tree_sitter_graph

Module reference

This section defines the graph DSL implemented by this library.

Overview

You can use tree-sitter to parse the content of source code into a *concrete syntax tree*. For instance, using the tree-sitter-python grammar, you can parse Python source code:

```
# test.py
from one.two import d, e.c
import three
print(d, e.c)
print three.f


$ tree-sitter parse test.py
(module [0, 0] - [4, 0]
  (import_from_statement [0, 0] - [0, 26]
    module_name: (dotted_name [0, 5] - [0, 12]
      (identifier [0, 5] - [0, 8])
      (identifier [0, 9] - [0, 12]))
    name: (dotted_name [0, 20] - [0, 21]
      (identifier [0, 20] - [0, 21]))
    name: (dotted_name [0, 23] - [0, 26]
      (identifier [0, 23] - [0, 24])
      (identifier [0, 25] - [0, 26])))
  (import_statement [1, 0] - [1, 12]
    name: (dotted_name [1, 7] - [1, 12]
      (identifier [1, 7] - [1, 12])))
  (expression_statement [2, 0] - [2, 13]
    (call [2, 0] - [2, 13]
      function: (identifier [2, 0] - [2, 5])
      arguments: (argument_list [2, 5] - [2, 13]
        (identifier [2, 6] - [2, 7])
        (attribute [2, 9] - [2, 12]
          object: (identifier [2, 9] - [2, 10])
          attribute: (identifier [2, 11] - [2, 12])))))
  (print_statement [3, 0] - [3, 13]
    argument: (attribute [3, 6] - [3, 13]
      object: (identifier [3, 6] - [3, 11])
      attribute: (identifier [3, 12] - [3, 13])))
```

There are many interesting use cases where you want to use this parsed syntax tree to create some *other* graph-like structure. This library lets you do that, using a declarative DSL to identify patterns in the parsed syntax tree, along with rules for which graph nodes and edges to create for the syntax nodes that match those patterns.

Terminology

One confusing aspect of the graph DSL is that we have to talk about two different kinds of "node": the nodes of the concrete syntax tree that is our input when executing a graph DSL file, and the nodes of the graph that we produce as an output. We will be careful to always use "syntax node" and "graph node" in this documentation to make it clear which kind of node we mean.

High-level structure

A graph DSL file consists of one or more ***stanzas***. Each stanza starts with a tree-sitter *query pattern*, which identifies a portion of the concrete syntax tree. The query pattern should use *captures* to identify particular syntax nodes in the matched portion of the tree. Following the query is a ***block***, which is a sequence of statements that construct ***graph nodes***, link them with ***edges***, and annotate both with ***attributes***.

Query patterns can be suffixed by quantification operators. If a pattern with a quantification suffix is captured, the suffix determines the capture value. In the case of `?`, the capture value is a null value, or a syntax node. In the case of `+` and `*`, the value is a list of syntax nodes.

Comments start with a semicolon, and extend to the end of the line.

Identifiers start with either an ASCII letter or underscore, and all remaining characters are ASCII letters, numbers, underscores, or hyphens. (More precisely, they satisfy the regular expression `/[a-zA-Z_][a-zA-Z0-9_-]*/`.) Identifiers are used as the names of attributes, functions, and variables.

To execute a graph DSL file against a concrete syntax tree, we execute each stanza in the graph DSL file exhaustively. For each stanza, we identify each place where the concrete syntax tree matches the query pattern. For each of these places, we end up with a different set of syntax nodes assigned to the query pattern's captures. We execute the block of statements for each of these capture assignments, creating any graph nodes, edges, or attributes mentioned in the block.

Regular execution will apply the stanzas *in order*, and it is important to make sure that scoped variables have been assigned before they are used. This is not a requirement when using the lazy evaluation strategy, which handles this implicitly. The lazy evaluation strategy is also more efficient when there are many stanzas, because it can reduce tree traversals. Therefore, using the lazy evaluation strategy is recommended, and will likely become the only supported strategy in future releases.

For instance, the following stanza would match all of the identifiers in our example syntax tree:

```
(identifier) @id
{
  ; Some statements that will be executed for each identifier in the
  ; source file.  You can use @id to refer to the matched identifier
  ; syntax node.
}
```

Expressions

The value of an expression in the graph DSL can be any of the following:

- null
- a boolean
- a string
- an integer (unsigned, 32 bits)
- a reference to a syntax node
- a reference to a graph node
- an ordered list of values
- a list comprehension
- an unordered set of values
- a set comprehension

The null value is spelled `#null`.

The boolean literals are spelled `#true` and `#false`.

String constants are enclosed in double quotes. They can contain backslash escapes `\\`, `\0`, `\n`, `\r`, and `\t`:

- `"a string"`
- `"a string with\na newline"`
- `"a string with\\a backslash"`

Integer constants are encoded in ASCII decimal:

- `0`
- `10`
- `42`

Lists consist of zero or more expressions, separated by commas, enclosed in square brackets. The elements of a list do not have to have the same type:

```
[0, "string", 0, #true, @id]
```

Sets have the same format, but are enclosed in curly braces instead of square brackets:

```
{0, "string", #true, @id}
```

Both lists and sets both allow trailing commas after the final element, if you prefer that style to play more nicely with source control diffs:

```
[
  0,
  "string",
  #true,
  @id,
]
```

List comprehensions allow mapping over a list and producing a new list with elements based on the given element expression:

```
[ (some-function x) for x in @xs ]
[ @x.scoped_var for x in @xs ]
```

Set comprehensions have similar syntax, but the resulting value will be a set instead of an ordered list:

```
{ (some-function x) for x in @xs }
{ @x.scoped_var for x in @xs }
```

List and set comprehensions are subject to the same restrictions as for loops, which means the list value that is iterated over must be local. It is therefore not possible to iterator over the value of a scoped variable. Using scoped variables in the element expression however is no problem.

Syntax nodes

Syntax nodes are identified by tree-sitter query captures (`@name`). For instance, in our example stanza, whose query is `(identifier) @id`, `@id` would refer to the `identifier` syntax node that the stanza matched against.

Unused query captures are considered errors, unless they start with an underscode. For example, a capture `@id` must be used within the stanza, but `@_id` does not.

Variables

You can use variables to pass information between different stanzas and statements in a graph DSL file. There are three kinds of variables:

- ***Global*** variables are provided externally by whatever process is executing the graph DSL file. You can use this, for instance, to pass in the path of the source file being analyzed, to use as part of the graph structure that you create.
- ***Local*** variables are only visible within the current execution of the current stanza. Once all of the statements in the stanza have been executed, all local variables disappear.
- ***Scoped*** variables are "attached to" syntax nodes. Their values carry over from stanza to stanza. Scoped variables are referenced by using a syntax node expression (typically a query capture) and a variable name, separated by a period: `@node.variable`.

Global variables are declared using a `global` declaration. The external process that executes the graph DSL file must provide values for all declared global variables. (It is not possible to define a global variable and give it a value from within the graph DSL.) The name of the global variable can be suffixed by a quantifier: '*' and '+' for lists, and '?' for optional values, which allows them to be used in iteration and conditional statements, respectively.

Local and scoped variables are created using `var` or `let` statements. A `let` statement creates an ***immutable variable***, whose value cannot be changed. A `var` statement creates a ***mutable variable***. You use a `set` statement to change the value of a mutable variable. Local variables are not allowed to have the same name as a declared global variable.

Local variables are block scoped. For example, a local variable defined in a `scan` arm is not visible in other scan arms, or after the `scan` statement. If you need to persist a value for use after a block, introduce a mutable variable before the block and assign to it inside the block.

```
global global_variable

(identifier) @id
{
  let local_variable = "a string"
  ; The following would be an error, since `let` variables are immutable:
  ; set local_variable = "a new value"

  ; The following is also an error, since you can't mutate a variable that
  ; doesn't exist:
  ; set missing_variable = 42

  ; The following is an error, since you cannot hide a global variable with
  ; a local one:
  ; let global_variable = "a new value"

  var mutable_variable = "first value"
  set mutable_variable = global_variable ; we can refer to the global variable

  var @id.kind = "id"
}
```

Variables can be referenced anywhere that you can provide an expression. It's an error if you try to reference a variable that hasn't been defined.

Functions

The process executing a graph DSL file can provide ***functions*** that can be called from within graph DSL stanzas.

Function calls use a Lisp-like syntax, where the name of the function being called is *inside* of the parentheses. The parameters to a function call are arbitrary expressions. For instance, if the executing process provides a function named `+`, you could call it as:

```
(identifier) @id
{
   let x = 4
   let @id.nine = (+ x 5)
}
```

Note that it's the process executing the graph DSL file that decides which functions are available. We do define a standard library, and most of the time those are the functions that are available, but you should double-check the documentation of whatever graph DSL tool you're using to make sure.

Graph nodes

You can use this graph DSL to create any graph structure that you want. There are no limitations on which graph nodes you create, nor on how you use edges to connect the graph nodes. You are not limited to creating a tree, and in particular, you are not limited to creating a tree that "lines" up with the parsed syntax tree.

There are two ways to create a new graph node. The first, and most common, is to use a `node` statement:

```
(identifier) @id
{
```

```
  node @id.node
}
```

This creates a graph node, and assigns a reference to the new node to a new immutable variable named `new_node`.

The second way to create a graph node is to call the `node` function:

```
(identifier) @id
{
  let @id.node = (node)
}
```

Note that these two examples do exactly the same thing! In fact, the `node` statement in the first example is just syntactic sugar for the `let` statement in the second example. Since the most common pattern is to create a graph node and immediately assign it to an immutable scoped variable, we provide the `node` statement as a convenient shorthand. If you need to do anything more complex, such as assigning the graph node reference to a *mutable* variable, you can call the `node` function directly.

By attaching a graph node to a syntax node using a scoped variable, you can refer to them from multiple stanzas:

```
(identifier) @id
{
  node @id.node
}

(dotted_name (identifier) @dotted_element)
{
  ; We will learn more about the attr statement below
  attr (@dotted_element.node) kind = "dotted"
}
```

In this example, we will create a graph node for *every* `identifier` syntax node. Each of those syntax nodes will have a `node` scoped variable, containing a reference to its graph node. But only the graph nodes of those identifiers that appear inside of a `dotted_name` will have a `kind` attribute. And even though we used different capture names, inside of different queries, to find those `identifier` nodes, the graph node references in both stanzas refer to the same graph nodes.

Edges

Edges are created via an `edge` statement, which specifies the two graph nodes that should be connected. Edges are directed, and the `->` arrow in the `edge` statement indicates the direction of the edge.

```
(import_statement name: (_) @name)
{
  node @name.source
  node @name.sink
  edge @name.source -> @name.sink
}
```

There can be at most one edge connecting any particular source and sink graph node in the graph. If multiple stanzas create edges between the same graph nodes, those are “collapsed” into a single edge.

Attributes

Graph nodes and edges have an associated set of ***attributes***. Each attribute has a name (which is an identifier), and a value.

You add attributes to a graph node or edge using an `attr` statement:

```
(import_statement name: (_) @name)
{
  node @name.source
  node @name.sink
  attr (@name.sink) kind = "module"
  attr (@name.source -> @name.sink) precedence = 10
}
```

Note that you have to have already created the graph node or edge, and the graph node or edge must not already have an attribute with the same name.

(Attributes might seem similar to scoped variables, but they are quite different. Attributes are attached to graph nodes and edges, while scoped variables are attached to syntax nodes. More importantly, scoped variables only exist while executing the graph DSL file. Once the execution has completed, the variables disappear. Attributes, on the other hand, are part of the output produced by the graph DSL file, and live on after execution has finished.)

Attribute shorthands

Commonly used combinations of attributes can be captured in ***shorthands***. Each shorthand defines the attribute name, a variable which captures the attribute value, and a list of attributes to which it expands.

Attribute shorthands are defined at the same level as stanzas. For example, the following shorthand takes a syntax node as argument, and expands to attributes for its source text and child index:

```
attribute node_props = node => node_text = (source-text node), node_index = (named-child-index node

(argument (_)@expr) {
```

```
  attr (@expr) node_props = @expr
}
```

Regular expressions

Starting at the beginning of the string, we determine the first character where one of the regular expressions matches. If more than one regular expression matches at this earliest character, we use the regular expression that appears first in the `scan` statement. We then execute the statements in the "winning" regular expression's block.

After executing the matching block, we try to match all of the regular expressions again, starting *after* the text that was just matched. We continue this process, applying the earliest matching regular expression in each iteration, until we have exhausted the entire string, or none of the regular expressions match.

Within each regular expression's block, you can use `$0`, `$1`, etc., to refer to any capture groups in the regular expression.

The value being scanned must be local, which means it cannot be derived from scoped variables.

For example, if `filepath` is a global variable containing the path of a Python source file, you could use the following `scan` statement to construct graph nodes for the name of the module defined in the file:

```
global filepath

(module) @mod
{
  var new_node = #null
  var current_node = (node)

  scan filepath {
    "([^/]+)/"
    {
      ; This arm will match any directory component of the file path.  In
      ; Python, this gives you the sequence of packages that the module
      ; lives in.

      ; Note that we keep appending additional tag names to the `current`
      ; graph node to create an arbitrary number of graph nodes linked to
      ; the @mod syntax node.
      set new_node = (node)
      attr (new_node) name = $1
      edge current_node -> new_node
      set current_node = new_node
    }

    "__init__\\.py$"
    {
      ; This arm will match a trailing __init__.py, indicating that the
      ; module's name comes from the last directory component.

      ; Expose the graph node that we created for that component as a
      ; scoped variable that later stanzas can see.
      let @mod.root = current_node
    }

    "([^/]+)\\.py$"
    {
      ; This arm will match any other trailing module name.  Note that
      ; __init__.py also matches this regular expression, but since it
      ; appears later, the __init__.py clause will take precedence.

      set new_node = (node)
      attr (new_node) name = $1
      edge current_node -> new_node
      let @mod.root = new_node
    }
  }
}
```

Conditionals

You can use `if` statements to make blocks of statements conditional on optional values. Conditions are comma-separated lists of clauses. The clause `some EXPRESSION` indicates that the optional value must be present. The clause `none EXPRESSION` indicates that the optional value is absent. A bare expression is evaluated as to boolean. All values in conditions must be local, which means they cannot be derived from scoped variables.

```
(lexical_declaration type:(_)? @type value:(_)? @value)
{
  if some @type, none @value {
    ; ...
  } elif some @value {
```

```
    ; ...
  } else {
    ; ...
  }
}
```

List iteration

You can use a `for` statement to execute blocks of statements for every element in list values. The list value must be local, which means it cannot be derived from scoped variables.

```
(module (_)* @stmts)
{
  for stmt in @stmts {
    print stmt
  }
}
```

Debugging

To support members of the Ancient and Harmonious Order of Printf Debuggers, you can use `print` statements to print out (to `stderr`) the content of any expressions during the execution of a graph DSL file:

```
(identifier) @id
{
   let x = 4
   print "Hi! x = ", x
}
```

Modules

functions	This section defines the standard library of functions available to graph DSL files.