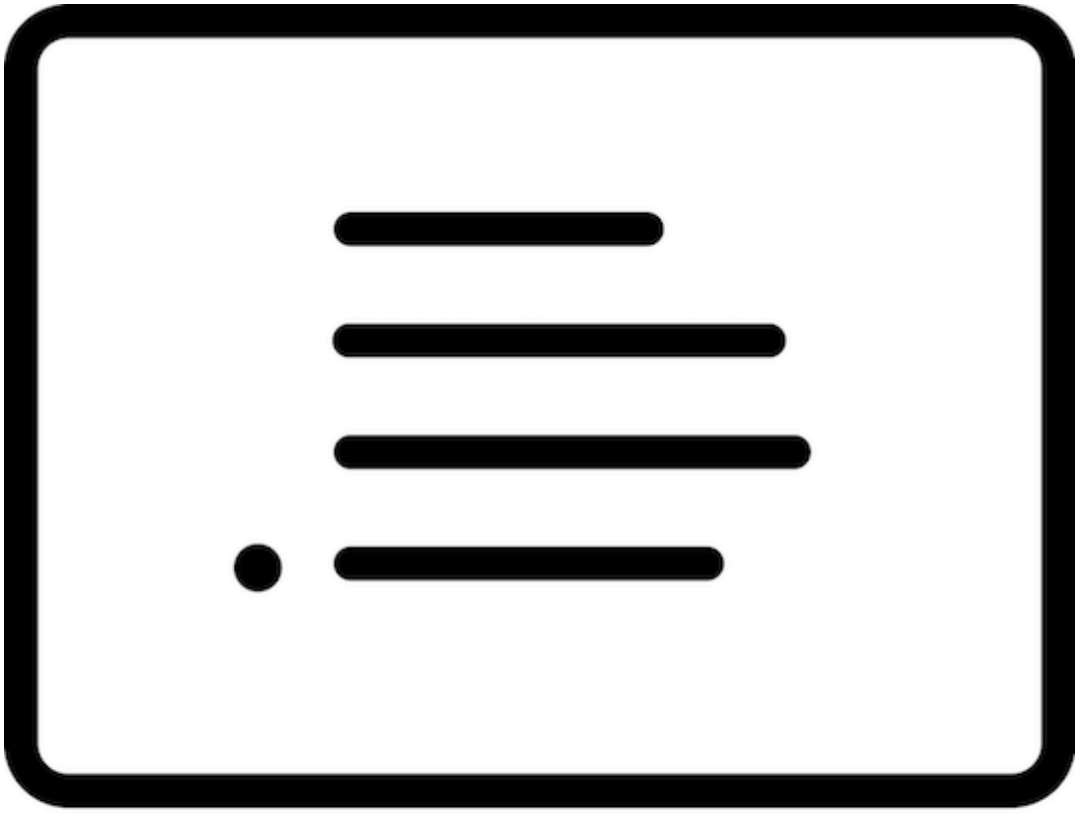
N4JS Design Specification

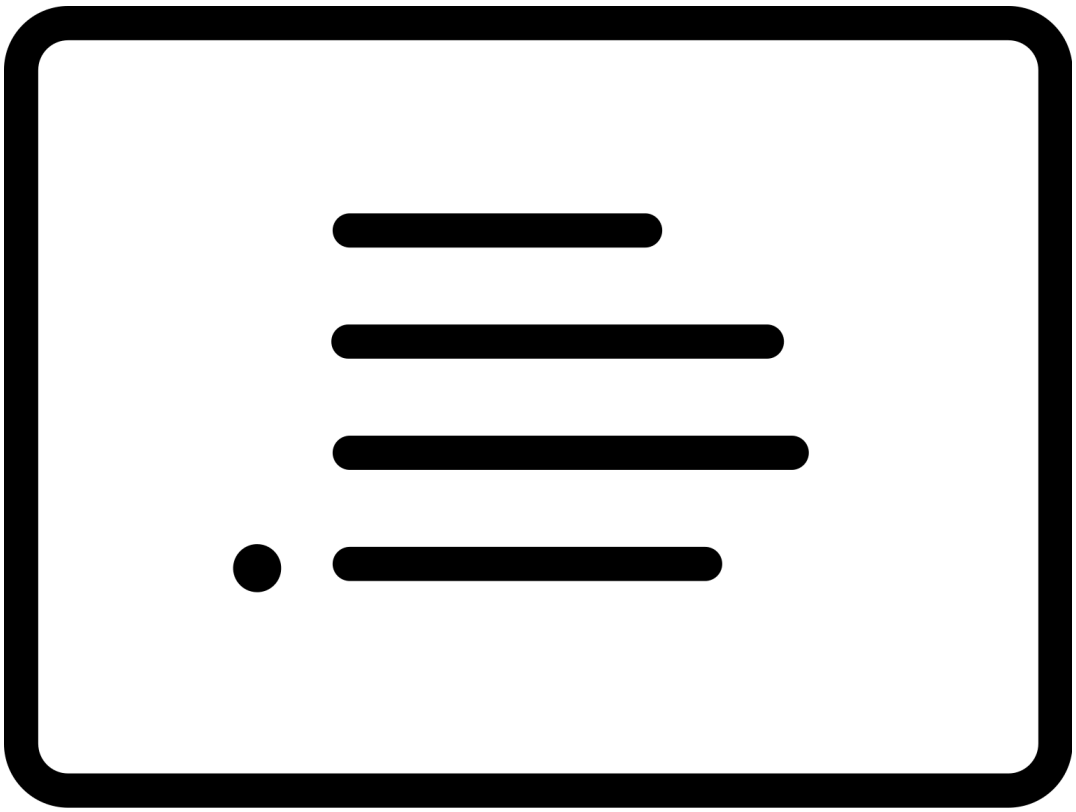


N4JS

2019-01-11 12:16:49 CET

Table of Contents

List of Figures



N4JS

Last Updated: 2019-01-11

Authors:
Jens von Pilgrim, Jakub Siberski, Mark-Oliver Reiser,
Torsten Krämer, Ákos Kitta, Sebastian Zarnekow, Lorenzo Bettini, Jörg Reichert, Kristian Duske, Marcus Mews, Minh Quang Tran, Luca Beurer-Kellner

This document contains the N4JS Design and Implementation documentation.

Chapter 1. Introduction

This document describes design aspects of the N4JS compiler and IDE. It relies on the following N4JS related specifications:

- N4JS Language Specification [N4JSSpec]

1.1. Notation

We reuse the notation specified in [N4JSSpec].

1.2. IDE Components

The N4JS and N4JSIDE components are organized via features. The following features with included plugins are defined (the common prefix "org.eclipse.n4js" is omitted at the plugin name):

Feature	Plugin	Description
org.eclipse.n4js.lang.sdk		N4JS core language with parser, validation etc.
	org.eclipse.n4js	Xtext grammar with generator and custom code for N4JS, scoping (and binding) implementation, basic validation (and Xsemantics type system).
	doc	(in doc folder) General documentation (including web page) written in AsciiDoc
	external.libraries	Support for N4JS libraries shipped with the IDE, i.e. core N4JS library and mangelhaft.
	ui	UI components for N4JS, e.g., proposal provider, labels, outline, quickfixes.
	jsdoc	Parser and model for JSDoc
	external.libraries.updatesite	**Not included in feature**. Updates the external library plugin
org.eclipse.n4js.ts.sdk		Type System
	ts	Xtext grammar with generator and custom code for type expressions and standalone type definitions.
	ts.model	Xcore based types model with helper classes etc.
	ts.ui	Xtext generated UI for type system, not really used as this TS files are not editable by users.
org.eclipse.n4js.unicode.sdk		
	common.unicode	Xtext grammar with generator and custom code used by all other grammars for proper unicode support.
org.eclipse.n4js.regex.sdk		Regular expression grammar and UI, used by N4JS grammar and UI
	regex	Xtext grammar with generator and custom code used by N4JS grammars for regular expressions.
	regex.ui	UI components for regular expressions, e.g., proposal provider, labels, outline, quickfixes.
org.eclipse.n4js.sdk		This feature defines the N4JSIDE. It contains core UI plugins and all includes (almost all) other features!
	environments	Utility plugin, registers n4scheme for EMF proxy resolution.
	model	Xcore based N4JS model with helper classes etc.
	product	N4JSIDE main application.
	releng.utils	(in releng folder) Contains utility classes only used for building the system, e.g., tools for generating antlr based parser with extended features.
	utils	general utilities
	utils.ui	general UI utilities
org.eclipse.n4js.compiler.sdk		Generators and Transpilers
	generator.common	**Not included in feature, logically associated.**
	generator.headless	N4JS headless generator (i.e. command line compiler).
	transpiler	Generic transpiler infrastructure
	transpiler.es	Transpiler to compile to EcmaScript
org.eclipse.n4js.json.sdk		N4JS JSON
	json	Xtext grammar with generator and custom code for a extensible JSON language support. Used in N4JS for the project description in terms of a `package.json` file.

Feature	Plugin	Description
	json.ui	UI components for extensible JSON language support, e.g., proposal provider, labels, outline.
	json.model	**Not included in feature, logically associated.** Xcore based model for the JSON language.
org.eclipse.n4js.semver.sdk		Semantic version string support.
	semver	Parser and tools for semantic version strings.
	semver.ui	UI tools for semantic version strings.
	semver.model	**Not included in feature, logically associated.** Xcore model of semantic version strings.
org.eclipse.n4js.runner.sdk		Runners for executing N4JS or JavaScript code
	runner	Generic interfaces and helper for runners, i.e. JavaScript engines executing N4JS or JavaScript code.
	runner.chrome	Runner for executing N4JS or JavaScript with Chrome.
	runner.chrome.ui	UI classes for launching the Chrome runner via the org.eclipse.debug.ui
	runner.nodejs	Runner for executing N4JS or JavaScript with node.js.
	runner.nodejs.ui	UI classes for launching the node.js runner via the org.eclipse.debug.ui
	runner.ui	Generic interfaces for configuring N4JS runner via the debug ui.
org.eclipse.n4js.tester.sdk		Runners and UI for tests (via mangelhaft).
	tester	Generic interfaces and helper for testers, i.e. JavaScript engines executing N4JS tests (using mangelhaft).
	tester.nodejs	Tester based on the nodejs runner for executing mangelhaft tests with node.js
	tester.nodejs.ui	UI for showing test results.
	tester.ui	Configuration of tests via the debug UI.
org.eclipse.n4js.jsdoc2spec.sdk		JSDoc2Spec
	jsdoc2spec	Exporter to generate API documentation with specification tests awareness
	jsdoc2spec.ui	UI for API doc exporter
org.eclipse.n4js.xpect.sdk		
	xpect	Xpect test methods.
	xpect.ui	UI for running Xpext tests methods from the N4JSIDE (for creating bug reports).
org.eclipse.n4js.smith.sdk		Feature for internal N4JS IDE plugins only intended for development (for example, the AST Graph view).
	smith	Non-UI classes for tools for smiths, that is, tools for developers of the N4JS IDE such as AST views etc.
	smith.ui	UI classes for tools for smiths, that is, tools for developers of the N4JS IDE such as AST views etc.
org.eclipse.n4js.tests.helper.sdk		Helper for tests.
org.eclipse.n4js.dependencies.sdk		Collection of all external non-ui dependencies, used for local mirroring of update sites.
org.eclipse.n4js.dependencies.ui.sdk		Collection of all external ui dependencies, used for local mirroring of update sites.
uncategorized plugins		
	flowgraphs	Control and data flow graph model and computer.
Fragments		not associated to features, only listed here for completeness
	utils.logging	Fragment only, configuration for loggers, in particular for the product and for the tests

1.2.1. Naming Conventions

In the above sections, tests were omitted. We use the following naming conventions (by example) for test and tests helper:

project	-
project.tests	tests for project, is a fragment
project.tests.helper	helper classes used ONLY by tests
project.tests.performance	performance tests
project.tests.integration	integration tests
project.ui	-

project.ui.tests	tests for ui project, fragment of project.ui
project.ui.tests.helper	helper classes used ONLY by tests
project.ui.tests.performance	
tests.helper	general test helper
ui.tests.helper	general ui test helper
project.xpect.tests	xpect tests for the project, despite dependnecies to UI the can be executed as plain JUnit tests
project.xpect.ui.tests	xpect tests for the project, need to be executed as eclipse plugin tests

Due to Maven, tests are in subfolder tests (incl. helpers), implementation bundles in plugins, and release engineering related bundles in releng.

Chapter 2. Eclipse Setup

2.1. Contribute

Eclipse developers who want to develop N4JS itself should use the Oomph Eclipse installer. The N4JS project is listed under "Eclipse Projects/N4JS" This setup installs the correct Eclipse version, creates a new workspace and clones all projects into it (for details see below).

2.1.1. Eclipse Installer

The recommended way to install the Eclipse IDE and set up the workspace is to use the Eclipse Installer. This installer is to be downloaded from https://wiki.eclipse.org/Eclipse_Installer

Run the installer and apply the following steps:

1. change to "Advance Mode" via the menu (upper-right corner) (no need to move the installer)
2. select a product, e.g. "Eclipse IDE for Eclipse Committers" with product version "Oxygen"
3. double-click the entry **Eclipse Projects/N4JS** so that it is shown in the catalog view below
4. on the next page, configure paths accordingly. You only have to configure the installation and workspace folder.
5. start installation

The installer will then guide you through the rest of the installation. All plug-ins are downloaded and configured automatically, so is the workspace including downloading the git repository and setting up the workspace.

2.1.2. Manual IDE Configuration

For a manual install, clone the code and import all top-level projects from the docs, features, plugins, releng, testhelpers, and tests folders. Activate the targetplatform contained in the `releng/org.eclipse.n4js.targetplatform/` project.

The N4JS IDE is developed with Eclipse Oxygen 4.7 or better since the system is based on Eclipse anyway. It is almost impossible to use another IDE to develop Eclipse plugins. The list of required plugins includes:

- Xtext/Xtend 2.10.0
- Xcore 1.4.0
- Xsemantics 1.10.0
- Xpect 0.1

It is important to use the latest version of Xtext and the corresponding service release of Xcore. You will find the latest version numbers and plugins used in the target platform definition at https://github.com/eclipse/n4js/blob/master/releng/org.eclipse.n4js.targetplatform/org.eclipse.n4js.targetplatform.target

Chapter 3. Release Engineering

3.1. Nightly build on Eclipse infrastructure

The N4JS IDE, headless n4jsc.jar, and the N4JS update site is being built on the Eclipse Common Build Infrastructure (CBI). For this purpose the N4JS project is using a dedicated Jenkins instance, referred to as a "Jenkins Instance Per Project" (JIPP) in Eclipse CBI documentation. At this time, the N4JS project's JIPP is running on the "old" infrastructure, not yet using docker. This will be migrated at a later point in time.

The N4JS JIPP is available at: https://ci.eclipse.org/n4js/

The nightly build performs the following main steps:

1. compile the N4JS implementation,
2. build the n4jsc.jar, the IDE products for MacOS, Windows, Linux, and the update site,
3. run tests,
4. sign the IDE product for macOS and package it in a .dmg file,
5. deploy to n4jsc.jar, IDE products and update sites to Eclipse download server (i.e. download.eclipse.org),
6. move all artifacts older than 7 days from download.eclipse.org to archive.eclipse.org.

Details about all the above steps can be found in the Jenkinsfile `eclipse-nightly.jenkinsfile`, located in the root folder of the N4JS source repository on GitHub.

The most accurate documentation for our JIPP can be found at https://wiki.eclipse.org/IT_Infrastructure_Doc. Note that many other documents do not apply to our JIPP, at the moment, as they refer to the new infrastructure, e.g. https://wiki.eclipse.org/CBI and https://wiki.eclipse.org/Jenkins.

3.2. Build the N4JS IDE from command line

Ensure you have

- Java 8
- Maven 3.2.x and
- Node.js 6

installed on your system.

Clone the repository

```
git clone https://github.com/Eclipse/n4js.git
```

Change to the n4js folder:

```
cd n4js
```

Run the Maven build:

```
mvn clean verify
```

You may have to increase the memory for maven via `export MAVEN_OPTS="-Xmx2048m"` (Unix) or `set MAVEN_OPTS="-Xmx2048m"` (Windows).

3.2.1. Publish maven-tooling `org.eclipse.n4js.releng.util`

For extending the N4JS-language in a different project, the `org.eclipse.n4js.releng.util` module needs to be published as a maven-plugin. You can deploy this SNAPSHOT-artifact to a local folder by providing the `local-snapshot-deploy-folder` -property pointing to an absolute path in the local file system:

```
mvn clean deploy -Dlocal-snapshot-deploy-folder=/var/lib/my/folder/local-mvn-deploy-repository
```

The existence of `local-snapshot-deploy-folder` will trigger a profile enabling the deploy-goal for the project `org.eclipse.n4js.releng.util`

3.2.2. Generation of Eclipse help for spec and design document

The HTML pages for N4JSSpec and N4JSDesign documents are generated from the Asciidoc sources in the project `org.eclipse.n4js.spec` `org.eclipse.n4js.design` by Asciispec.

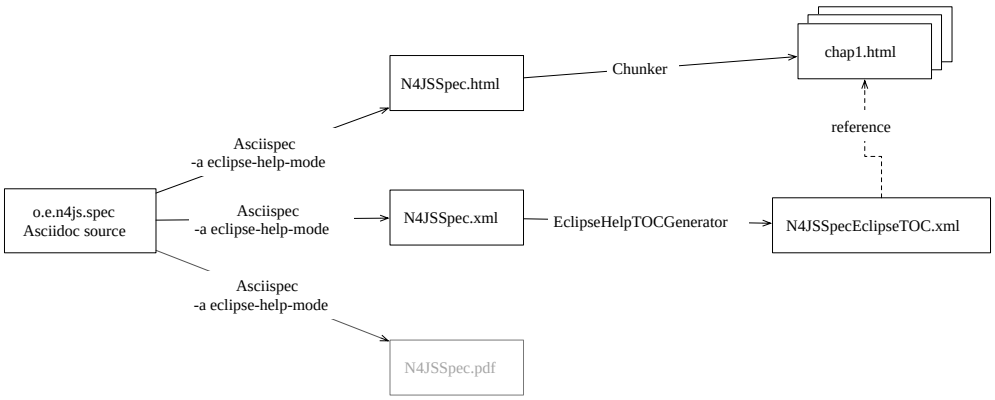


Figure 3.1. The process of creating Eclipse help for N4JSSpec

Figure Figure 3.1, “The process of creating Eclipse help for N4JSSpec” shows the generation process for N4JSSpec document. The process for N4JSDesign is the same. The following explains the diagram.

- `Asciispec` is used to compile the source N4JSSpec Asciidoc into a single large `N4JSSpec.html` file which contains all the chapters. The use of the custom parameter `-a eclipse-help-mode` indicates that a specical header and footer styles as well as CSS style should be used (i.e. no table of content menu, no download links etc.). Here, we are using the possibility provided by Asciidoctor to configure header/footer as well as CSS style via parameter `:docinfodir:` and `:stylesheet:`.
- Our custom tool `Chunker` splits `N4JSSpec.html` into multiple chunked HTML files, each of which corresponds to either the `index` file or a chapter.
- Another custom tool `EclipseHelpTOCGenerator` takes to Docbook file `N4JSSpec.xml` and generates an XML file describing the table of content (TOC) in the Eclipse format. This TOC file references the chunked HTML files above.

Chapter 4. Parser

Some of the concepts described here were presented at EclipseCon 2013 and XtextCon 2014. Note that the material presented at the linked videos may be outdated.

4.1. Overview

The parser is created from an Xtext grammar. Actually, there are several grammars used as shown in Figure CD Grammars. These grammars and the parsers generated from them are described more closely in the following sections.

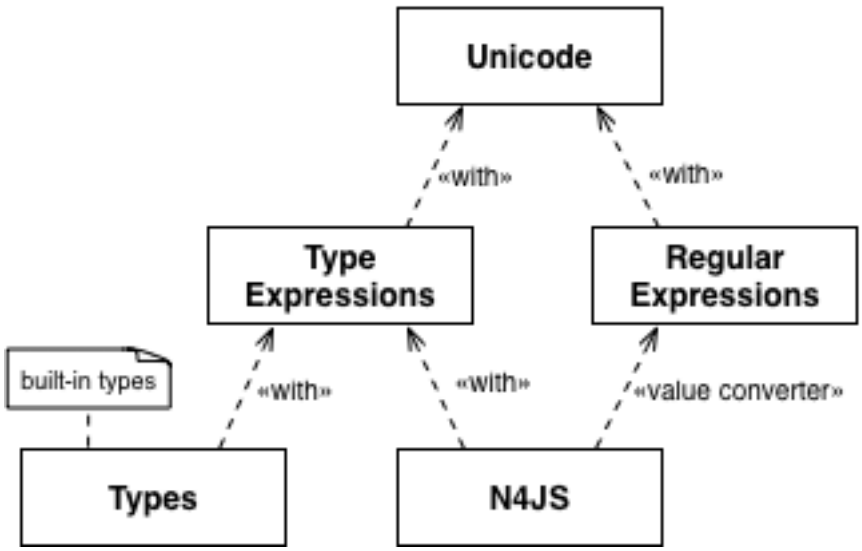


Figure 4.1. N4 Grammars

4.2. N4JS Parser

One of the most tricky parts of JavaScript is the parsing because there is a conceptual mismatch between the ANTLR runtime and the specified grammar. Another challenge is the disambiguation of regular expressions and binary operations. Both features require significant customizing of the generated parser (see figure below).

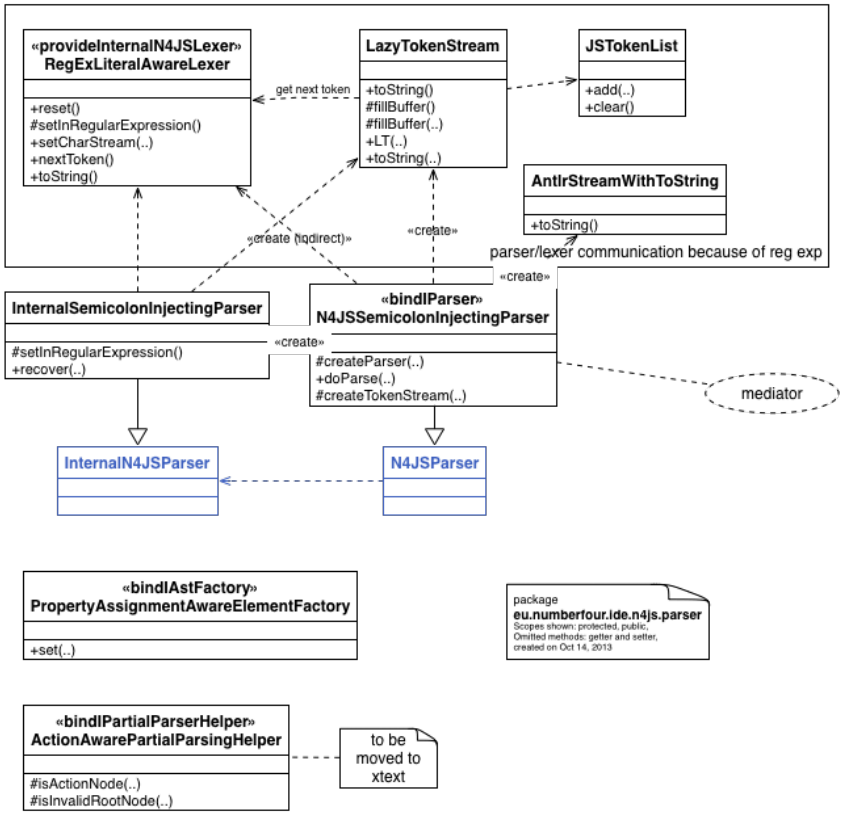


Figure 4.2. Overview custom parser implementation (runtime only)

4.3. Parser Generation Post-Processing

The ANTLR grammar that is generated by Xtext is post-processed to inject custom code into the grammar file before it is passed to the ANTLR tool. This is required in particular due to ASI (Automated Semicolon Insertion), but for some other reasons as well.

Actually, there are several injections:

1. Due to Xtext restrictions, the generated ANTLR grammar file (***.g**) is modified. This means that some some additional actions are added and some rules are rewritten.
2. Due to ANTLR restrictions, the generated ANTLR Java parser (***.java**) os modified. This means that some generated rules are slightly modified to match certain requirements.
3. Due to Java restrictions, the generated Java parser needs to be preprocessed in order to reduce the size of certain methods since they must not exceed 64k characters. This is implemented by means of an MWE fragment, activated after the other post processing steps are done.

The first two steps are handled by `AntlrGeneratorWithCustomKeywordLogic`, which is configured with additional helpers in `GenerateN4JS.mwe2`. shows the customized classes which modify the code generation. These classes are all part of the `releng.utils` bundle.

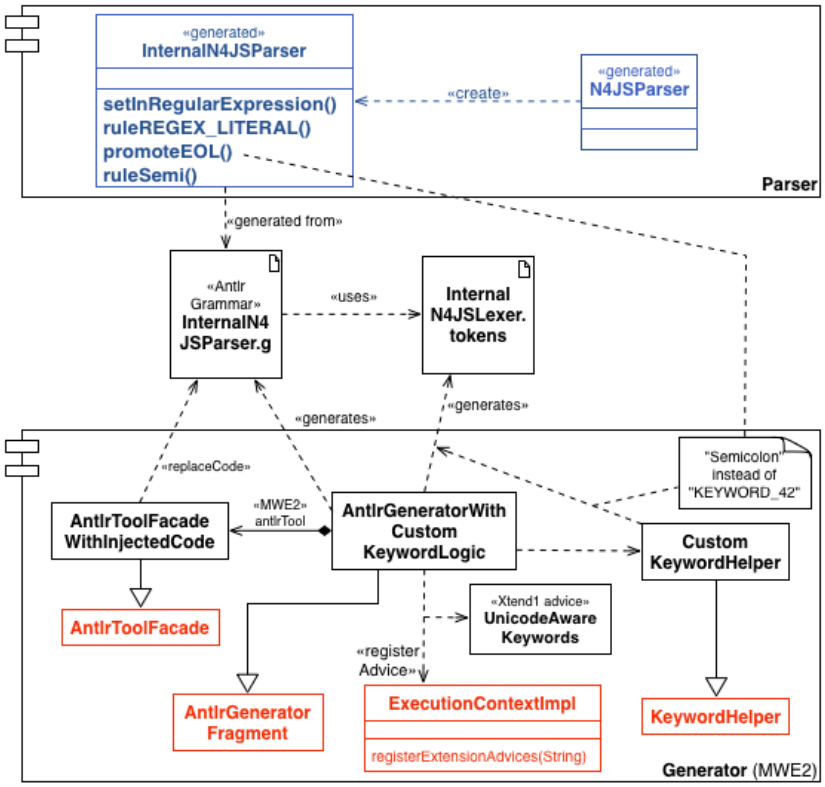


Figure 4.3. Class Diagram Parser Generation

4.3.1. Automatic Semicolon Insertion

The EcmaScript specification mandates that valid implementations automatically insert a semicolon as a statement delimiter if it is missing and the input file would become invalid due to the missing semicolon. This is known as ASI. It implies that not only valid implementations have to perform this, but a valid parser has to mimic this behavior in order to parse executable code. The ASI is implemented by two different means.

The parser's error recovery strategy is customized so it attempts to insert a semicolon if it was expected. Both strategies have to work hand in hand in order to consume all sorts of legal JavaScript code.

4.3.1.1. Injected code in the Antlr grammar file

Under certain circumstances, the parser has to actively promote a token to become a semicolon even though it may be a syntactically a closing brace or line break. This has to happen before that token is consumed thus the rules for return statements, continue statements and break statements are enhanced to actively promote these tokens to semicolons.

The same rule is applied to promote line breaks between an expression and a possible postfix operator `++` or `--`. At this location the line break is always treated as a semicolon even though the operator may be validly consumed and produce a postfix expression.

In both cases, the method `promoteEOL()` is used to move a token that may serve as an automatically injected semicolon from the so called hidden token channel to the semantic channel. The hidden tokens are usually not handled by the parser explicitly thus they are semantically invisible (therefore the term hidden token). Nevertheless, they can be put on the semantic channel explicitly to make them recognizable. That's implemented in the EOL promotion. The offending tokens include the hidden line terminators and multi-line comments that include line breaks. Furthermore, closing braces (right curly brackets) are included in the set of offending tokens as well as explicit semicolons.

4.3.1.2. Customized error recovery

Since the EOL promotion does not work well with Antlr prediction mode, another customization complements that feature. As soon as an invalid token sequence is attempted to be parsed and missing semicolon would make that sequence valid, an offending token is sought and moved to the semantic channel. This is implemented in the custom recovery strategy.

4.3.2. Async and `No line terminator allowed here` Handling

There is no way of directly defining `No line terminator allowed here`. This is required not only for ASI, but also for `async`. This requires not only a special rule (using some rules from ASI), but also a special error recovery since the token 'async' may be rejected (by the manually enriched rule) which is of course unexpected behavior from the generated source code.

4.3.3. Regular Expression

The ANTLR parsing process can basically be divided into three steps. First of all, the file contents has to be read from disk. This includes the proper encoding of bytes to characters. The second step is the lexing or tokenizing of the character stream. A token is a basically a typed region in the stream, that is a triplet of token-id, offset and length. The last step is the parsing of these tokens. The result is a semantic model that is associated with a node tree. All necessary information to validate the model can be deduced from these two interlinked representations.

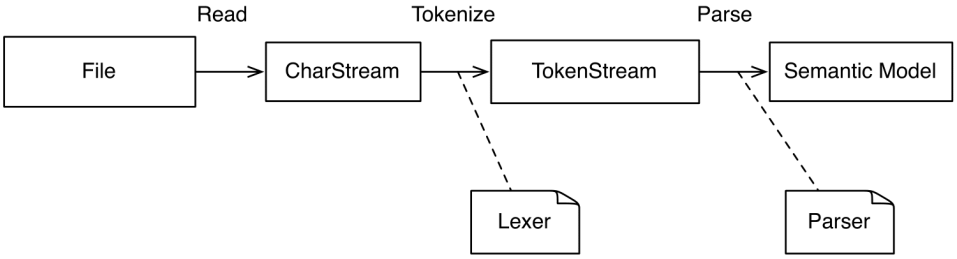


Figure 4.4. Simplified visualization of the parsing

Since the default semantics and control flow of Antlr generated parsers do not really fit the requirements of a fully working JavaScript parser, some customizations are necessary. **Regular expression literals in JavaScript cannot be syntactically disambiguated from div operations without contextual information.** Nevertheless, the spec clearly describes, where a regular expression may appear and where it is prohibited. Unfortunately, it is not possible to implement these rules in the lexer alone, since it does not have enough contextual information. Therefore, the parser has been enhanced to establish a communication channel with the lexer. It announces when it expects a regular expression rather than a binary operation.

This required a reworking of the Antlr internals. Instead of a completely pre-populated `TokenStream`, the parser works on a lazy implementation that only reads as many characters as possible without a disambiguation between regular expression literals and divide operators.

Only after the parser has read this buffered tokens and potentially announced that it expects a regular expression, another batch of characters is processed by the lexer until the next ambiguous situation occurs. This is fundamentally different from the default behavior of Antlr.

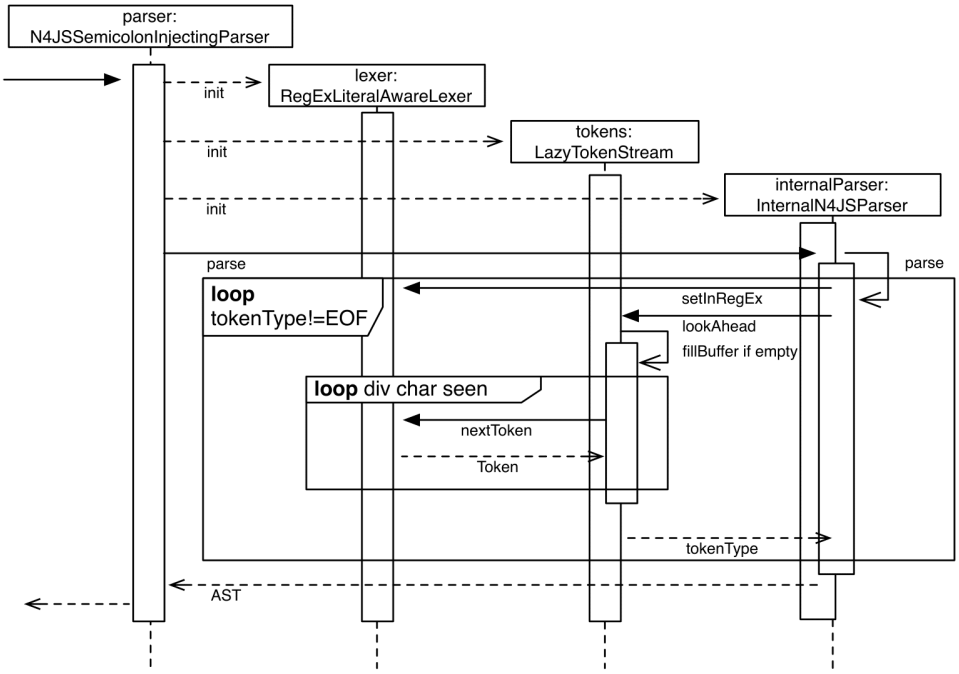


Figure 4.5. Abstract control flow during parsing

shows the involved classes which allow for this lexer-parser communication.

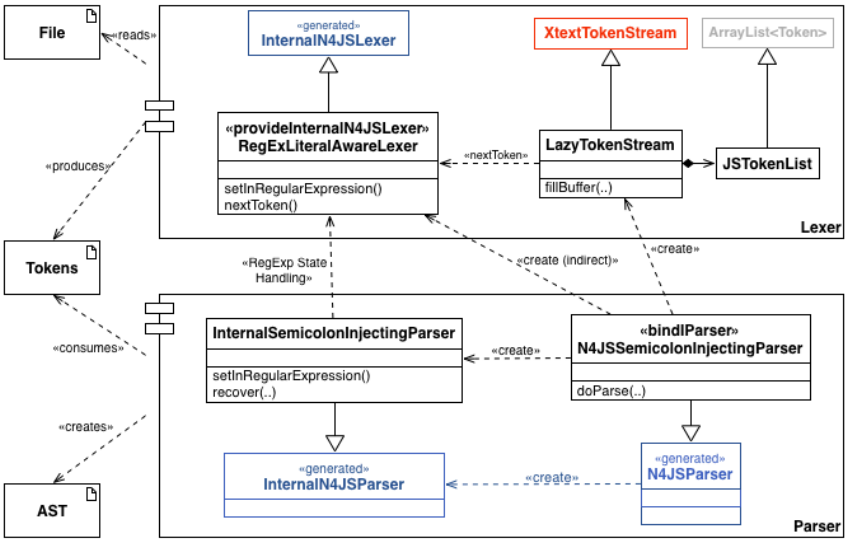


Figure 4.6. Class Diagram Parser-Lexer Communication

4.3.4. Unicode

Unicode support in JavaScript includes the possibility to use unicode escape sequences in identifiers, string literals and regular expression literals. Another issue in this field is the specification of valid identifiers in JavaScript. They are described by means of unicode character classes. These have to be enumerated in the terminal rules in order to fully accept or reject valid or invalid JS identifiers.

For that purpose, a small code generator is used to define the terminal fragments for certain unicode categories. The `UnicodeGrammarGenerator` basically iterates all characters from `Character.MIN_VALUE` to `Character.MAX_VALUE` and adds them as alternatives to the respective terminal fragments, e.g. `UNICODE_DIGIT_FRAGMENT`.

The real terminal rules are defined as a composition of these generated fragments. Besides that, each character in an identifier, in a string literal or in a regular expression literal may be represented by its unicode escape value, e.g. ` u0060`. These escape sequences are handled and validated by the `IValueConverter` for the corresponding terminal rules.

The second piece of the puzzle are the unicode escaped sequences that may be used in keywords. This issue is covered by the `UnicodeKeywordHelper` which replaces the default terminal representation in the generated Antlr grammar by more elaborated alternatives. The keyword `if` is not only lexed as `'if'` but as seen in snippet Terminal if listing.

Terminal if

```
If :
    ( 'i' | '\\' 'u' '0``   0``   6``   9' )
    ( 'f' | '\\' 'u' '0``   0``   6``   6' );
```

4.3.5. Literals

Template literals are also to be handled specially, see `TemplateLiteralDisambiguationInjector` for details.

4.4. Modifiers

On the AST side, all modifiers are included in a single enumeration `N4Modifier`. In the types model however, the individual modifiers are mapped to two different enumerations of *access* modifiers (namely `TypeAccessModifier` and `MemberAccessModifier`) and a number of boolean properties (in case of non-access modifiers such as `abstract` or `static`). This mapping is done by the types builder, mostly by calling methods in class `ModifierUtils`.

The grammar allows the use of certain modifiers in many places that are actually invalid. Rules where a certain modifier may appear in the AST are implemented in method isValid(EClass,N4Modifier) in class `ModifierUtils` and checked via several validations in `N4JSSyntaxValidator`. Those validations also check for a particular order of modifiers that is not enforced by the grammar.

See API documentation of enumeration `N4Modifier` in file `N4JS.xcore` and the utility class `ModifierUtils` for more details.

4.5. Conflict Resolutions

4.5.1. Reserved Keywords vs. Identifier Names

Keywords and identifiers have to be distinguished by the lexer. Therefore, there is no means to decide upfront whether a certain keyword is actually used as a keyword or whether it is used as an identifier in a given context. This limitation is idiomatically overcome by a data type rule for valid identifiers. This data type rule enumerates all keywords which may be used as identifiers and the pure IDENTIFIER terminal rule as seen in Keywords as Identifier listing.

Keywords as Identifier

```
N4JSIdentifier: IDENTIFIER
    | 'get'
    | 'set'
    ...
;
```

4.5.2. Operators and Generics

The ambiguity between shift operators and nested generics arises also from the fact, that Antlr lexer upfront without any contextual information. When implemented naively, the grammar will be broken, since a token sequence `a>>b` can either be part of `List<List<a>> b` or it can be part of a binary operation `int c = a >> b`. Therefore the shift operator may not be defined with a single token but has to be composed from individual characters (see Shift Operator listing).

Shift Operator listing

```
ShiftOperator:
      '>' '>' '>'?
    | '<' '<'
    ;
```

Chapter 5. Flow Graphs

5.1. Flow graphs overview

In this chapter, the control and data flow analyses are introduced. Since not all AST elements are relevant for the control or data flow analyses, a new marker class is introduced called `ControlFlowElement`. All AST elements which are part of the control flow graph implement this class. The term control flow is abbreviated as *CF* and hence `ControlFlowElement`s are abbreviated as *CF elements*.

#120

The following picture shows the control flow graph of the function `f`.

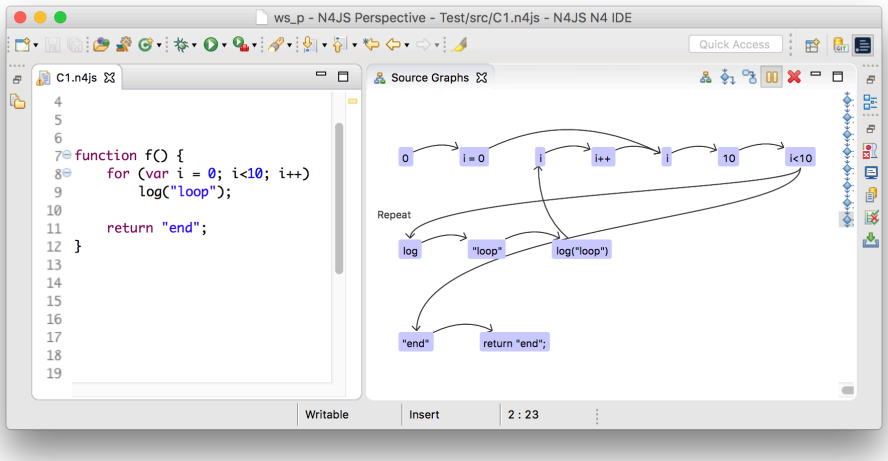


Figure 5.1. Control flow graph of a simple function

5.1.1. Internal graph

Every *internal graph* refers to a single *control flow container*. The graph consists of *nodes* and *edges*, where the edges are instances of `ControlFlowEdge`, and nodes are instances of `Node`. Additionally, a so called *complex node* is used to group nodes that belong to the same CF element.

Internal graph

Control flow graphs are created based on the AST elements. Nevertheless, a fine-grained abstraction is used that is called *internal graph*. The internal graph reflects all control flows and data effects that happen implicitly and are part of the language's semantics. For instance, the for-of statement on iterable objects forks the control flow after invoking the `next()` method. This is done implicitly and not part of the written source code. Moreover, this invocation could cause side effects. These control flows and effects are reflected using the internal graph. To implement analyses that refer to AST elements, an API for flow analyses is provided which hides the internal graph and works with AST elements only. In the following, the term *control flow graph* refers to the internal graph.

Control flow container

At several places in the AST, an execution of statements or elements can happen. Obviously, statements can be executed in bodies of methods or function expressions. In addition, execution can also happen in field initializers or the `Script` itself. Since all these AST elements can contain executable control flow elements (CF elements), they thus contain a control flow graph. In the following, these AST elements are called *control flow containers* or CF containers.

Nodes

Nodes represent complete CF elements or parts of them. For instance, simple CF elements like a `break` statement are represented using only one node. Regarding more complex CF elements that introduce a more complex control flow, due to e.g. nested expressions, several nodes represent one CF element. All nodes of a single CF element are grouped within a complex node.

Edges

Edges reference a start and an end node which reflects a forward control flow direction, which is in the following called forward traverse direction. Traversing from end to start of an edge is thus called backward traverse direction. The so called *next node* is either the end node in context of forward, or the start node in context of backward traverse direction. Edges also reflect the reason of the control flow using a control flow type. The default control flow type is called `Successor` and such edges connect two ordinary subsequent nodes. Other types like `Return` or `Break` indicate control flow

that happens due to return or break statements. A special control flow type is `Repeat` that indicates the entry of a loop body. This edge is treated specially when traversing the control flow graph to avoid infinitive traversals of loops.

Complex node

The complex node always has a single entry and exit node, no matter the control flow it causes. For instance, although the for-statement can contain various control flows among its nested CF elements, its complex node still has a single entry and exit node. This simplifies concatenating subsequent complex nodes, since only their exit nodes have to be connected to the following entry node. Aside from exit and entry node, a complex node usually contains additionally nodes to represent the CF element. These nodes are connected by control flow edge so that their control flow lies within complex node. However, regarding nested CF elements, the control flow leaves and re-enters a complex node. To specify which CF element is nested, a delegating node (`DelegatingNode`) is created that points to the nested CF element.

Consider that source code elements can be nested like expressions that have sub-expressions as in `1 + 2 * 3`. Also statements can contain other statements like in `if (true) return;`. The design of the control flow graph deals with this nesting by mapping CF elements to several nodes. All nodes of one CF element are aggregated into the *complex node*.

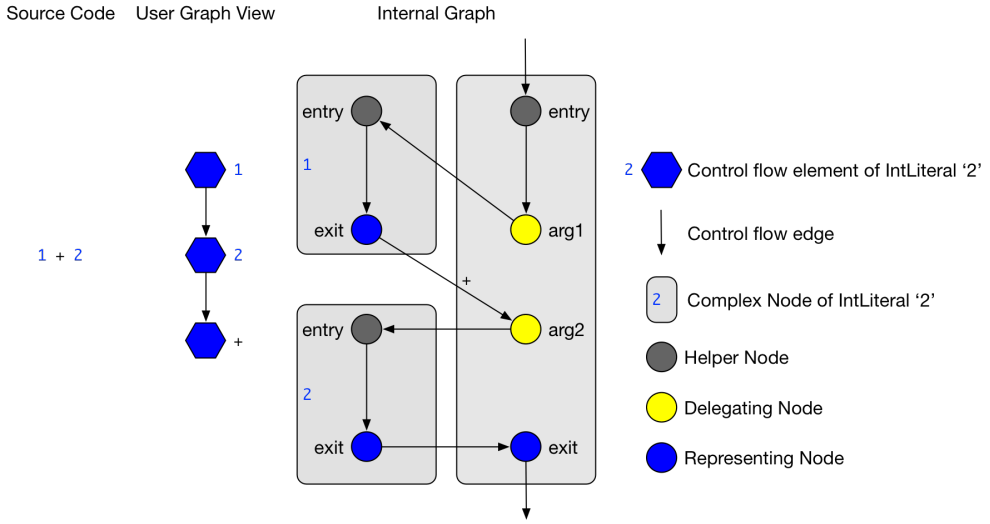


Figure 5.2. The source code of `1+2` creates an internal graph of three complex nodes to deal with nested integer literals

The example in the figure above shows the internal graph produced by the source code `1+2`. Additionally, a simpler version of the internal graph is shown (called *User Graph View*), with which client analyses deal. The user graph view is only a view on the internal graph, but does not exist as an own instance. In the figure, the nesting of the integer literals becomes obvious: The control flow edges of delegating nodes targets entry nodes of different CF elements. Also, there are CF edges from the exit nodes of these nested CF elements to return the control flow.

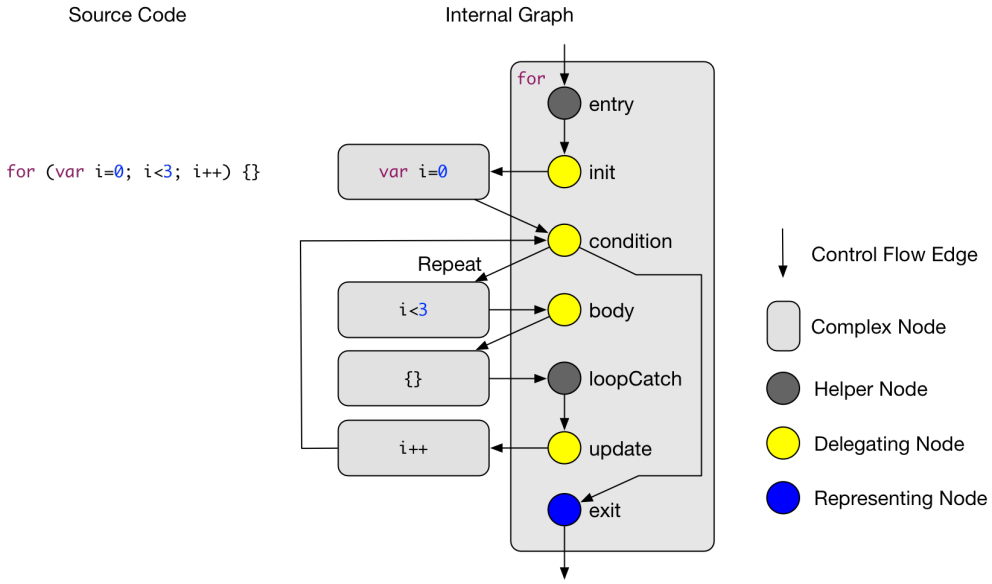


Figure 5.3. Complex Node of for statement

In the above figure, the complex node of the for statement is shown. The details of the complex nodes of the nested CF elements (such as the initializer or the body statement) are omitted. The figure displays the control flow fork after the condition and also shows the *Repeat* edge that targets the for body. The node called *Catch Node* is used in situations when there are jumping control flows introduced for instance by a continue statement. The catch will will then be the target of an control flow edge that starts at the continue statement.

The graph of nodes and edges is constructed in the following order.

- First, for every CF element a complex node and all its nodes are created. Also, the nodes within the complex node are connected according to their control flow behavior.
- Second, all subsequent complex nodes are connected by connecting their exit and entry nodes. Moreover, nested complex nodes are connected by interpreting the delegating nodes.
- Third, jumping control due to `return`, `throw`, `break` or `continue` statements is computed. This is done by first deleting the successor edge of the jumping statement and introducing a new control flow edge that ends at the jump target.

5.1.2. Optimizations

The internal graph contains many nodes to simplify the graph construction. However, these nodes carry no relevant information when traversing the graph. Consequently, in an optimization step, they are removed from the graph for performance reasons.

A node removal for a node n_2 is done by replacing the path $n_1 -> n_2 -> n_3$ by the new path $n_1 -> n_3$. These removals are done for delegating nodes that only have one incoming and one outgoing edge.

A second kind but similar optimization reduces the number of helper nodes that are used as entry nodes. In case a complex nodes consists only of exactly one entry and one exit node, both of these nodes are collapsed into one node. This remaining node then is the representing node of the AST element.

5.1.3. API for client analyses

To implement client analyses based on the control flow graph, the three classes `GraphVisitor`, `GraphExplorer` and `BranchWalker` are provided. They provide the means to visit CF elements in a control flow graph and also to traverse single control flow paths. The method `N4JSFlowAnalyses#analyze` can execute several client analyses in one run to maintain scalability.

5.1.3.1. Mapping from internal to AST elements

The API classes work with AST elements such as `ControlFlowElement` instead of the internally used graph classes `ComplexNode`, `Node` or `ControlFlowEdge`. The mapping from internal classes to AST elements is done in the `GraphVisitor` class.

Note that the control flow graph has the following properties:

- `ExpressionStatement`s are not represented. Instead, only their expressions are represented. Nevertheless, the API can deal with calls that refer to expression statements, e.g. when requesting their successors.
- Control statements are also not represented in the graph, but can also be used in calls to the API. The reason is, that it is unclear when a control statement (e.g. a for loop) is visited exactly.
- Since a `FlowEdge` which connects two `ControlFlowElement`s can represent multiple internal edges, it can have multiple `ControlFlowType`s.

5.1.3.2. Graph visitor

Graph visitors traverse the control flow graphs of every CF container of a script instance in the following two traverse directions:

- *Forward*: from the container's start to all reachable CF graph elements.
- *Backward*: from the container's end to all reachable CF graph elements.

In each traverse direction, the graph visitor visits every reachable CF element and edge. Note that neither empty statements nor control statements are part of the control flow graph. The order of visited CF elements is related to either a breadth or a depth search on the CF graph. However, no specific order assumptions are guaranteed.

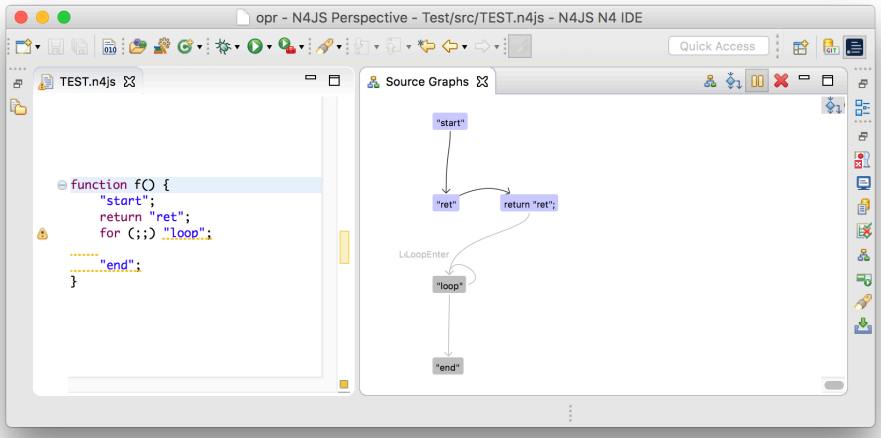


Figure 5.4. The CF elements `"loop"` and `"end"` are dead code and displayed in grey.

5.1.3.3. Graph explorer

Graph visitors can request a *graph explorer* to be activated under specific conditions related to the client analysis. A graph explorer is the start point to analyze control flow branches. The first control flow branches is started directly at the graph explorer's creation site, but of course this first branches might fork eventually. The graph explorer keeps track of all forked branches that originate at its activation site. It also provides the means to join previously forked branches again.

5.1.3.4. Branch walker

With every graph explorer, a branch walker is created that traverses the control flow graph beginning from the activation site of the graph explorer. On every such branch, the two visit methods of CF elements and edges respectively, are called in the order of the traverse direction. Every time the branch forks, the fork method of the branch walker is invoked and creates another branch walker which will continue the traversal on the forked branch. The fork method can be used to copy some path data or state to the newly forked branch walker. Note that every edge is always followed by the branch walker except for repeat edges which are followed exactly twice. The reason to follow them twice is that first, following them only once would hide those control flows that re-visit the same CF elements due to the loop. Second, following them more than twice does not reveal more insights, but only increases the number of branches. When control flow branches merge again, for instance at the end of an `if`-statement, two or more branch walkers are merged into a new succeeding one. The graph explorer provides the means to do this. In case a CF element has no next elements, the branch walker terminates.

5.1.3.5. Example 1: Compute string for each path

Let's assume that we want to compute all control flow branches of a function and use the client API for that. The function `f()` in the following code snippet has four control flow branches: `1 # 2`, `# 3 #`, `# 4 #` and `5`.

Function `f()` has four control flow branches.

```
function f() {
 1;
 if (2)
  3;
 else
  4;
 5;
}
```

To compute these control flow branches, the class `AllBranchPrintVisitor` extends the `GraphVisitor`. Already in the method `initializeMode()` a graph explorer is activated. Note that the method `requestActivation()` can be understood as a `addListener` method for a listener that listens to visit events on nodes and edges. Immediately after the activation request, the first branch walker is created in the method `firstBranchWalker()`.

The first visited CF element of the branch walker will then be the expression `1`. It is formed into a string and added to the variable `curString`. After expression `1`, the flow edge from `1` to `2` is visited. This will concatenate the string `#` to the path string. Variable `curString` will eventually hold the branch string like `1 # 2`. Since the control flow forks after `2`, the method `forkPath()` is called and creates two new instances of a branch walker. These new instances succeed the the first branch walker instance and each traverses one of the branches of the `if`-statement. When the `if`-statement is passed, these two branches are

merged into a new succeeding branch walker. After all branch walkers are terminated, the graph explorer and graph visitor are also terminated. The method `getBranchStrings()` collects all four computed strings from the variable `curString` of all branch walkers.

Implementation of a graph visitor that computes all control flow paths

```
class AllBranchPrintVisitor extends GraphVisitor {
 protected void initializeMode(Mode curDirection, ControlFlowElement curContainer) {
  super.requestActivation(new AllBranchPrintExplorer());
 }

 class AllBranchPrintExplorer extends GraphExplorer {
  class AllBranchPrintWalker extends BranchWalker {
   String curString = "";

   protected void visit(ControlFlowElement cfe) {
    curString += cfe.toString();
   }

   protected void visit(FlowEdge edge) {
    curString += " -> ";
   }

   protected AllBranchPrintWalker forkPath() {
    return new AllBranchPrintWalker();
   }
  }

  protected BranchWalker joinBranches(List<BranchWalker> branchWalkers) {
   // TODO Auto-generated method stub
   return null;
  }

  protected BranchWalkerInternal firstBranchWalker() {
   return new AllBranchPrintWalker();
  }
 }

 List<String> getBranchStrings() {
  List<String> branchStrings = new LinkedList<>();
  for (GraphExplorerInternal app : getActivatedExplorers()) {
   for (BranchWalkerInternal ap : app.getAllBranches()) {
    AllBranchPrintWalker printPath = (AllBranchPrintWalker) ap;
    branchStrings.add(printPath.curString);
   }
  }
  return branchStrings;
 }
}
```

5.1.3.6. Path quantor

Graph explorers are typically used to reason on all branch walkers that start at a specific location. For instance, such a reasoning might determine whether some source element is reachable or whether a variable is used or not. To simplify this, quantors are provided. Since branch walkers originating from a single activation point can fork, the reasoning has to include all these forked branch walkers. Hence, graph explorers are instantiated using a quantor which can be either *For All*, *At Least One* OR *None* that refers to all branches. After all branch walkers of an explorer are terminated, the explorer is regarded as either passed or failed. Paths also can be aborted manually using the methods `pass()` or `fail()`. When *pass* or *fail* are used, the graph explorer might be terminated in the following cases:

- If the quantor of the graph explorer is *For All*, and `fail()` is called on a branch walker.
- If the quantor of the graph explorer is *At Least One*, and `pass()` is called on a branch walker.

Additionally, a graph explorer can be aborted manually by canceling all its branches.

5.1.4. Control flow analyses

5.1.4.1. Dead code analysis

The dead code analysis uses the graph visitor in all four modes and collects all visited CF elements. The collected CF elements are saved separately for every mode. After the graph visitor is terminated, the unreachable CF elements are computed like follows:

- CF elements, that are collected during forward and catch block mode are reachable.

- CF elements, that are collected during islands mode are unreachable.
- CF elements, that are *only* collected during backward mode, are also unreachable.

In a later step, the unreachable elements are merged into unreachable text regions that are used for error markers.

5.2. Dataflow

#331 #464

5.2.1. Dataflow graph

The data flow graph provides means to reason about *symbols*, *effects*, *data flow*, *aliases* and *guards* in the control flow graph. The main classes of the data flow API are `DataflowVisitor` and `Assumption`.

Symbol Symbols represent a program variable in the sence that it represents all AST elements, that bind to the same variable declaration (according to scoping). The terms *symbol* and *variable* are used synonymously.

Effect Effects are reads, writes and declarations of symbols. For instance, a typical CF element with a write effect is an assignment such as `a = null;`. Every effect refers to a single symbol and graph node. The following effects are provided:

- *Declaration*: is the declaration of a variable.
- *Write*: is the definition of a variable's value, which is typically done with an assignment.
- *Read*: is the read of a variable's value, which could happen when passing a variable as an argument to a method call.
- *MethodCall*: is the call of a property method of a variable.

Note that the term *value use* means either write or method call of a variable. The term *value definition* means that a variable is written.

Data flow The term data flow is used for assignments of all kind. For instance, the assigments `a = b`, `a = 1`, `a = null` or even `for (let [a] of [[0],[undefined]]);` are data flows. The data is always flowing from the right hand side to the left hand side.

Alias Due to data flow, other symbols can get important for an analysis. For instance, the data flow `a = b` makes `b` important when reasoning about `a` since the value of `b` is assigned to `a`. In the API is `b` therefore called an alias of `a`.

Guard Guards are conditions that appear in e.g. `it`-statements. For instance, a typical guard is the null-check in the following statement: `if (a == null) foo();`. For every CF element, guards can hold either *always*, *never* or *sometimes*. Note that the null-check-guard always holds at the method invocation `foo();`.

`DataflowVisitor` The class `DataflowVisitor` provides means to visit all code locations where either effects happen or guards are declared. For instance, when a variable is written, the callback method `DataflowVisitor#visitEffect(EffectInfo effect, ControlFlowElement cfe)` gets called. In case a guard is declared, the callback method `visitGuard(Guard guard)` gets called.

`Assumption` The class `Assumption` provides means to track the data flow of a specific symbol from a specific code location. For instance, assumptions are used to detect whether the symbol `s` in the property access `s.prop` is or may be undefined. In this example, the assumption symbol is `s` and its start location is the property access. From there, the data flow of `s` is tracked in backwards traverse direction. Also, (transitive) aliases of `s` are tracked. In case a data flow that happens on `s` or its aliases, the callback method `holdsOnDataflow(Symbol lhs, Symbol rSymbol, Expression rValue)` is called. For every effect that affects `s` or one of its aliases, the callback method `holdsOnEffect(EffectInfo effect, ControlFlowElement container)` is called. And finally, for all guards that hold always/never at the start location regarding symbol `s`, the callback method `holdsOnGuards(Multimap<GuardType, Guard> neverHolding, Multimap<GuardType, Guard> alwaysHolding)` is called.

5.2.2. Dataflow analyses

5.2.2.1. Def#Def / Def#Nothing analysis

A Def#Def analysis finds all defintions of a variable that are always a predecessor of another definition. Its result is a set of all obsolete definition sites.

A Def#!Use analysis finds all definitions of a variable that are not followed by either a read or a method call. These definition are therefore obsolete and can be removed.

Both of these analyses are performed in traverse direction *Forward*.

5.2.2.2. Def|Use#Decl analysis

A Def|Use#Decl analysis finds all preceding *def* or *use* sites of a declarations of a specific variable. The paths might contain other *defs* or *uses* of the same variable. In case such paths exists, the variable is used before it is declared. This analysis is done in traverse direction *Backward*.

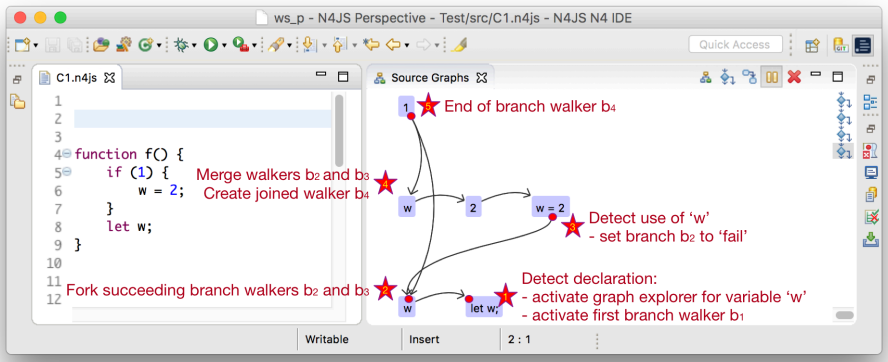
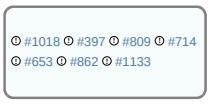


Figure 5.5. Finding use or def sites can be done using the graph visitor in traverse direction *Backward*.

In the above figure a graph visitor would visit all CF elements. When it visits the declaration in line 8 (`let w`), it will activate a graph explorer (star 1 in the figure) for variable `w` . Now, the first branch walker b_1 is created and walks the control in backward traverse direction. When b_1 encounters the exit node of the `if` -statement, it will create two forked branches b_2 and b_3. Now, b_2 enters then the branch of the `if` -statement (star 2), while b_3 traverses directly to the condition of the `if` -statement. Next, b_2 visits the def site of variable `w` (star 3). This means, that there exist a def site of `w` before `w` was declared and hence, an error should be shown. Since there could exist more cases like this, neither the branch walker nor the graph explorer are terminated. When reaching star 4, the two branch walkers b_2 and b_3 are joined and the follow-up branch walker b_4 is created. At star 5, the end the CF container is reached and the b_4 will be terminated. After all branch walkers are terminated, the graph explorer for the declaration site of variable `w` is evaluated: All use or def sites, that were reachable should be marked with an error saying that the declaration has to be located before the use of a variable.

Note this analysis is currently implemented as a control flow analysis since it does not rely on guards, aliases. Also, it only relies on local variables and hence does not need the symbols that are provided by the data flow API.

Chapter 6. External Libraries



External libraries are N4JS projects that are provided by the N4JS IDE: the *built-in/shipped* libraries, and all *3rd-party libraries* that were installed by the *N4JS library manager*. Each external library consist of a valid package.json file located in the project root and an arbitrary number of files supported by N4JS projects, e.g. *.n4js*, *.njsd* and *.js* files. The purpose of the external libraries is to share and to provide core and third party functionality for N4JS developers both in compile and runtime without rebuilding them.

Section 6.3, “Built-in External Libraries / Shipped Code” are external libraries that provide some basic functionality for N4JS programmers, such as the class `N4Injector`.

3rd-party libraries are external libraries that are not built-in/shipped with the N4JS IDE. Instead, they can be installed later by the user from third party providers. Currently, only *npm packages* are supported.

The **N4JS index** is populated when the external libraries are compiled. However, this compilation is only triggered through the library manager, but not when building workspace projects. (Self-evidently, the index is also populated when compiling workspace projects.)

Name clashes of projects can happen and they are solved in the following order:

1. User workspace projects always shadow external libraries.
2. In case of a name clash between a shipped and a 3rd-party library, the 3rd-party library shadows the shipped project.

The **N4JS library manager** is a tool in the N4JS IDE to view and manage external libraries. In particular, the user can (un-)install new 3rd-party libraries, or can trigger the build of all external libraries to re-populate the N4JS index. The library manager also supports other maintenance actions such as deleting all 3rd-party libraries.

6.1. Major Components

External libraries are supported based on different components all over the application.

The followings are the most important ones:

- **External Resources** (`IExternalResource`)
  - These are customized `IResource` implementations for external projects, folders and files.
  - With this approach the `IProject` , `IFolder` and `IFile` interfaces have been implemented. Each implementation is backed by a pure `java.io.File` based resource.
  - When accessing such external resources for example visiting purposes, getting the members of the resource or simply deleting the resource, internally each requests will be directly performed on the wrapped `java.io.File` without accessing the `org.eclipse.core.resources.IWorkspace` instance.
- **External Library Workspace**
  - This is a kind of dedicated workspace for external libraries and their dependencies.
  - Any query requests to retrieve a particular project or any dependencies of a particular project via the `IN4JSCore` singleton service will delegated to its wrapped `N4JSModel` singleton. Internally the `N4JSModel` has a reference to a workspace for all the ordinary workspace projects and another reference to the workspace for external libraries. Each query requests will be forwarded to the workspace for the ordinary projects first, and then to the external library workspace. If ordinary project workspace can provide any meaningful response for a request, then the external library workspace will not be accessed at all. Otherwise the query will be executed against the external library workspace. This fallback mechanism provides a pragmatic solution to the project shadowing feature. The project shadowing will be described in details later in this section.

  - The **External Library Workspace** is only supported and available in the IDE case, in the headless case there are no external libraries available from this dedicated workspace. Since the Xtext index creation and the entire build infrastructure is different, it is supported via target platform file. This is described in more details in a later section (Headless External Library Support]).
- **External Library Preference Store**
  - This preference store is being used to register and un-register external library root folders into its underlying ordered list. A folder is called as an external library root folder if it is neither equal with the Eclipse workspace root nor being nested in the workspace root and contains zero to any external libraries.
  - Whenever any modifications are being saved in this preference store the *External Library Workspace* will be updated as well, new libraries will be registered into the workspace and removed libraries will be cleaned up from the workspace.
  - When the N4JS IDE application is started in production mode, the initial state of the preference store is being pre-populated with default values. This is necessary to provide built-in libraries to end users. These default values and additional advanced configurations will be mentioned in more details later in this section.
- **Library Manager**
  - This service is responsible for downloading and installing third party *npm* packages into the `node_modules` folder of the N4JS IDE. After downloading, the newly-installed and/or updated packages are registered as external libraries into the system.
- **External Library Builder**
  - This service is responsible for updating the persistent Xtext index with the currently available external libraries.
  - Unlike in case of any other ordinary projects, this builder does not triggers a build via the `org.eclipse.core.internal.events.BuildManager` but modifies the persisted Xtext index (`IBuilderState`) directly.
  - Considers shadowed external libraries when updating the persisted Xtext index.
  - Makes sure that the external library related Xtext index is persistent and will be available on the next application startup.
- **External Library Xtext Index Persister**
  - This class is responsible for recovering the consistent external library Xtext index state at application startup.
  - Scheduled on the very first application startup to prepare the Xtext index for the available external libraries.
  - Recovers the Xtext index state after a force quit and/or application crash.
- **External Library Preference Page**
  - Preference page to configure and update the state of the *External Library Preference Store*.
  - Provides a way to install *npm* dependencies as external libraries into the application.
  - Reloads the external libraries. Gets the most recent state of N4JS type definition files and updates the Xtext index content based on the current state of the external libraries.
  - Exports the current npm dependency configuration as a target platform file. This will be discussed in another section ([sec:Headless_External_Library_Support]).
- **Miscellaneous UI Features**
  - Searching for types provided by external libraries.
  - Opening external modules in read-only editor.
  - Navigation between external types.
  - *Project Explorer* contribution for showing external dependencies for ordinary workspace projects.
  - Editor-navigator linking support for external modules.
  - Installing third party npm dependencies directly from package.json editor via a quick fix.

6.1.1. External Resources

This approach provides a very pragmatic and simple solution to support external libraries in both in the `IN4JSCore` and in the `IBuilderState`. While `IN4JSCore` supports a completely transparent way of external libraries via the `IN4JSProject` interface all over in the application, the `IBuilderState` is responsible for keeping the Xtext index content up to date with the external libraries. Below picture depicts the hierarchy between the ordinary `IResource` and the `IExternalResource` instances. As described above each external resource is backed by a `java.io.File` resource and each access and operation being invoked on the `IResource` interface will be delegated to this backing resource.

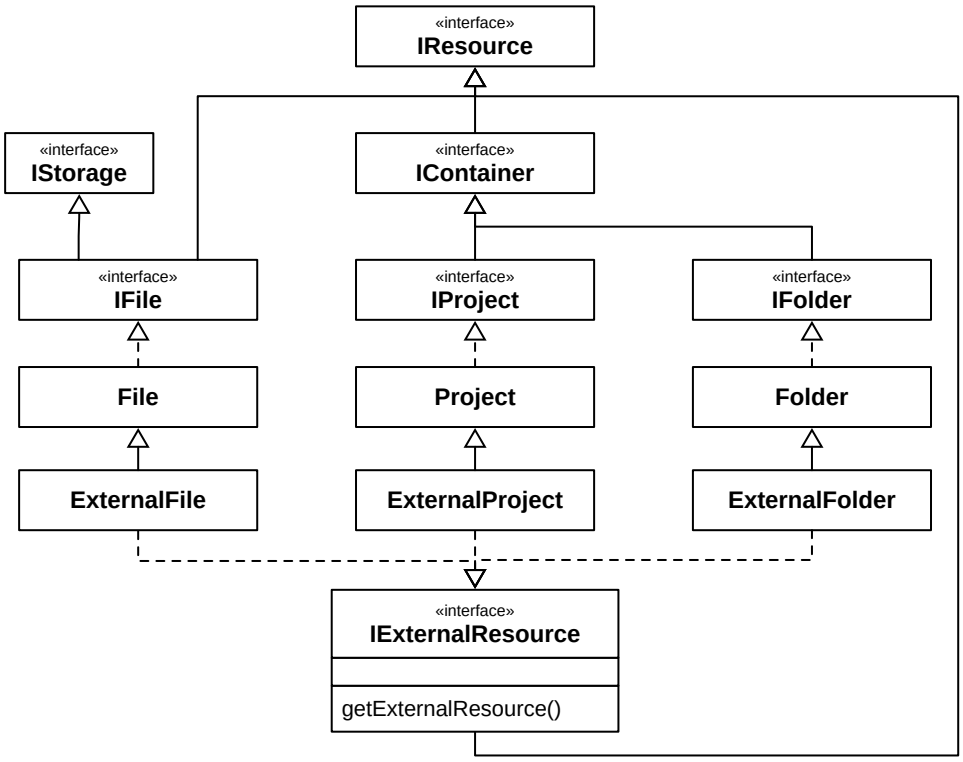


Figure 6.1. External Resources Hierarchy

6.1.2. External Library Workspace

External library workspace is an extension of the `InternalN4JSWorkspace`. This workspace is used for storing and managing external libraries all over the application. External libraries can be registered into the workspace by providing one to many external library root folder locations. The provided root folder locations will be visited in an ordered fashion and the contained external libraries (N4JS projects) will be registered into the application. If an external library from a root folder has been registered, then a forthcoming occurrence of an external library with the same artefact identifier (and same folder name) will be ignored at all. For instance let assume two external library root locations are available `ER1` and `ER2`, also `ER1` contains `P1` and `P2` external libraries, while `ER2` contains `P2` and `P3`. After registering the two roots into the workspace `ER1` will be processed first, and `P1` and `P2` will be registered to the workspace, when processing the forthcoming `ER2` root, `P2` will be ignored at all as an external with the same name exists. Finally `P3` will be registered to the workspace. External libraries cannot be registered directly into the workspace it is done automatically by the *External Library Preference Store* and by the *npm Manager*.

6.1.3. External Library Preference Store

This persistent cache is used for storing an ordered enumeration of registered external library root folder locations. Whenever its internal state is being persisted after a modification, all registered modification listeners will be synchronously notified about this change. All listeners will receive the store itself with the updated state. There are a couple of registered listeners all over the application listening to store update events but the most important one is the *External Library Workspace* itself. After receiving an external library preference store update event, the external library workspace will calculate the changes from its own state: creates a sort of difference by identifying added, removed and modified external libraries. Also tracks external library root location order changes. Once the workspace has calculated the changes[1] it will interact with the *External Library Builder Helper* which will eventually update the persisted Xtext index directly through the `IBuilderState`. After the Xtext index content update all ordinary workspace projects that directly depend either on a built or a cleaned external library will be automatically rebuilt by the external library workspace.

6.1.4. Library Manager

This service is responsible for downloading, installing third party npm dependencies into the local file system. This is done directly by `npm` from `Node.js`. Once an npm package has been downloaded and installed it will be registered into the external library workspace. As part of the registration, the Xtext index content will be updated and all dependent ordinary workspace projects will be rebuilt automatically. An npm package cannot be installed via the *Library Manager* if it already installed previously.

[1] Calculates a list of external library projects that have to be build and another list of projects that have to be cleaned.

6.1.5. External Library Builder

This builder is responsible for updating the persisted Xtext index state with external library content directly through the `IBuilderState`. When providing a subset of external libraries to either build or clean, internally it orders the provided external libraries based on the project dependencies. Also, it might skip building all those external libraries that have are being shadowed by a workspace counterpart. An external library is being shadowed by an ordinary workspace project, if the workspace project is accessible and has exactly the same project name as the external library.

6.1.6. External Library Xtext Index Persister

By default Xtext provides a way to fix corrupted index or to recreate it from scratch in case of its absence. Such inconsistent index states could occur due to application crashes or due to non-graceful application shutdowns. Although this default recovery mechanism provided by Xtext works properly, it is provided only for projects that are available in the Eclipse based workspace (`org.eclipse.core.resources.IWorkspace`) but non of the external libraries are not available from the Eclipse based workspace, so inconsistent external library index content cannot be recovered by this default mechanism. N4JS IDE contributes its own logic to recover index state of external N4JS libraries. When the default Xtext index recovery runs, then it will trigger a external reload as well. This external reload is guaranteed to run always after the default recovery mechanism.

6.1.7. External Library Preference Page

This preference page provides a way to configure the external libraries by adding and removing external library root folders, also allows the user to reorder the configured external library root locations. Besides that, npm packages can be installed into the application as external libraries. Neither removing nor reordering built-in external libraries are supported, hence these operations are disabled for built-ins on the preference page. No modifications will take effect unless the changes are persisted with the `Apply` button. One can reset the configurations to the default state by clicking on the `Restore Defaults` button then on the `Apply` button. The `Reload` button will check whether new type definition files are available for npm dependencies, then reloads the persistent Xtext index content based on the available external libraries. Once the external library reloading has been successfully finished, all dependent workspace projects will be rebuilt as well. From the preference page one can export the installed and used third party npm packages as a target platform. This exported target platform file can be used with the headless compiler. After setting up the headless compiler with this exported target platform file, the headless tool will collect and download all required third party npm dependencies.

6.2. Headless External Library Support

The headless compiler is not capable of supporting built-in libraries. The whole build and Xtext index creation infrastructure is different in the IDE and in the headless case. Also, due to its archive nature (`n4jsc.jar`) of the headless tool, neither the runtime nor the `Mangelhaft` libraries can be loaded into the headless compiler.

The headless compiler supports downloading, installing and using third party `npm` packages. To enable this feature one has to configure the target platform via the `-targetPlatformFile` (or simply `-tp`) and the `-targetPlatformInstallLocation` (or simply `-tl`) arguments.

If the target platform file argument is configured, then all third party dependencies declared in the target platform file will be downloaded, installed and made available for all the N4JS projects before the compile (and run) phase. If the target platform file is given but the target platform install location is not specified (via the `-targetPlatformInstallLocation` argument), then a the compilation phase will be aborted and the execution will be interrupted.

For more convenient continuous integration and testing purposes there are a couple of additional exception cases with respect to the the target platform file and location that users of the headless compiler have to keep in mind. These are the followings:

- `-targetPlatformSkipInstall`. Usually dependencies defined in the target platform file will be installed into the folder defined by option `-targetPlatformInstallLocation`. If this flag is provided, this installation will be skipped, assuming the given folder already contains the required files and everything is up-to-date. Users have to use this flag with care, because no checks will be performed whether the location actually contains all required dependencies.
- If `-targetPlatformSkipInstall` is provided the `-targetPlatformInstallLocation` parameter is completely ignored.
- If `-targetPlatformSkipInstall` is provided the `-targetPlatformFile` parameter is completely ignored.
- If neither `-targetPlatformInstallLocation` not `-targetPlatformFile` parameters are specified the headless tool will treat this case as an implicit `-targetPlatformSkipInstall` configuration.

If the target platform install location is configured, and the target platform file is given as well, then all third party dependencies specified in the target platform file will be downloaded to that given location. If the target platform file is given, but the target platform install location is not specified, then a the compilation phase will be aborted and the execution will be interrupted.

```
java -jar n4jsc.jar -projectlocations /path/to/the/workspace/root -t allprojects --systemLoader cjs -tp /absolute/path/to/the/file
 -tl /path/to/the/target/platform/install/location -rw nodejs -r moduleToRun
```

6.2.1. Custom npm settings

In some cases there is a need for custom npm settings, e.g. custom npm registry. Those kind of configurations are supported via `.npmrc` file (see https://docs.npmjs.com/files/npmrc).

In N4JSIDE user can specify path to his custom configuration file in the preference page.

For the commandline N4JSC.jar provides special option `-npmrcRootLocation` that allows headless compiler to use custom settings.

6.3. Built-in External Libraries / Shipped Code

The library manager is provided with a number of built-in external libraries, consisting of:

- default runtime environments and runtime libraries, e.g. `n4js-es5`, `n4js-runtime-es2015`, `n4js-runtime-v8`.
- a library `n4js.lang` providing N4JS implementations for runtime functionality required by some N4JS language features (currently this contains only dependency injection support, implemented in file `N4Injector.n4js`).
- the mangelhaft test framework.

At runtime, these appear as default libraries in the library manager and are available in the workspace without any further installation. However, at the moment, this is only supported within the N4JS IDE, not the `n4jsc.jar`.

The above libraries are located in the N4JS Git repository in two distinct locations:

- below the top-level folder `n4js-libs`: this is the main source and will be edited by developers working on fixes / improvements of the runtime environments, libraries, and mangelhaft.
- below the folder `shipped-code` in bundle `org.eclipse.n4js.external.libraries`: this is a copy of the code contained below top-level folder `n4js-libs`, plus the transpiled output code, plus NPM dependencies.

The contents of folder `shipped-code` is what will actually be bundled into the N4JS IDE product. These contents should **never** be modified manually. Instead, an MWE2 work flow exists for updating this shipped code: `UpdateShippedCode.mwe2`, located in its own bundle `org.eclipse.n4js.external.libraries.update`. When executed, this work flow will

1. compile the code below top-level folder `n4js-libs`,
2. clean folder `shipped-code`,
3. copy everything from top-level folder `n4js-libs` to folder `shipped-code` (including the transpiled output code generated in step 1, above),
4. run `npm install` in those projects below folder `shipped-code`, that have third-party dependencies (currently this applies only to runtime environment `n4js-node`).

A JUnit test is located in bundle `org.eclipse.n4js.external.libraries.update.tests` that will assert the shipped code is up-to-date. When this test fails, it should be enough to re-run the MWE2 work flow and commit the resulting changes in folder `shipped-code`.

6.4. Additional Notes

6.4.1. Enabling Built-in External Libraries

By default, built-in external libraries (such as *node_modules*, *N4JS Language*, *N4JS Runtime* and *Mangelhaft*), also known as "shipped code", are only available when the N4JS IDE is running in production mode, i.e. when `Platform#inDebugMode()` and `Platform#inDevelopmentMode()` both return `false`). This means built-in external libraries are not available in the N4JS IDE when it is started from Eclipse development environment or when running any plug-in, plug-in UI and/or SWTBot tests.

There are two ways of enabling built-in external libraries in NON-production mode:

1. start the application with the following VM argument (but note the comment below!):

   ```
   -Dorg.eclipse.n4js.includesBuiltInLibraries=true
   ```

2. invoke methods `ExternalLibrariesSetupHelper#setupExternalLibraries(boolean, boolean)` with first argument set to `true`, and `ExternalLibrariesSetupHelper#tearDownExternalLibraries(boolean)` inside the test

code. For SWTBot tests there are corresponding convenience methods provided in `BaseSwtBotTest`. For an example, see `N4jsTasksExampleSwtBotTest`.

The first option is only intended for the case of starting an N4JS IDE from an Eclipse development environment; in all kinds of tests, the helper methods mentioned above should be used instead (i.e. option 2.). In other words, the configuration property `org.eclipse.n4js.includesBuiltInLibraries` should only be used in our two launch configurations "N4JS__IDE.launch" and "N4JS_N4__IDE.launch" that are checked-in to our repository.

6.5. Future Work

Some aspects not covered in current design, but worth consideration in the future

6.5.1. Multiple Dependency Scope

npm scope dependencies

DEPENDENCY_DEVELOPMENT	https://docs.npmjs.com/files/package.json#devdependencies
DEPENDENCY_PEER	https://docs.npmjs.com/files/package.json#peerdependencies
DEPENDENCY_BUNDLE	https://docs.npmjs.com/files/package.json#bundleddependencies
DEPENDENCY_OPTIONAL	https://docs.npmjs.com/files/package.json#optionaldependencies
DEPENDENCY_PROVIDES	http://www.rpm.org/wiki/PackagerDocs/Dependencies#Provides
DEPENDENCY_WEAK	http://www.rpm.org/wiki/PackagerDocs/Dependencies#Weakdependencies

6.5.2. Run Tests from TestLibrary

Imagine we are implementing some API, and we want to run tests for that API. Tests are delivered to us as separate package, and there is not direct association between implementation and test projects (tests are not depending on implementation). Still we want to run provided tests to see if our implementation complies with API tests, e.g. AcceptanceTest suite for Application written against application sdk.

Appendix A. Acronyms

CDep	Compile-Time Dependency	RDep	Run-Time Dependency
LDep	Load-Time Dependency	IDep	Initialization-Time Dependency
EDep	Execution-Time Dependency	AC	Acceptance Criteria
ANTLR	ANother Tool for Language Recognition	API	Application Programming Interface
AST	Abstract Syntax Tree	ASI	Automatic Semicolon Insertion
AST	Abstract Syntax Tree	BNF	Backus-Naur Form
CA	Content-Assist	CSP	Constraint Satisfaction Problem
CLI	Command Line Interface	DOM	Document Object Model
DSL	Domain Specific Language	EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework	EPL	Eclipse Public License
FQN	Fully Qualified Name	GLB	Greatest Lower Bound, also known as infimum
GPL	GNU General Public License	IDE	Integrated Development Environment
IDL	Interface Definition Language	LSP	Liskov Substitution Principle
LUB	Least Upper Bound, also known as supremum	N4JS	NumberFour JavaScript
UI	User Interface	UML	Unified Modeling Language
VM	Virtual Machine	XML	Extensible Markup Language
XSLT	XSL Transformations	XSL	Extensible Stylesheet Language
WYSIWYG	What You See Is What You Get	WLOG	without loss of generality

Appendix B. Licence

This specification and the accompanying materials is made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at http://www.eclipse.org/legal/epl-v10.html

Eclipse Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE (`AGREEMENT`). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

`Contribution` means:

1. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
2. in the case of each subsequent Contributor:
   a. changes to the Program, and
   b. additions to the Program;

   where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which:
   i. are separate modules of software distributed in conjunction with the Program under their own license agreement, and
   ii. are not derivative works of the Program.

`Contributor` means any person or entity that distributes the Program.

`Licensed Patents` mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

`Program` means the Contributions distributed in accordance with this Agreement.

`Recipient` means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

1. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.
2. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.
3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.
4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and
2. its license agreement:

    a. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

    b. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

    c. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

    d. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and
2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor (**`Commercial Contributor`**) hereby agrees to defend and indemnify every other Contributor (**`Indemnified Contributor`**) against any losses, damages and costs (collectively **`Losses`**) arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN **`AS IS`** BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement , including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Appendix C. Bibliography

N4JS Project. (2018). *N4JS Language Specification*. Retrieved from https://www.eclipse.org/n4js/spec/N4JSSpec.html