import {FileImporter, Importer} from './importer';

import {Logger} from './logger';

import {Value} from './value';

import {PromiseOr} from './util/promise\_or';

/\*\*

\* Syntaxes supported by Sass:

\*

\* - `'scss'` is the [SCSS

\* syntax](https://sass-lang.com/documentation/syntax#scss).

\* - `'indented'` is the [indented

\* syntax](https://sass-lang.com/documentation/syntax#the-indented-syntax)

\* - `'css'` is plain CSS, which is parsed like SCSS but forbids the use of any

\* special Sass features.

\*

\* @category Options

\*/

export type Syntax = 'scss' | 'indented' | 'css';

/\*\*

\* Possible output styles for the compiled CSS:

\*

\* - `"expanded"` (the default for Dart Sass) writes each selector and

\* declaration on its own line.

\*

\* - `"compressed"` removes as many extra characters as possible, and writes

\* the entire stylesheet on a single line.

\*

\* @category Options

\*/

export type OutputStyle = 'expanded' | 'compressed';

/\*\*

\* A callback that implements a custom Sass function. This can be passed to

\* [[Options.functions]].

\*

\* ```js

\* const result = sass.compile('style.scss', {

\* functions: {

\* "sum($arg1, $arg2)": (args) => {

\* const arg1 = args[0].assertNumber('arg1');

\* const value1 = arg1.value;

\* const value2 = args[1].assertNumber('arg2')

\* .convertValueToMatch(arg1, 'arg2', 'arg1');

\* return new sass.SassNumber(value1 + value2).coerceToMatch(arg1);

\* }

\* }

\* });

\* ```

\*

\* @typeParam sync - A `CustomFunction<'sync'>` must return synchronously, but

\* in return it can be passed to [[compile]] and [[compileString]] in addition

\* to [[compileAsync]] and [[compileStringAsync]].

\*

\* A `CustomFunction<'async'>` may either return synchronously or

\* asynchronously, but it can only be used with [[compileAsync]] and

\* [[compileStringAsync]].

\*

\* @param args - An array of arguments passed by the function's caller. If the

\* function takes [arbitrary

\* arguments](https://sass-lang.com/documentation/at-rules/function#taking-arbitrary-arguments),

\* the last element will be a [[SassArgumentList]].

\*

\* @returns The function's result. This may be in the form of a `Promise`, but

\* if it is the function may only be passed to [[compileAsync]] and

\* [[compileStringAsync]], not [[compile]] or [[compileString]].

\*

\* @throws any - This function may throw an error, which the Sass compiler will

\* treat as the function call failing. If the exception object has a `message`

\* property, it will be used as the wrapped exception's message; otherwise, the

\* exception object's `toString()` will be used. This means it's safe for custom

\* functions to throw plain strings.

\*

\* @category Custom Function

\*/

export type CustomFunction<sync extends 'sync' | 'async'> = (

args: Value[]

) => PromiseOr<Value, sync>;

/\*\*

\* Options that can be passed to [[compile]], [[compileAsync]],

\* [[compileString]], or [[compileStringAsync]].

\*

\* @typeParam sync - This lets the TypeScript checker verify that asynchronous

\* [[Importer]]s, [[FileImporter]]s, and [[CustomFunction]]s aren't passed to

\* [[compile]] or [[compileString]].

\*

\* @category Options

\*/

export interface Options<sync extends 'sync' | 'async'> {

/\*\*

\* If this is `true`, the compiler will exclusively use ASCII characters in

\* its error and warning messages. Otherwise, it may use non-ASCII Unicode

\* characters as well.

\*

\* @defaultValue `false`

\* @category Messages

\*/

alertAscii?: boolean;

/\*\*

\* If this is `true`, the compiler will use ANSI color escape codes in its

\* error and warning messages. If it's `false`, it won't use these. If it's

\* undefined, the compiler will determine whether or not to use colors

\* depending on whether the user is using an interactive terminal.

\*

\* @category Messages

\*/

alertColor?: boolean;

/\*\*

\* If `true`, the compiler may prepend `@charset "UTF-8";` or U+FEFF

\* (byte-order marker) if it outputs non-ASCII CSS.

\*

\* If `false`, the compiler never emits these byte sequences. This is ideal

\* when concatenating or embedding in HTML `<style>` tags. (The output will

\* still be UTF-8.)

\*

\* @defaultValue `true`

\* @category Output

\* @compatibility dart: "1.54.0", node: false

\*/

charset?: boolean;

/\*\*

\* Additional built-in Sass functions that are available in all stylesheets.

\* This option takes an object whose keys are Sass function signatures like

\* you'd write for the [`@function

\* rule`](https://sass-lang.com/documentation/at-rules/function) and whose

\* values are [[CustomFunction]]s.

\*

\* Functions are passed JavaScript representations of [Sass value

\* types](https://sass-lang.com/documentation/js-api#value-types), and must

\* return the same.

\*

\* When writing custom functions, it's important to make them as user-friendly

\* and as close to the standards set by Sass's core functions as possible. Some

\* good guidelines to follow include:

\*

\* \* Use `Value.assert\*` methods, like [[Value.assertString]], to cast untyped

\* `Value` objects to more specific types. For values that were passed

\* directly as arguments, pass in the argument name as well. This ensures

\* that the user gets good error messages when they pass in the wrong type

\* to your function.

\*

\* \* Individual classes may have more specific `assert\*` methods, like

\* [[SassNumber.assertInt]], which should be used when possible.

\*

\* \* In Sass, every value counts as a list. Rather than trying to detect the

\* [[SassList]] type, you should use [[Value.asList]] to treat all values as

\* lists.

\*

\* \* When manipulating values like lists, strings, and numbers that have

\* metadata (comma versus space separated, bracketed versus unbracketed,

\* quoted versus unquoted, units), the output metadata should match the

\* input metadata.

\*

\* \* When in doubt, lists should default to comma-separated, strings should

\* default to quoted, and numbers should default to unitless.

\*

\* \* In Sass, lists and strings use one-based indexing and use negative

\* indices to index from the end of value. Functions should follow these

\* conventions. [[Value.sassIndexToListIndex]] and

\* [[SassString.sassIndexToStringIndex]] can be used to do this

\* automatically.

\*

\* \* String indexes in Sass refer to Unicode code points while JavaScript

\* string indices refer to UTF-16 code units. For example, the character

\* U+1F60A SMILING FACE WITH SMILING EYES is a single Unicode code point but

\* is represented in UTF-16 as two code units (`0xD83D` and `0xDE0A`). So in

\* JavaScript, `"a😊b".charCodeAt(1)` returns `0xD83D`, whereas in Sass

\* `str-slice("a😊b", 1, 1)` returns `"😊"`. Functions should follow Sass's

\* convention. [[SassString.sassIndexToStringIndex]] can be used to do this

\* automatically, and the [[SassString.sassLength]] getter can be used to

\* access a string's length in code points.

\*

\* @example

\*

\* ```js

\* sass.compileString(`

\* h1 {

\* font-size: pow(2, 5) \* 1px;

\* }`, {

\* functions: {

\* // Note: in real code, you should use `math.pow()` from the built-in

\* // `sass:math` module.

\* 'pow($base, $exponent)': function(args) {

\* const base = args[0].assertNumber('base').assertNoUnits('base');

\* const exponent =

\* args[1].assertNumber('exponent').assertNoUnits('exponent');

\*

\* return new sass.SassNumber(Math.pow(base.value, exponent.value));

\* }

\* }

\* });

\* ```

\*

\* @category Plugins

\*/

functions?: Record<string, CustomFunction<sync>>;

/\*\*

\* Custom importers that control how Sass resolves loads from rules like

\* [`@use`](https://sass-lang.com/documentation/at-rules/use) and

\* [`@import`](https://sass-lang.com/documentation/at-rules/import).

\*

\* Loads are resolved by trying, in order:

\*

\* - The importer that was used to load the current stylesheet, with the

\* loaded URL resolved relative to the current stylesheet's canonical URL.

\*

\* - Each [[Importer]] or [[FileImporter]] in [[importers]], in order.

\*

\* - Each load path in [[loadPaths]], in order.

\*

\* If none of these return a Sass file, the load fails and Sass throws an

\* error.

\*

\* @category Plugins

\*/

importers?: (Importer<sync> | FileImporter<sync>)[];

/\*\*

\* Paths in which to look for stylesheets loaded by rules like

\* [`@use`](https://sass-lang.com/documentation/at-rules/use) and

\* [`@import`](https://sass-lang.com/documentation/at-rules/import).

\*

\* A load path `loadPath` is equivalent to the following [[FileImporter]]:

\*

\* ```js

\* {

\* findFileUrl(url) {

\* // Load paths only support relative URLs.

\* if (/^[a-z]+:/i.test(url)) return null;

\* return new URL(url, pathToFileURL(loadPath));

\* }

\* }

\* ```

\*

\* @category Input

\*/

loadPaths?: string[];

/\*\*

\* An object to use to handle warnings and/or debug messages from Sass.

\*

\* By default, Sass emits warnings and debug messages to standard error, but

\* if [[Logger.warn]] or [[Logger.debug]] is set, this will invoke them

\* instead.

\*

\* The special value [[Logger.silent]] can be used to easily silence all

\* messages.

\*

\* @category Messages

\*/

logger?: Logger;

/\*\*

\* If this option is set to `true`, Sass won’t print warnings that are caused

\* by dependencies. A “dependency” is defined as any file that’s loaded

\* through [[loadPaths]] or [[importer]]. Stylesheets that are imported

\* relative to the entrypoint are not considered dependencies.

\*

\* This is useful for silencing deprecation warnings that you can’t fix on

\* your own. However, please <em>also</em> notify your dependencies of the deprecations

\* so that they can get fixed as soon as possible!

\*

\* \*\*Heads up!\*\* If [[compileString]] or [[compileStringAsync]] is called

\* without [[StringWithoutImporter.url]], <em>all</em> stylesheets it loads

\* will be considered dependencies. Since it doesn’t have a path of its own,

\* everything it loads is coming from a load path rather than a relative

\* import.

\*

\* @defaultValue `false`

\* @category Messages

\*/

quietDeps?: boolean;

/\*\*

\* Whether or not Sass should generate a source map. If it does, the source

\* map will be available as [[CompileResult.sourceMap]].

\*

\* \*\*Heads up!\*\* Sass doesn't automatically add a `sourceMappingURL` comment

\* to the generated CSS. It's up to callers to do that, since callers have

\* full knowledge of where the CSS and the source map will exist in relation

\* to one another and how they'll be served to the browser.

\*

\* @defaultValue `false`

\* @category Output

\*/

sourceMap?: boolean;

/\*\*

\* Whether Sass should include the sources in the generated source map.

\*

\* This option has no effect if [[sourceMap]] is `false`.

\*

\* @defaultValue `false`

\* @category Output

\*/

sourceMapIncludeSources?: boolean;

/\*\*

\* The [[OutputStyle]] of the compiled CSS.

\*

\* @example

\*

\* ```js

\* const source = `

\* h1 {

\* font-size: 40px;

\* code {

\* font-face: Roboto Mono;

\* }

\* }`;

\*

\* let result = sass.compileString(source, {style: "expanded"});

\* console.log(result.css.toString());

\* // h1 {

\* // font-size: 40px;

\* // }

\* // h1 code {

\* // font-face: Roboto Mono;

\* // }

\*

\* result = sass.compileString(source, {style: "compressed"})

\* console.log(result.css.toString());

\* // h1{font-size:40px}h1 code{font-face:Roboto Mono}

\* ```

\*

\* @category Output

\*/

style?: OutputStyle;

/\*\*

\* By default, Dart Sass will print only five instances of the same

\* deprecation warning per compilation to avoid deluging users in console

\* noise. If you set `verbose` to `true`, it will instead print every

\* deprecation warning it encounters.

\*

\* @defaultValue `false`

\* @category Messages

\*/

verbose?: boolean;

}

/\*\*

\* Options that can be passed to [[compileString]] or [[compileStringAsync]].

\*

\* If the [[StringOptionsWithImporter.importer]] field isn't passed, the

\* entrypoint file can load files relative to itself if a `file://` URL is

\* passed to the [[url]] field.

\*

\* @typeParam sync - This lets the TypeScript checker verify that asynchronous

\* [[Importer]]s, [[FileImporter]]s, and [[CustomFunction]]s aren't passed to

\* [[compile]] or [[compileString]].

\*

\* @category Options

\*/

export interface StringOptionsWithoutImporter<sync extends 'sync' | 'async'>

extends Options<sync> {

/\*\*

\* The [[Syntax]] to use to parse the entrypoint stylesheet.

\*

\* @default `'scss'`

\*

\* @category Input

\*/

syntax?: Syntax;

/\*\*

\* The canonical URL of the entrypoint stylesheet.

\*

\* A relative load's URL is first resolved relative to [[url]], then resolved

\* to a file on disk if it's a `file://` URL. If it can't be resolved to a

\* file on disk, it's then passed to [[importers]] and [[loadPaths]].

\*

\* @category Input

\*/

url?: URL;

}

/\*\*

\* Options that can be passed to [[compileString]] or [[compileStringAsync]].

\*

\* If the [[StringOptionsWithImporter.importer]] field is passed, the entrypoint

\* file uses it to load files relative to itself and the [[url]] field is

\* mandatory.

\*

\* @typeParam sync - This lets the TypeScript checker verify that asynchronous

\* [[Importer]]s, [[FileImporter]]s, and [[CustomFunction]]s aren't passed to

\* [[compile]] or [[compileString]].

\*

\* @category Options

\*/

export interface StringOptionsWithImporter<sync extends 'sync' | 'async'>

extends StringOptionsWithoutImporter<sync> {

/\*\*

\* The importer to use to handle loads that are relative to the entrypoint

\* stylesheet.

\*

\* A relative load's URL is first resolved relative to [[url]], then passed to

\* [[importer]]. If the importer doesn't recognize it, it's then passed to

\* [[importers]] and [[loadPaths]].

\*

\* @category Input

\*/

importer: Importer<sync> | FileImporter<sync>;

/\*\*

\* The canonical URL of the entrypoint stylesheet. If this is passed along

\* with [[importer]], it's used to resolve relative loads in the entrypoint

\* stylesheet.

\*

\* @category Input

\*/

url: URL;

}

/\*\*

\* Options that can be passed to [[compileString]] or [[compileStringAsync]].

\*

\* This is a [[StringOptionsWithImporter]] if it has a

\* [[StringOptionsWithImporter.importer]] field, and a

\* [[StringOptionsWithoutImporter]] otherwise.

\*

\* @typeParam sync - This lets the TypeScript checker verify that asynchronous

\* [[Importer]]s, [[FileImporter]]s, and [[CustomFunction]]s aren't passed to

\* [[compile]] or [[compileString]].

\*

\* @category Options

\*/

export type StringOptions<sync extends 'sync' | 'async'> =

| StringOptionsWithImporter<sync>

| StringOptionsWithoutImporter<sync>;