Crate tree_sitter_stack_graphs

This crate lets you construct stack graphs using tree-sitter's graph construction DSL. The graph DSL lets you construct arbitrary graph structures from the parsed syntax tree of a source file. If you construct a graph using the vocabulary of attributes described below, then the result of executing the graph DSL will be a valid stack graph, which we can then use for name binding lookups.

Prerequisites

To process a particular source language, you will first need a tree-sitter grammar for that language. There are already tree-sitter grammars available for many languages. If you do not have a tree-sitter grammar for your language, you will need to create that first. (Check out the tree-sitter discussion forum if you have questions or need pointers on how to do that.)

You will then need to create *stack graph construction rules* for your language. These rules are implemented using tree-sitter's graph construction DSL. They define the particular stack graph nodes and edges that should be created for each part of the parsed syntax tree of a source file.

Graph DSL vocabulary

Please note: This documentation assumes you are already familiar with stack graphs, and how to use different stack graph node types, and the connectivity between nodes, to implement the name binding semantics of your language. We assume that you know what kind of stack graph you want to produce; this documentation focuses only on the mechanics of *how* to create that stack graph content.

As mentioned above, your stack graph construction rules should create stack graph nodes and edges from the parsed content of a source file. You will use TSG stanzas to match on different parts of the parsed syntax tree, and create stack graph content for each match.

Creating stack graph nodes

To create a stack graph node for each identifier in a Python file, you could use the following TSG stanza:

```
(identifier) {
  node new_node
}
```

(Here, `node` is a TSG statement that creates a new node, and `new_node` is the name of a local variable that the new node is assigned to, letting you refer to the new node in the rest of the stanza.)

By default, this new node will be a *scope node*. If you need to create a different kind of stack graph node, set the `type` attribute on the new node:

```
(identifier) {
  node new_node
  attr (new_node) type = "push_symbol"
}
```

The valid `type` values are:

- `drop_scopes`: a *drop scopes* node
- `pop_symbol`: a *pop symbol* node
- `pop_scoped_symbol`: a *pop scoped symbol* node
- `push_symbol`: a *push symbol* node
- `push_scoped_symbol`: a *push scoped symbol* node
- `scope`: a *scope* node

A node without an explicit `type` attribute is assumed to be of type `scope`.

Certain node types — `pop_symbol`, `pop_scoped_symbol`, `push_symbol` and `push_scoped_symbol` — also require you to provide a `symbol` attribute. Its value must be a string, but will typically come from the content of a parsed syntax node using the `source-text` function and a syntax capture:

```
(identifier) @id {
  node new_node
  attr (new_node) type = "push_symbol", symbol = (source-text @id)
}
```

Node types `pop_symbol` and `pop_scoped_symbol` allow an optional `is_definition` attribute, which marks that node as a proper definition. Node types `push_symbol` and `push_scoped_symbol` allow an optional `is_reference` attribute, which marks the node as a proper reference. When `is_definition` or `is_reference` are set, the `source_node` attribute is required.

```
(identifier) @id {
  node new_node
  attr (new_node) type = "push_symbol", symbol = (source-text @id), is_reference, source_node = @ic
}
```

A *push scoped symbol* node requires a `scope` attribute. Its value must be a reference to an `exported` node that you've already created. (This is the exported scope node that will be pushed onto the scope stack.) For instance:

```
(identifier) @id {
  node new_exported_scope_node
```

```
  attr (new_exported_scope_node) is_exported
  node new_push_scoped_symbol_node
  attr (new_push_scoped_symbol_node)
    type = "push_scoped_symbol",
    symbol = (source-text @id),
    scope = new_exported_scope_node
}
```

Nodes of type `scope` allow an optional `is_exported` attribute, that is required to use the scope in a `push_scoped_symbol` node.

Annotating nodes with location information

You can annotate any stack graph node that you create with location information, identifying the portion of the source file that the node "belongs to". This is *required* for definition and reference nodes, since the location information determines which parts of the source file the user can *click on*, and the *destination* of any code navigation queries the user makes. To do this, add a `source_node` attribute, whose value is a syntax node capture:

```
(function_definition name: (identifier) @id) @func {
  node def
  attr (def) type = "pop_symbol", symbol = (source-text @id), source_node = @func, is_definition
}
```

Note how in this example, we use a different syntax node for the *target* of the definition (the entirety of the function definition) and for the *name* of the definition (the content of the function's `name`).

Adding the `empty_source_span` attribute will use an empty source span located at the start of the span of the `source_node`. This can be useful when a proper reference or definition is desired, and thus `source_node` is required, but the span of the available source node is too large. For example, a module definition which is located at the start of the program, but does span the whole program:

```
(program)@prog {
  ; ...
  node mod_def
  attr mod_def type = "pop_symbol", symbol = mod_name, is_definition, source_node = @prog, empty_sc
  ; ...
}
```

Annotating nodes with syntax type information

You can annotate any stack graph node with information about its syntax type. To do this, add a `syntax_type` attribute, whose value is a string indicating the syntax type.

```
(function_definition name: (identifier) @id) @func {
  node def
  ; ...
  attr (def) syntax_type = "function"
}
```

Annotating definitions with definiens information

You cannot annotate definitions with a definiens, which is the thing the definition covers. For example, for a function definition, the definiens would be the function body. To do this, add a `definiens_node` attribute, whose value is a syntax node that spans the definiens.

```
(function_definition name: (identifier) @id body: (_) @body) @func {
  node def
  ; ...
  attr (def) definiens_node = @body
}
```

Definiens are optional and setting them to `#null` explicitly is allowed.

Connecting stack graph nodes with edges

To connect two stack graph nodes, use the `edge` statement to add an edge between them:

```
(function_definition name: (identifier) @id) @func {
  node def
  attr (def) type = "pop_symbol", symbol = (source-text @id), source_node = @func, is_definition
  node body
  edge def -> body
}
```

To implement shadowing (which determines which definitions are selected when multiple are available), you can add a `precedence` attribute to each edge to indicate which paths are prioritized:

```
(function_definition name: (identifier) @id) @func {
  node def
  attr (def) type = "pop_symbol", symbol = (source-text @id), source_node = @func, is_definition
  node body
  edge def -> body
```

```
  attr (def -> body) precedence = 1
}
```

(If you don't specify a `precedence`, the default is 0.)

Referring to the singleton nodes

The *root node* and *jump to scope node* are singleton nodes that always exist for all stack graphs. You can refer to them using the `ROOT_NODE` and `JUMP_TO_SCOPE_NODE` global variables:

```
global ROOT_NODE

(function_definition name: (identifier) @id) @func {
  node def
  attr (def) type = "pop_symbol", symbol = (source-text @id), source_node = @func, is_definition
  edge ROOT_NODE -> def
}
```

Attaching debug information to nodes

It is possible to attach extra information to nodes for debugging purposes. This is done by adding `debug_*` attributes to nodes. Each attribute defines a debug entry, with the key derived from the attribute name, and the value the string representation of the attribute value. For example, mark a scope node with a kind as follows:

```
(function_definition name: (identifier) @id) @func {
  ; ...
  node param_scope
  attr (param_scope) debug_kind = "param_scope"
  ; ...
}
```

Working with paths

Built-in path functions are available to compute symbols that depend on path information, such as module names or imports. The path of the file is provided in the global variable `FILE_PATH`.

The following path functions are available:

- `path-dir`: get the path consisting of all but the last component of the argument path, or `#null` if it ends in root
- `path-fileext`: get the file extension, i.e. everything after the final `.` of the file name of the argument path, or `#null` if it has extension
- `path-filename`: get the last component of the argument path, or `#null` if it has no final component
- `path-filestem`: get the file stem of the argument path, i.e., everything before the extension, or `#null` if it has no file name
- `path-join`: join all argument paths together
- `path-normalize`: normalize the argument path by eliminating `.` and `..` components where possible
- `path-split`: split the argument path into a list of its components

The following example computes a module name from a file path:

```
global FILE_PATH

(program)@prog {
  ; ...
  let dir = (path-dir FILE_PATH)
  let stem = (path-filestem FILE_PATH)
  let mod_name = (path-join dir stem)
  node mod_def
  attr mod_def type = "pop_symbol", symbol = mod_name, is_definition, source_node = @prog
  ; ...
}
```

The following example resolves an import relative to the current file:

```
global FILE_PATH

(import name:(_)@name)@import {
  ; ...
  let dir = (path-dir FILE_PATH)
  let mod_name = (path-normalize (path-join dir (Source-text @name)))
  node mod_def
  attr mod_def type = "pop_symbol", symbol = mod_name, is_definition, source_node = @prog
  ; ...
}
```

Using this crate from Rust

If you need very fine-grained control over how to use the resulting stack graphs, you can construct and operate on `StackGraph` instances directly from Rust code. You will need Rust bindings for the tree-sitter grammar for your source language — for instance, `tree-sitter-python`. Grammar Rust bindings provide a global symbol `language` that you will need. For this example we assume the source of the stack graph rules is defined in a constant `STACK_GRAPH_RULES`.

Once you have those, and the contents of the source file you want to analyze, you can construct a stack graph as follows:

```
let python_source = r#"
  import sys
  print(sys.path)
"#;
let grammar = tree_sitter_python::LANGUAGE.into();
let tsg_source = STACK_GRAPH_RULES;
let mut language = StackGraphLanguage::from_str(grammar, tsg_source)?;
let mut stack_graph = StackGraph::new();
let file_handle = stack_graph.get_or_create_file("test.py");
let globals = Variables::new();
language.build_stack_graph_into(&mut stack_graph, file_handle, python_source, &globals, &NoCancella
```

Modules

functions	Define tree-sitter-graph functions
loader	Defines file loader for stack graph languages
test	Defines a test file format for stack graph resolution.

Structs

AtomicCancellationFlag	
Builder	
CancelAfterDuration	
CancellationError	
NoCancellation	
OrCancellationFlag	
StackGraphLanguage	Holds information about how to construct stack graphs for a particular language.
Variables	Environment of immutable variables

Enums

BuildError	An error that can occur while loading a stack graph from a TSG file
LanguageError	An error that can occur while loading in the TSG stack graph construction rules for a language
VariableError	

Constants

FILE_PATH_VAR	Name of the variable used to pass the file path. If a root path is given, it should be a descendant of the root path.
JUMP_TO_SCOPE_NODE_VAR	Name of the variable used to pass the jump-to-scope node.
ROOT_NODE_VAR	Name of the variable used to pass the root node.
ROOT_PATH_VAR	Name of the variable used to pass the root path. If given, should be an ancestor of the file path.

Traits

CancellationFlag	Trait to signal that the execution is cancelled
FileAnalyzer	