated when the `init` command is run. Each key is a language name, and the value is a boolean.

- `c` (default: `true`)
- `go` (default: `true`)
- `java` (default: `false`)
- `node` (default: `true`)
- `python` (default: `true`)
- `rust` (default: `true`)
- `swift` (default: `false`)
- `zig` (default: `false`)

Options

`-u/--update`

Update outdated generated files, if possible.

Note: Existing files that may have been edited manually are *not* updated in general. To force an update to such files, remove them and call `tree-sitter init -u` again.

`-p/--grammar-path <PATH>`

The path to the directory containing the grammar.

tree-sitter generate

The most important command for grammar development is `tree-sitter generate`, which reads the grammar in structured form and outputs C files that can be compiled into a shared or static library (e.g., using the `build` command).

```
tree-sitter generate [OPTIONS] [GRAMMAR_PATH] # Aliases: gen, g
```

The optional `GRAMMAR_PATH` argument should point to the structured grammar, in one of two forms:

- `grammar.js` a (ESM or CJS) JavaScript file; if the argument is omitted, it defaults to `./grammar.js`.
- `grammar.json` a structured representation of the grammar that is created as a byproduct of `generate`; this can be used to regenerate a missing `parser.c` without requiring a JavaScript runtime (useful when distributing parsers to consumers).

If there is an ambiguity or *local ambiguity* in your grammar, Tree-sitter will detect it during parser generation, and it will exit with a `Unresolved conflict` error message. To learn more about conflicts and how to handle them, see the section on `Structuring Rules Well` in the user guide.

Generated files

- `src/parser.c` implements the parser logic specified in the grammar.
- `src/tree_sitter/parser.h` provides basic C definitions that are used in the generated `parser.c` file.
- `src/tree_sitter/alloc.h` provides memory allocation macros that can be used in an external scanner.
- `src/tree_sitter/array.h` provides array macros that can be used in an external scanner.
- `src/grammar.json` contains a structured representation of the grammar; can be used to regenerate the parser without having to re-evaluate the `grammar.js`.
- `src/node-types.json` provides type information about individual syntax nodes; see the section on `Static Node Types`.

Options

`-l/--log`

Print the log of the parser generation process. This includes information such as what tokens are included in the error recovery state, what keywords were extracted, what states were split and why, and the entry point state.

`--abi <VERSION>`

The ABI to use for parser generation. The default is ABI 15, with ABI 14 being a supported target.

`--no-parser`

Only generate `grammar.json` and `node-types.json`

`-o/--output`

The directory to place the generated parser in. The default is `src/` in the current directory.

`--report-states-for-rule <RULE>`

Print the overview of states from the given rule. This is useful for debugging and understanding the generated parser's item sets for all given states in a given rule. To solely view state count numbers for rules, pass in `-` for the rule argument. To view the overview of states for every rule, pass in `*` for the rule argument.

`--json-summary`

Report conflicts in a JSON format.

`--js-runtime <EXECUTABLE>`

The path to the JavaScript runtime executable to use when generating the parser. The default is `node`. Note that you can also set this with `TREE_SITTER_JS_RUNTIME`. Starting from version 0.26, you can also pass in `native` to use the experimental native QuickJS runtime that comes bundled with the CLI. This avoids the dependency on a JavaScript runtime entirely. The native QuickJS runtime is compatible with ESM as well as with CommonJS in strict mode. If your grammar depends on `npm` to

install dependencies such as base grammars, the native runtime can be used *after* running `npm install`.

`--disable-optimization`

Disable optimizations when generating the parser. Currently, this only affects the merging of compatible parse states.

tree-sitter build

The `build` command compiles your parser into a dynamically-loadable library, either as a shared object (`.so`, `.dylib`, or `.dll`) or as a Wasm module.

```
tree-sitter build [OPTIONS] [PATH] # Aliases: b
```

You can change the compiler executable via the `CC` environment variable and add extra flags via `CFLAGS`. For macOS or iOS, you can set `MACOSX_DEPLOYMENT_TARGET` or `IPHONEOS_DEPLOYMENT_TARGET` respectively to define the minimum supported version.

The path argument allows you to specify the directory of the parser to build. If you don't supply this argument, the CLI will attempt to build the parser in the current working directory.

On Unix platforms, the CLI will attempt to use `nm` to validate the symbols included in your parser library. Note that this safety check is *not* performed on Windows. If performed, this check ensures that:

- All non tree-sitter functions are marked static, to avoid conflicts when building with another tree-sitter project
- If an external scanner is used by your parser, all of the following symbols are present:
  - `tree_sitter_<name>_external_scanner_create`
  - `tree_sitter_<name>_external_scanner_destroy`
  - `tree_sitter_<name>_external_scanner_serialize`
  - `tree_sitter_<name>_external_scanner_deserialize`
  - `tree_sitter_<name>_external_scanner_scan`

Options

`-w/--wasm`

Compile the parser as a Wasm module. This command looks for the Wasi SDK indicated by the `TREE_SITTER_WASI_SDK_PATH` environment variable. If you don't have the binary, the CLI will attempt to download it for you to `<CACHE_DIR>/tree-sitter/wasi-sdk/`, where `<CACHE_DIR>` is resolved according to the XDG base directory or Window's Known_Folder_Locations.

`-o/--output`

Specify where to output the shared object file (native or Wasm). This flag accepts either an absolute path or a relative path. If you don't supply this flag, the CLI will attempt to figure out what the language name is based on the parent directory name to use for the output file. If the CLI can't figure it out, it will default to `parser`, thus generating `parser.so` or `parser.wasm` in the current working directory.

`--reuse-allocator`

Reuse the allocator that's set in the core library for the parser's external scanner. This is useful in applications where the author overrides the default allocator with their own, and wants to ensure every parser that allocates memory in the external scanner does so using their allocator.

`-0/--debug`

Compile the parser with debug flags enabled. This is useful when debugging issues that require a debugger like `gdb` or `lldb`.

`-v/--verbose`

Display verbose build information including working directory (if present), compiler, arguments, and environment variables.

tree-sitter parse

The `parse` command parses source files using a Tree-sitter parser. You can pass any number of file paths and glob patterns to `tree-sitter parse`, and it will parse all the given files. If no paths are provided, input will be parsed from stdin. The command will exit with a non-zero status code if any parse errors occurred.

```
tree-sitter parse [OPTIONS] [PATHS]... # Aliases: p
```

Options

`--paths <PATHS_FILE>`

The path to a file that contains paths to source files to parse.

`-p/--grammar-path <PATH>`

The path to the directory containing the grammar.

`-l/--lib-path`

The path to the parser's dynamic library. This is used instead of the cached or automatically generated dynamic library.

`--lang-name`

If `--lib-path` is used, the name of the language used to extract the library's language function

`--scope <SCOPE>`

The language scope to use for parsing. This is useful when the language is ambiguous.

`-d/--debug`

Outputs parsing and lexing logs. This logs to stderr.

`-0/--debug-build`

Compile the parser with debug flags enabled. This is useful when debugging issues that require a debugger like `gdb` or `lldb`.

`-D/--debug-graph`

Outputs logs of the graphs of the stack and parse trees during parsing, as well as the actual parsing and lexing message. The graphs are constructed with graphviz dot, and the output is written to `log.html`.

`--wasm`

Compile and run the parser as a Wasm module (only if the tree-sitter CLI was built with `--features=wasm`).

`--dot`

Output the parse tree with graphviz dot.

`-x/--xml`

Output the parse tree in XML format.

`-c/--cst`

Output the parse tree in a pretty-printed CST format.

`-s/--stat`

Show parsing statistics.

`--timeout <TIMEOUT>`

Set the timeout for parsing a single file, in microseconds.

`-t/--time`

Print the time taken to parse the file. If edits are provided, this will also print the time taken to parse the file after each edit.

`-q/--quiet`

Suppress main output.

`--edits <EDITS>...`

Apply edits after parsing the file. Edits are in the form of `row,col|position delcount insert_text` where row and col, or position are 0-indexed.

`--encoding <ENCODING>`

Set the encoding of the input file. By default, the CLI will look for the `BOM` to determine if the file is encoded in `UTF-16BE` or `UTF-16LE`. If no `BOM` is present, `UTF-8` is the default. One of `utf8`, `utf16-le`, `utf16-be`.

`--open-log`

When using the `--debug-graph` option, open the log file in the default browser.

`-j/--json-summary`

Output parsing results in a JSON format.

`--config-path <CONFIG_PATH>`

The path to an alternative configuration (`config.json`) file. See the init-config command for more information.

`-n/--test-number <TEST_NUMBER>`

Parse a specific test in the corpus. The test number is the same number that appears in the output of `tree-sitter test`.

`-r/--rebuild`

Force a rebuild of the parser before running tests.

`--no-ranges`

Omit the node's ranges from the default parse output. This is useful when copying S-Expressions to a test file.

tree-sitter test

The `test` command is used to run the test suite for a parser.

```
tree-sitter test [OPTIONS] # Aliases: t
```

Options

-i/--include <INCLUDE>

Only run tests whose names match this regex.

-e/--exclude <EXCLUDE>

Skip tests whose names match this regex.

--file-name <NAME>

Only run tests from the given filename in the corpus.

-p/--grammar-path <PATH>

The path to the directory containing the grammar.

--lib-path

The path to the parser's dynamic library. This is used instead of the cached or automatically generated dynamic library.

--lang-name

If `--lib-path` is used, the name of the language used to extract the library's language function

-u/--update

Update the expected output of tests.

> **Info**
>
> Tests containing `ERROR` nodes or `MISSING` nodes will not be updated.

-d/--debug

Outputs parsing and lexing logs. This logs to stderr.

-0/--debug-build

Compile the parser with debug flags enabled. This is useful when debugging issues that require a debugger like `gdb` or `lldb`.

-D/--debug-graph

Outputs logs of the graphs of the stack and parse trees during parsing, as well as the actual parsing and lexing message. The graphs are constructed with [graphviz dot][dot], and the output is written to `log.html`.

--wasm

Compile and run the parser as a Wasm module (only if the tree-sitter CLI was built with `--features=wasm`).

--open-log

When using the `--debug-graph` option, open the log file in the default browser.

`--config-path <CONFIG_PATH>`

The path to an alternative configuration (`config.json`) file. See the init-config command for more information.

`--show-fields`

Force showing fields in test diffs.

`--stat <STAT>`

Show parsing statistics when tests are being run. One of `all` , `outliers-and-total` , or `total-only` .

- `all` : Show statistics for every test.
- `outliers-and-total` : Show statistics only for outliers, and total statistics.
- `total-only` : Show only total statistics.

`-r/--rebuild`

Force a rebuild of the parser before running tests.

`--overview-only`

Only show the overview of the test results, and not the diff.

`--json-summary`

Output the test summary in a JSON format.

tree-sitter version

The `version` command upgrades the version of your grammar.

```
tree-sitter version <VERSION> # Aliases: publish
```

This will update the version in several files, if they exist:

- tree-sitter.json
- Cargo.toml
- Cargo.lock
- package.json
- package-lock.json
- Makefile
- CMakeLists.txt
- pyproject.toml

Alternative forms can use the version in `tree-sitter.json` to bump automatically:

```
tree-sitter version --bump patch # patch bump
tree-sitter version --bump minor # minor bump
tree-sitter version --bump major # major bump
```

As a grammar author, you should keep the version of your grammar in sync across different bindings. However, doing so manually is error-prone and tedious, so this command takes care of the burden. If you are using a version control system, it is recommended to commit the changes made by this command, and to tag the commit with the new version.

To print the current version without bumping it, use:

```
tree-sitter version
```

Note that some of the binding updates require access to external tooling:

- Updating Cargo.toml and Cargo.lock bindings requires that `cargo` is installed.
- Updating package-lock.json requires that `npm` is installed.

Options

`-p/--grammar-path <PATH>`

The path to the directory containing the grammar.

`--bump`

Automatically bump the version. Possible values are:

- `patch` : Bump the patch version.
- `minor` : Bump the minor version.
- `major` : Bump the major version.

tree-sitter fuzz

The `fuzz` command is used to fuzz a parser by performing random edits and ensuring that undoing these edits results in consistent parse trees. It will fail if the parse trees are not equal, or if the changed ranges are inconsistent.

```
tree-sitter fuzz [OPTIONS] # Aliases: f
```

Options

-s/--skip <SKIP>

A list of test names to skip fuzzing.

--subdir <SUBDIR>

The directory containing the parser. This is primarily useful in multi-language repositories.

-p/--grammar-path

The path to the directory containing the grammar.

--lib-path

The path to the parser's dynamic library. This is used instead of the cached or automatically generated dynamic library.

--lang-name

If `--lib-path` is used, the name of the language used to extract the library's language function

--edits <EDITS>

The maximum number of edits to perform. The default is 3. This value can also be set via the `TREE_SITTER_EDITS` environment variable.

--iterations <ITERATIONS>

The number of iterations to run. The default is 10. This value can also be set via the `TREE_SITTER_ITERATIONS` environment variable.

-i/--include <INCLUDE>

Only run tests whose names match this regex.

-e/--exclude <EXCLUDE>

Skip tests whose names match this regex.

--log-graphs

Outputs logs of the graphs of the stack and parse trees during parsing, as well as the actual parsing and lexing message. The graphs are constructed with graphviz dot, and the output is written to `log.html`.

-l/--log

Outputs parsing and lexing logs. This logs to stderr.

-r/--rebuild

Force a rebuild of the parser before running the fuzzer.

tree-sitter query

The `query` command is used to run a query on a parser, and view the results.

```
tree-sitter query [OPTIONS] <QUERY_PATH> [PATHS]... # Aliases: q
```

Options

-p/--grammar-path <PATH>

The path to the directory containing the grammar.

--lib-path

The path to the parser's dynamic library. This is used instead of the cached or automatically generated dynamic library.

--lang-name

If `--lib-path` is used, the name of the language used to extract the library's language function

-t/--time

Print the time taken to execute the query on the file.

-q/--quiet

Suppress main output.

--paths <PATHS_FILE>

The path to a file that contains paths to source files in which the query will be executed.

--byte-range <BYTE_RANGE>

The range of byte offsets in which the query will be executed. The format is `start_byte:end_byte`.

--containing-byte-range <BYTE_RANGE>

The range of byte offsets in which the query will be executed. Only the matches that are fully contained within the provided byte range will be returned.

--row-range <ROW_RANGE>

The range of rows in which the query will be executed. The format is `start_row:end_row`.

--containing-row-range <ROW_RANGE>

The range of rows in which the query will be executed. Only the matches that are fully contained within the provided row range will be returned.

--scope <SCOPE>

The language scope to use for parsing and querying. This is useful when the language is ambiguous.

-c/--captures

Order the query results by captures instead of matches.

--test

Whether to run query tests or not.

--config-path <CONFIG_PATH>

The path to an alternative configuration (`config.json`) file. See the init-config command for more information.

`-n/--test-number <TEST_NUMBER>`

Query the contents of a specific test.

`-r/--rebuild`

Force a rebuild of the parser before executing the query.

tree-sitter highlight

You can run syntax highlighting on an arbitrary file using `tree-sitter highlight`. This can either output colors directly to your terminal using ANSI escape codes, or produce HTML (if the `--html` flag is passed). For more information, see the syntax highlighting page.

```
tree-sitter highlight [OPTIONS] [PATHS]... # Aliases: hi
```

Options

-H/--html

Output an HTML document with syntax highlighting.

--css-classes

Output HTML with CSS classes instead of inline styles.

--check

Check that the highlighting captures conform strictly to the standards.

--captures-path <CAPTURES_PATH>

The path to a file with captures. These captures would be considered the "standard" captures to compare against.

--query-paths <QUERY_PATHS>

The paths to query files to use for syntax highlighting. These should end in `highlights.scm`.

--scope <SCOPE>

The language scope to use for syntax highlighting. This is useful when the language is ambiguous.

-t/--time

Print the time taken to highlight the file.

-q/--quiet

Suppress main output.

--paths <PATHS_FILE>

The path to a file that contains paths to source files to highlight

-p/--grammar-path <PATH>

The path to the directory containing the grammar.

--config-path <CONFIG_PATH>

The path to an alternative configuration (`config.json`) file. See the init-config command for more information.

-n/--test-number <TEST_NUMBER>

Highlight the contents of a specific test.

-r/--rebuild

Force a rebuild of the parser before running the fuzzer.

tree-sitter tags

You can run symbol tagging on an arbitrary file using `tree-sitter tags`. This will output a list of tags. For more information, see the code navigation page.

```
tree-sitter tags [OPTIONS] [PATHS]...
```

Options

`--scope <SCOPE>`

The language scope to use for symbol tagging. This is useful when the language is ambiguous.

`-t/--time`

Print the time taken to generate tags for the file.

`-q/--quiet`

Suppress main output.

`--paths <PATHS_FILE>`

The path to a file that contains paths to source files to tag.

`-p/--grammar-path <PATH>`

The path to the directory containing the grammar.

`--config-path <CONFIG_PATH>`

The path to an alternative configuration (`config.json`) file. See the init-config command for more information.

`-n/--test-number <TEST_NUMBER>`

Generate tags from the contents of a specific test.

`-r/--rebuild`

Force a rebuild of the parser before running the tags.

tree-sitter playground

The `playground` command allows you to start a local playground to test your parser interactively.

```
tree-sitter playground [OPTIONS] # Aliases: play, pg, web-ui
```

> **Note**
>
> For this to work, you must have already built the parser as a Wasm module. This can be done with the `build` subcommand (`tree-sitter build --wasm`).

Options

`-q/--quiet`

Don't automatically open the playground in the default browser.

`--grammar-path <GRAMMAR_PATH>`

The path to the directory containing the grammar and wasm files.

`-e/--export <EXPORT_PATH>`

Export static playground files to the specified directory instead of serving them.

tree-sitter dump-languages

The `dump-languages` command prints out a list of all the languages that the CLI knows about. This can be useful for debugging purposes, or for scripting. The paths to search comes from the config file's `parser-directories` object.

```
tree-sitter dump-languages [OPTIONS] # Aliases: langs
```

Options

`--config-path`

The path to the configuration file. Ordinarily, the CLI will use the default location as explained in the init-config command. This flag allows you to explicitly override that default, and use a config defined elsewhere.

tree-sitter complete

The `complete` command generates a completion script for your shell. This script can be used to enable autocompletion for the `tree-sitter` CLI.

```
tree-sitter complete --shell <SHELL> # Aliases: comp
```

Options

`--shell <SHELL>`

The shell for which to generate the completion script.

Supported values: `bash`, `elvish`, `fish`, `power-shell`, `zsh`, and `nushell`.