Published in DailyJS

Alena Khineika Follow

Jul 3, 2018 · 15 min read · Listen

Save

Compiler in JavaScript using ANTLR

Early this year, I joined the team which develops and supports MongoDB Compass, a graphical interface for MongoDB. Compass users via Intercom requested a tool which would enable them to write database queries using any convenient programming language supported by the MongoDB driver. So, we needed a capability to transform (compile) the Mongo Shell language into other languages and vice versa.

This article can be viewed as both a practical guide for writing a JavaScript compiler and a theoretical resource that describes the basic concepts and principles of compiler design. At the end of the article you can find a list of all used references, as well as links to supplementary materials for further study. Information in the article is presented consistently, beginning with the study of the subject area and then gradually complexifying the functionality of the example application. If when reading the article you feel that you do not catch the transition from one step to another, examining the full version of this program might help.

Terms

Compiler — a program that transforms a source program written in a high-level programming language into an equivalent program in another language that can be executed without the compiler.

Lexer — an element of a compiler that performs lexical analysis.

Lexical analysis (Tokenizing) — the process of converting a sequence of input characters into meaningful sequences called lexemes or tokens.

Parser — an element of a compiler that takes input data and builds a parse tree.

Parsing (Syntax analysis) — the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.

Lexeme (Token) — a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyser as an instance of that token.

Visitor — a pattern of working with a tree, when processing of the node descendants requires invoking their traversal methods manually.

Listener (Walker) — a pattern of working with a tree, when processing methods for node descendants are called automatically. The listener interface contains `enterNode()` and `exitNode()` methods, which are invoked when entering and exiting a given node.

Parse tree — an ordered, rooted tree that produced by parser. It concretely reflects the syntax of the input language and clearly contains all relationships between individual elements.

Abstract syntax tree (AST) — a tree without nodes and edges for those syntax rules that do not affect the program semantics (unlike parse tree).

Universal abstract syntax tree (UAST) — a normalised form of AST with language-independent annotations.

Depth-first search (DFS) — one of the graph traversal methods. As the name implies, the strategy is to go as deep as possible into the graph.

nodes.

Leaf (Terminal node) — a node without children.

Parent — a node with a child. Each tree node has zero or more child nodes.

Literal — a notation for representing a some fixed value or data type.

Code generator — a program that takes an internal representation of the source program as input and transforms it into the target language.

Investigation

There are fundamentally three ways of transforming one programming language into another (source-to-source transformation):

- Use existing parsers for specific programming languages;
- Create your own parser;
- Use a tool or a library to generate parsers.

The first way is for sure effective, but it can be used only for the most known and supported languages. For JavaScript there are such parsers as Esprima, Falafel, UglifyJS, Jison and others. Before writing something new from scratch, it is worth to

check the existing tools, there's a good chance that they might have all the required functionality.

If you were not lucky enough to find a parser for the language you need or the parser, which you have found, does not satisfy all your requirements, you can resort to the second way and write a compiler yourself.

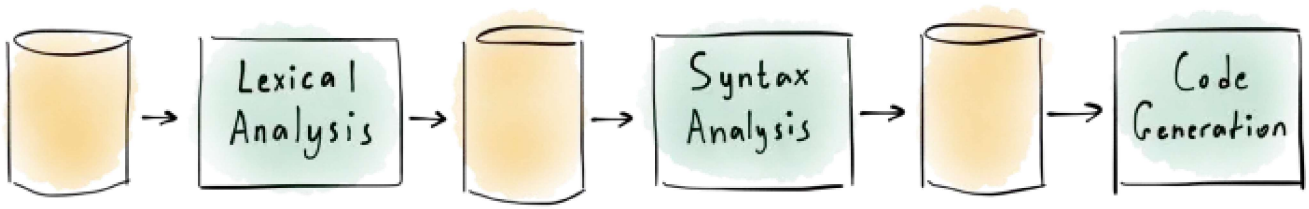
Super Tiny Compiler might be a good start point for understanding the process of writing a compiler from scratch. If you remove all the comments this file would only be ~200 lines of actual code, but you will find there all the basic principles of a modern compiler.

Author of the Implementing a Simple Compiler on 25 Lines of JavaScript article shares his own experience of wri 854 5 JavaScript. It covers such concepts as lexical analysis, syntax analysis (parsing) and code generation.

How to write a simple interpreter in JavaScript is another useful resource that performs a basic overview of writing an interpreter for a simple language that can be used by a calculator application.

Writing a compiler from scratch is a complex and laborious process, which requires a proper preliminary investigation of the syntax features of compiled languages. Not only the keywords, but also their position relative to each other must be recognised. The rules of source code analysis must be unambiguous and produce an identical output under the same input conditions.

Tools and libraries for parser generation can help with this task. They split the raw source code into tokens (lexical analysis), then match the linear token sequences to their formal grammar (parsing) and put them into a tree-like organised, ordered, rooted structure, from which a new code can be generated. Some examples are provided in the Parsing in JavaScript: Tools and Libraries article.

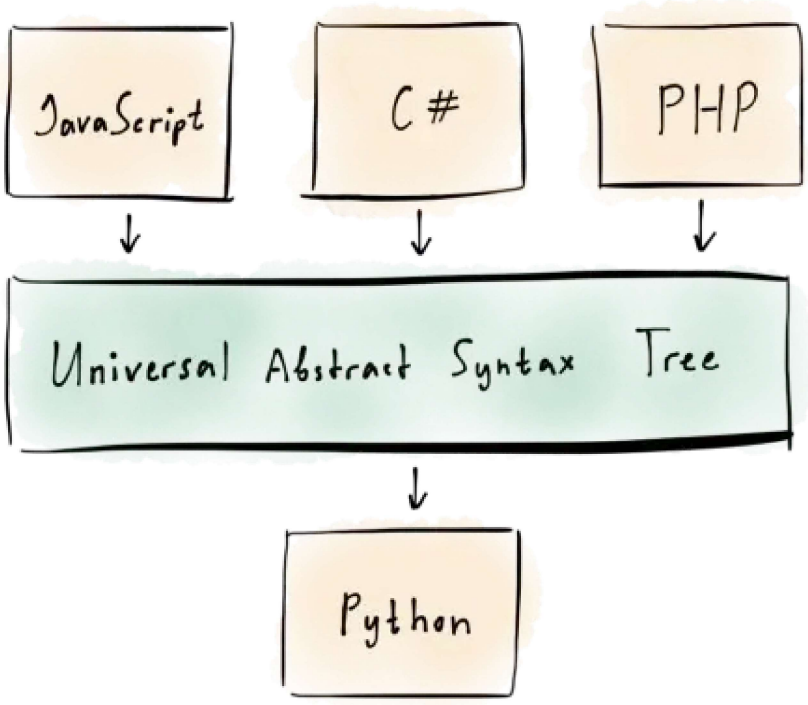


At first glance it may seem that we have done already all we need to solve the problem, however, to make the parser be able to recognise the source code, we have to spend more man-hours to write instructions (grammars). And in case if the compiler must support several programming languages, the task becomes much more complicated.

It is quite obvious that we are not the first developers to encounter this problem. Any IDE performs code analysis. Babel transforms modern JavaScript into a standard format supported by all browsers. This means that there must be existing grammars that we could reuse and by this not only simplify the task, but also prevent a number of potentially serious errors and inaccuracies.

Thus, we chose ANTLR, which best meets our requirements, as it contains grammars for almost all programming languages.

Alternatively, you can try Babelfish, which can parse any file, in any supported language, extracting an AST from it and converting it to a UAST, where nodes are independent from the source language syntax. The input can be JavaScript or C# code, but there will be no differences at the UAST level. In terms of compilers, the process of converting AST to a universal type is known as a transformation process.



A beginner compiler developer may also be interested in Astexplorer, an interface that allows previewing a syntax tree for the given code fragment and parser. It can be useful for debugging or getting acquainted with the AST structure.

Writing a compiler

ANTLR (Another Tool for Language Recognition) is a parser generator written in Java. It analyses the input code basing on grammars and converts it into an organised structure which can be used to create an abstract syntax tree. ANTLR 3 had AST generation function, but it was removed later in ANTLR 4 in support of StringTemplates, and this version uses only the parse tree concept.

For further information, refer to the documentation or check a The ANTLR Mega Tutorial, which explains the basics, what a parser is, what it can be used for, how to setup, how to test and much more with tons of examples.

For converting one programming language into another, we chose ANTLR 4 and ECMAScript.g4, which is one of its grammars. We preferred JavaScript because its syntax matches the MongoDB shell language syntax and is also the development language of the Compass application. Interestingly, we can generate a parse tree using C# lexer and parser and traverse it using ECMAScript grammar nodes.

Of course it requires more in-depth investigation and testing. And we already can say that not all code structures will be correctly recognised by default and it will be necessary to extend the traversal functionality with new methods and checks. At the same time, it is also obvious that ANTLR is a great tool when several parsers are required within a single application.

ANTLR provides us a list of auxiliary files for working with trees. Since the content of the auxiliary files is directly related to the rules defined in the grammar, any changes to the grammar will require regeneration of these files. This means that we should not use them directly for writing a code; otherwise we will lose the changes with the next iteration. We should create new classes and inherit them from the classes created by ANTLR.

The code generated by ANTLR aids the creation of parse tree, which is the basic mean of new code generation. The principle is to call the child nodes from left to right (assuming that it is the order of source code) to return the formatted code they represent.

If the node is a literal, the actual value must be returned. It is more difficult than it seems, if the accuracy of the result is important. It requires accurate output of floating-point numbers, as well as numbers in different numeral systems. For string literals, it is necessary to consider the supported type of quotes and the sequences

of characters that need to be escaped. Should code comments be supported? Should the user input format (i.e. spaces and empty lines) be kept, or should the code be transformed to a more standard human-readable form? On the one hand, the second option will look more professional; on the other hand, the user of your compiler may be not satisfied if he expects the original code format of the output. There is no universal solution for these problems, as they require more detailed investigation of the scope of the compiler.

Let's take a look at a simpler example to focus on the basics of writing a compiler using ANTLR.

ANTLR installation

```
$ brew cask install java
$ cd /usr/local/lib
$ curl -O http://www.antlr.org/download/antlr-4.7.1-complete.jar
$ export CLASSPATH=".:/usr/local/lib/antlr-4.7.1-
complete.jar:$CLASSPATH"
$ alias antlr4='java -Xmx500M -cp "/usr/local/lib/antlr-4.7.1-
complete.jar:$CLASSPATH" org.antlr.v4.Tool'
$ alias grun='java org.antlr.v4.gui.TestRig'
```

To check that installation is completed successfully, launch *org.antlr.v4.Tool* directly:

```
$ java org.antlr.v4.Tool
```

ANTLR version and command help should be displayed.

```
Alenas-MacBook:js-runtime temera$ java org.antlr.v4.Tool
ANTLR Parser Generator  Version 4.7.1
 -o ___              specify output directory where all output is generated
 -lib ___            specify location of grammars, tokens files
 -atn                generate rule augmented transition network diagrams
 -encoding ___       specify grammar file encoding; e.g., euc-jp
 -message-format ___ specify output style for messages in antlr, gnu, vs2005
 -long-messages      show exception details when available for errors and warnings
 -listener           generate parse tree listener (default)
 -no-listener        don't generate parse tree listener
 -visitor            generate parse tree visitor
 -no-visitor         don't generate parse tree visitor (default)
 -package ___        specify a package/namespace for the generated code
 -depend             generate file dependencies
 -D<option>=value    set/override a grammar-level option
 -Werror             treat warnings as errors
 -XdbgST             launch StringTemplate visualizer on generated code
 -XdbgSTWait         wait for STViz to close before continuing
 -Xforce-atn         use the ATN simulator for all predictions
 -Xlog               dump lots of logging info to antlr-timestamp.log
 -Xexact-output-dir  all output goes into -o dir regardless of paths/package
Alenas-MacBook:js-runtime temera$
```

Creating a Node.js project with ANTLR

```
$ mkdir js-runtime
$ cd js-runtime
$ npm init
```

Install *JavaScript runtime*. It requires *antlr4* npm package (*JavaScript target for ANTLR 4*).

```
$ npm i antlr4 --save
```

Download the *ECMAScript.g4* grammar, which will be fed to ANTLR later.

```
$ mkdir grammars
$ curl --http1.1 https://github.com/antlr/grammars-
v4/blob/master/ecmascript/ECMAScript.g4 --output
grammars/ECMAScript.g4
```

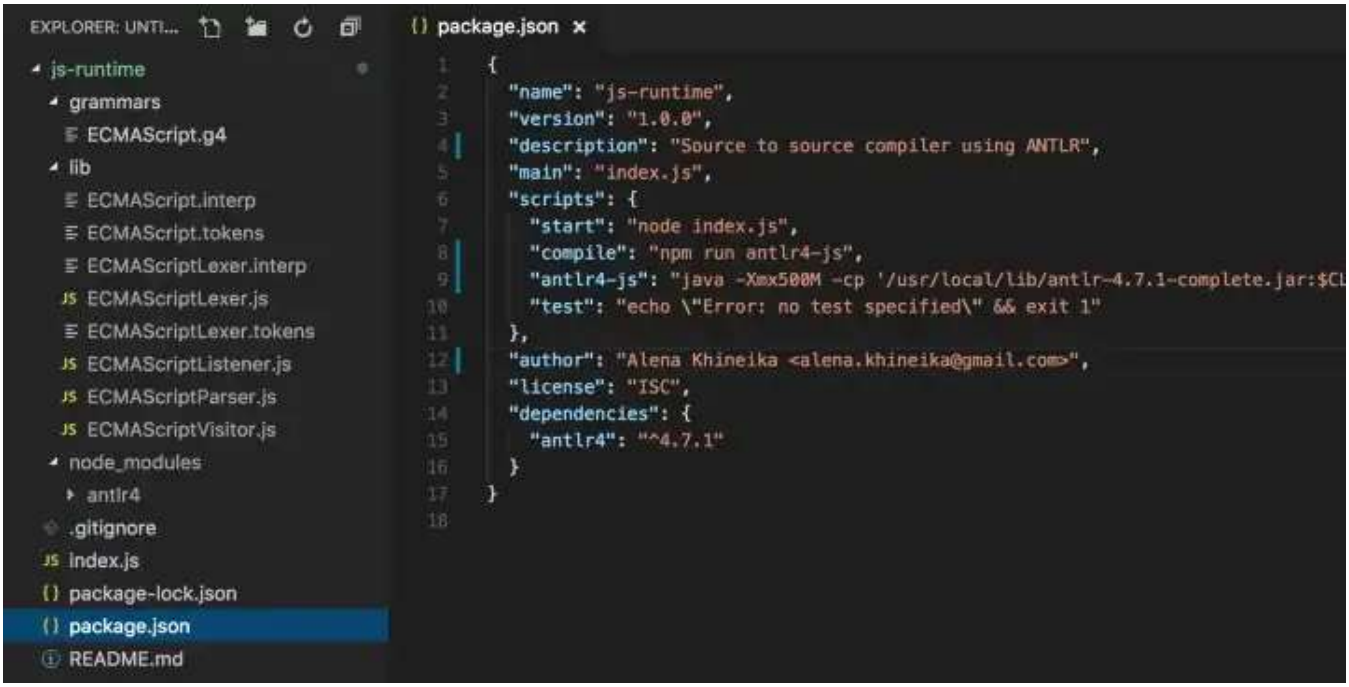
By the way, on the Development Tools section of the ANTLR page you can find links to plugins for such IDEs as Intellij, NetBeans, Eclipse, Visual Studio Code, and jEdit. Colour

themes, semantic error checking and diagram visualisation can help to write and debug grammars.

Finally, let's run ANTLR.

```
$ java -Xmx500M -cp '/usr/local/lib/antlr-4.7.1-
complete.jar:$CLASSPATH' org.antlr.v4.Tool  -Dlanguage=JavaScript -
lib grammars -o lib -visitor -Xexact-output-dir
grammars/ECMAScript.g4
```

Save this script into *package.json* to always have access to it. Any changes to the grammar file require restarting ANTLR.



If we open *lib* folder, we will see that ANTLR has created a list of files for us. Let's take a closer look at these files:

- **ECMAScriptLexer.js** splits a source code character stream into a token stream according to the rules specified in the grammar.
- **ECMAScriptParser.js** generates an abstract connected tree structure (i.e. parse tree) from the token stream.
- **ECMAScriptVisitor.js** is responsible for traversing the generated tree. Technically, we could manually process the tree by depth-first recursive traversal of children. However, if we have a large number of node types and

complex processing logic, it is preferable to visit each node type using its special predefined method, as visitor does.

Note that by default ANTLR does not create **Visitor.js*. The standard tree traversal method in ANTLR is listener. If you want to generate and then use visitor instead of listener, you need to explicitly specify this with the `` `visitor` `` flag, as we did in our script:

```
$ java -Xmx500M -cp '/usr/local/lib/antlr-4.7.1-
complete.jar:$CLASSPATH' org.antlr.v4.Tool  -Dlanguage=JavaScript -
lib grammars -o lib -visitor -Xexact-output-dir
grammars/ECMAScript.g4
```

The operating principle (and the result) of both methods is very similar, however visitor provides more clean code and better control over the transformation process. The user can set the tree node visiting order or disable visiting. Also, the user can modify the existing nodes during traversal, which eliminates the need to store information about visited nodes. This subject is reviewed with examples in the article Antlr4 — Visitor vs Listener Pattern.

Source code analysis and syntax tree construction

Let's finally write some code. You can easily find many examples of similar code if you will search by combination of "ANTLR" and "JavaScript".

```
const antlr4 = require('antlr4');
const ECMAScriptLexer = require('./lib/ECMAScriptLexer.js');
const ECMAScriptParser = require('./lib/ECMAScriptParser.js');

const input = '{x: 1}';

const chars = new antlr4.InputStream(input);
const lexer = new ECMAScriptLexer.ECMAScriptLexer(chars);

lexer.strictMode = false; // do not use js strictMode

const tokens = new antlr4.CommonTokenStream(lexer);
const parser = new ECMAScriptParser.ECMAScriptParser(tokens);
const tree = parser.program();

console.log(tree.toStringTree(parser.ruleNames));
```

js-runtime-index-1.js hosted with ❤ by GitHub view raw

We have just transformed the original string to a tree in LISP format (standard tree output format in ANTLR 4) using lexer and parser. According to the grammar, the root node of the ECMAScript tree is the `program` rule, so we chose it as the starting point of tree traversal in our example. However, this does not mean that this node must always be the first. It would be absolutely correct to begin traversal with `expressionSequence` for the original `{x: 1}` string.

```
const tree = parser.expressionSequence();
```

js-runtime-expression-sequence.js hosted with ❤ by GitHub view raw

```
Alenas-MacBook:js-runtime temera$ npm start

> js-runtime@1.0.0 start /Users/temera/www/js-runtime
> node index.js

(expressionSequence (singleExpression (objectLiteral { (propertyNameAndValueList (propertyAssignment (property
Name (identifierName x)) : (singleExpression (literal (numericLiteral 1))))) })))
Alenas-MacBook:js-runtime temera$
```

If you remove `.toStringTree()` formatting, you will see the internal structure of the tree object.

Conventionally, the whole process of transforming one programming language into another can be divided into three major stages:

- Source code analysis.

- Syntax tree construction.
- New code generation.

As we can see, ANTLR can greatly simplify the process, so two complete stages can be performed in few lines. Certainly we will return to them, update the grammars and transform the tree, but still the groundwork is already done. Grammars downloaded from the repository can also be incomplete or buggy, but at the same time they may be free of the mistakes you could potentially made if you started writing a grammar from scratch. The point is that you should not put blind trust in a code written by other developers, but you can save time writing identical rules to improve existing ones, and maybe the next generation of ANTLR beginners will download your well-polished version of the grammar.

Code generation

Let's create a new *codegeneration* directory in the project and a new *PythonGenerator.js* file inside it.

```
$ mkdir codegeneration
$ cd codegeneration
$ touch PythonGenerator.js
```

As you may have guessed, we are going to transform something from JavaScript to Python as an example.

The generated *ECMAScriptVisitor.js* file contains a huge list of methods, each of which is automatically called during tree traversal if the corresponding node is visited. And at this point we can change the current node. To do this, create a class that will be inherited from ECMAScriptVisitor and redefine the required methods.

```
const ECMAScriptVisitor = require('../lib/ECMAScriptVisitor').ECMAScriptVisitor;

/**
 * This Visitor walks the tree generated by parsers and produces Python code
 *
 * @returns {object}
 */
class Visitor extends ECMAScriptVisitor {
  /**
   * Entry point of tree visiting
   *
   * @param {object} ctx
   * @returns {string}
   */
  start(ctx) {
    return this.visitExpressionSequence(ctx);
  }
}

module.exports = Visitor;
```

js-runtime-python-generator-1.js hosted with ❤ by GitHub view raw

In addition to the methods corresponding to the syntax rules of the grammar, ANTLR also supports four special public methods:

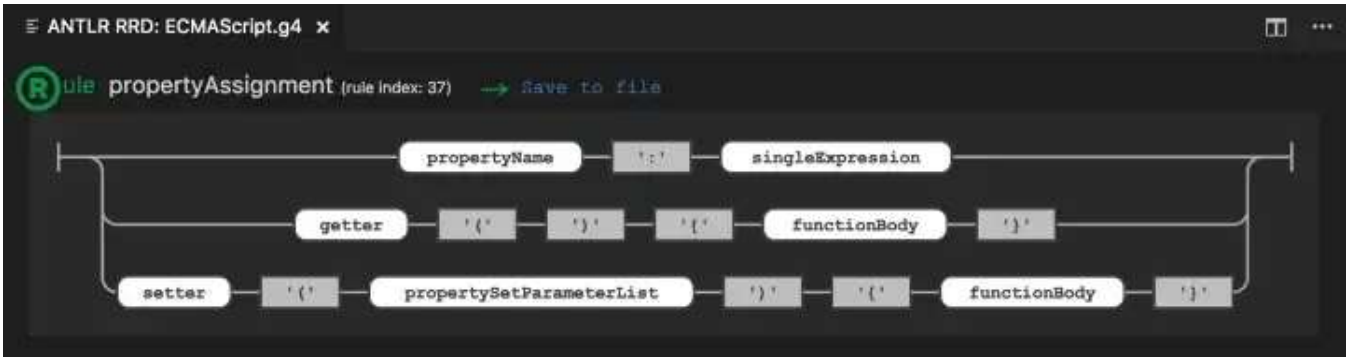
- **visit**() is responsible for tree traversal;
- **visitChildren**() is responsible for node traversal;
- **visitTerminal**() is responsible for token traversal;
- **visitErrorNode**() is responsible for incorrect token traversal.

Let's implement `visitChildren()`, `visitTerminal()` and `visitPropertyExpressionAssignment()` methods.

```
/**
 * Visits children of current node
 *
 * @param {object} ctx
 * @returns {string}
 */
visitChildren(ctx) {
  let code = '';

  for (let i = 0; i < ctx.getChildCount(); i++) {
    code += this.visit(ctx.getChild(i));
  }

  return code.trim();
}

/**
 * Visits a leaf node and returns a string
 *
 * @param {object} ctx
 * @returns {string}
 */
visitTerminal(ctx) {
  return ctx.getText();
}

/**
 * Visits Property Expression Assignment
 *
 * @param {object} ctx
 * @returns {string}
 */
visitPropertyExpressionAssignment(ctx) {
  const key = this.visit(ctx.propertyName());
  const value = this.visit(ctx.singleExpression());

  return `'${key}': ${value}`;
}
```

is-runtime-python-generator-2.js hosted with ❤ by GitHub view raw

The `visitPropertyExpressionAssignment` function visits the node responsible for assigning the `1` value to the `x` parameter. In Python, string parameters of an object must be enclosed in single quotes, while in JavaScript it is optional. In this particular case, this is the only modification we need to transform a fragment of JavaScript code into Python code.



Add the `PythonGenerator` call to *index.js*:

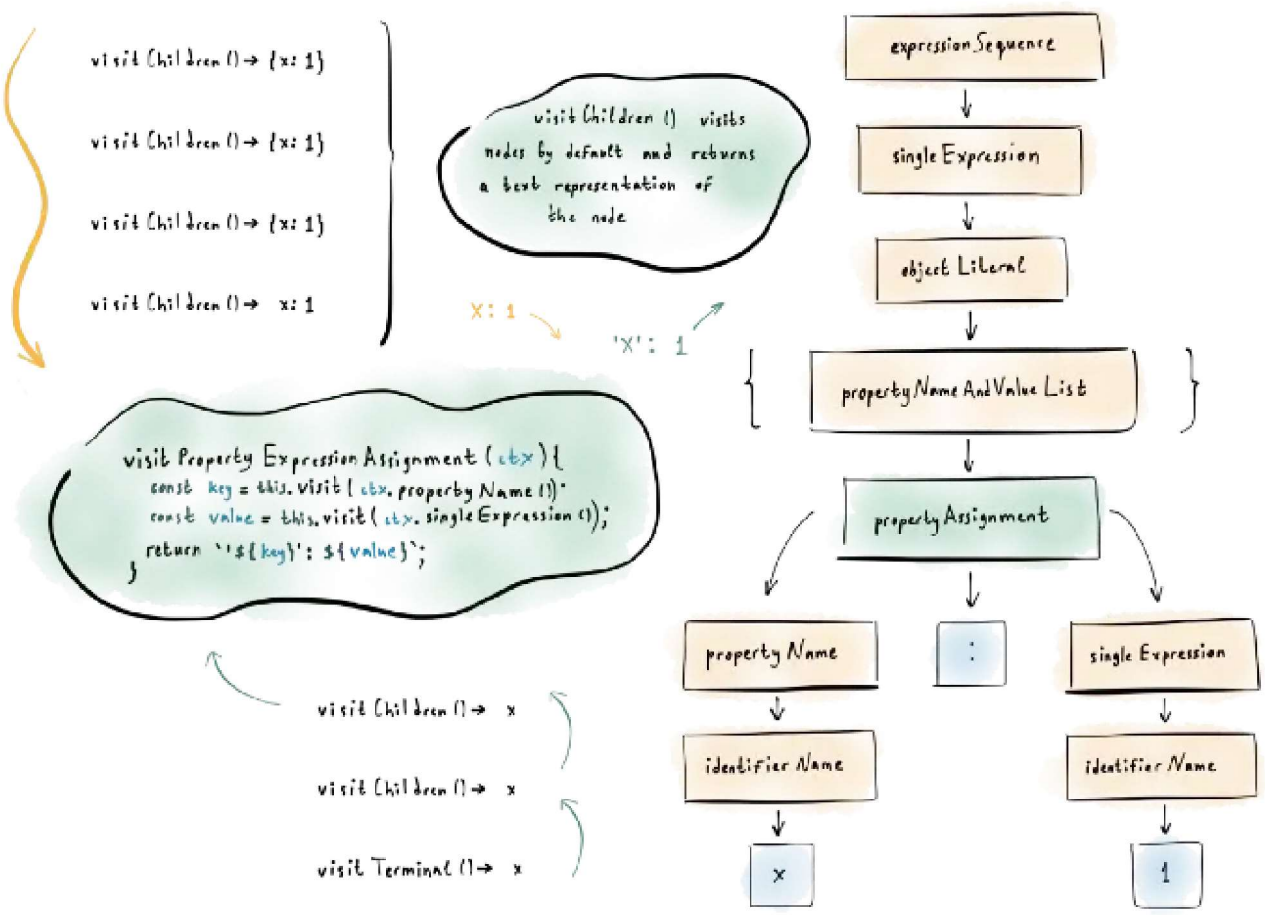
```
1  console.log('JavaScript input:');
2  console.log(input);
3  console.log('Python output:');
4
5  const output = new PythonGenerator().start(tree);
6
7  console.log(output);
```

js-runtime-index-2.js hosted with ❤ by GitHub view raw

At the program output we can see that our compiler has successfully done the task and transformed a JavaScript object into a Python object.

```
Alenas-MacBook:js-runtime temera$ npm start

> js-runtime@1.0.0 start /Users/temera/www/js-runtime
> node index.js

JavaScript input:
{x: 1}
Python output:
{'x': 1}
Alenas-MacBook:js-runtime temera$
```

We traverse the tree from the parent to the child, gradually descending to the leaves. Then we proceed in the reverse order and substitute the formatted values to a higher level, thus replacing the entire node chain with their text representation corresponding to the syntax of the new programming language.



Let's add some debug info to our `visitPropertyExpressionAssignment` function:

```
// Return the text of all tokens in the stream
console.log(ctx.getText());
// How many children are there? If there is none, then this node represents a leaf node
console.log(ctx.getChildCount());
// console.log(ctx.propertyName().getText()) 'x' property
console.log(ctx.getChild(0).getText());
// :
console.log(ctx.getChild(1).getText());
// console.log(ctx.singleExpression().getText()) '1' value
console.log(ctx.getChild(2).getText());
```

js-runtime-visit-roperty-assignment.js hosted with ❤ by GitHub view raw

Children can be referred by name or by order number. The children are also nodes, so to retrieve the text value instead of their object representation we used the `.getText()` method.

Let's update our *ECMAScript.g4* and teach our compiler to recognise a `Number` keyword.

```
| singleExpression '=' expressionSequence                    # AssignmentExpression
| singleExpression assignmentOperator expressionSequence     # AssignmentOperatorExpression
| This                                                       # ThisExpression
| Identifier                                                 # IdentifierExpression
| Number arguments                                           # NumberExpression
| literal                                                    # LiteralExpression
| arrayLiteral                                               # ArrayLiteralExpression
| objectLiteral                                              # ObjectLiteralExpression
| '(' expressionSequence ')'                                 # ParenthesizedExpression
;
```

```
Function   : 'function';
This       : 'this';
With       : 'with';
Default    : 'default';
If         : 'if';
Throw      : 'throw';
Delete     : 'delete';
In         : 'in';
Try        : 'try';
Number     : 'Number';

/// 7.6.1.2 Future Reserved Words
Class   : 'class';
Enum    : 'enum';
```

Regenerate visitor to apply the changes we made to the grammar.

```
$ npm run compile
```

Now let's update *PythonGenerator.js* and add a list of new methods to it.

```
1  /**
2   * Because Python doesn't need `New`, we can skip the first child
3   *
4   * @param {object} ctx
5   * @returns {string}
6   */
7  visitNewExpression(ctx) {
8    return this.visit(ctx.singleExpression());
9  }
10
11 /**
12  * Visits `Number` Keyword
13  *
14  * @param {object} ctx
15  * @returns {string}
16  */
17 visitNumberExpression(ctx) {
18   const argumentList = ctx.arguments().argumentList();
19
20   // JavaScript Number requires only one argument,
21   // otherwise method returns error message
22   if (argumentList === null || argumentList.getChildCount() !== 1) {
23     return 'Error: Number requires one argument';
24   }
25
26   const arg = argumentList.singleExpression()[0];
27   const number = this.removeQuotes(this.visit(arg));
28
29   return `int(${number})`;
30 }
31
32 /**
33  * Removes quotes from string
34  *
35  * @param {String} str
36  * @returns {String}
37  */
38 removeQuotes(str) {
39   let newStr = str;
40
41   if (
42     (str.charAt(0) === '"' && str.charAt(str.length - 1) === '"') ||
43     (str.charAt(0) === '\'' && str.charAt(str.length - 1) === '\'')
44   ) {
45     newStr = str.substr(1, str.length - 2);
46   }
47
48   return newStr;
```

```
49    }
```

child (remove `new` keyword). Then we replace `Number` keyword with `int`, which is its Python equivalent. Since `Number` is an expression, we have access to its arguments via the `.arguments()` method.

```
Alenas-MacBook:js-runtime temera$ npm start

> js-runtime@1.0.0 start /Users/temera/www/js-runtime
> node index.js

JavaScript input:
{x: new Number(1)}
Python output:
{'x': int(1)}
Alenas-MacBook:js-runtime temera$
```

In the same way we can process all the methods listed in *ECMAScriptVisitor.js* and transform all JavaScript literals, symbols, rules, etc. into their Python equivalents (or any other programming language).

Error handling

By default ANTLR checks the input against the grammar syntax. If there are any inconsistencies, the corresponding error information will be printed in the console and the string will be returned as it was recognised by ANTLR. If you remove the colon from the `{x:2}` source string, ANTLR will replace the unrecognised nodes with `undefined`.

```
Alenas-MacBook-Pro:js-runtime alenakhineika$ npm start

> js-runtime@1.0.0 start /Users/alenakhineika/www/js-runtime
> node index.js

line 1:4 no viable alternative at input 'x 2}'
JavaScript input:
{x 2}
Python output:
{undefinedundefined}
Alenas-MacBook-Pro:js-runtime alenakhineika$
```

We can change this behaviour so that error details will be printed instead of a bad string. First, create in the application *root* directory a new module responsible for custom error type generation.

```
$ mkdir error
$ cd error
$ touch helper.js
$ touch config.json
```

I will not elaborate on the module implementation, since it is beyond the scope of the topic. You can copy the ready code below, or write your own variant, more suitable for your particular application infrastructure.

All error types to be used in the application are specified in *config.json*.

Then *error.js* iterates through the list from the config and for each entry it creates a separate class, inherited from `Error`.

Let's update the `visitNumberExpression` method. Now instead of an `Error` text message it will generate `SemanticArgumentCountMismatchError`, which is easier to catch and distinguish a successful result from an error.

Next, let's handle errors directly related to ANTLR, namely those that appear during code parsing. Create a new *ErrorListener.js* file in the *codegeneration* directory with the class inherited from `` `antlr4.error.ErrorListener` ``.

To override the standard error output method we will use two methods available to the ANTLR parser:

- **parser.removeErrorListeners()**, which removes the standard ConsoleErrorListener;
- **parser.addErrorListener()**, which adds a custom ErrorListener.

This must be done after creating the parser, but before calling it. The full code of the updated *index.js* will look like this:

Now an error information is more detailed and useful. And we need just to decide how to handle the occurred exception: interrupt or continue the program execution, save the output info to application log or write tests that support correct and incorrect compiler source data (which is also important).

```
Alenas-MacBook-Pro:js-runtime alenakhineika$ npm start

> js-runtime@1.0.0 start /Users/alenakhineika/www/js-runtime
> node index.js

JavaScript input:
{x 2}
Python output:
{ Error: no viable alternative at input 'x 2}'
    at ErrorListener.syntaxError (/Users/alenakhineika/www/js-runtime/codegeneration/ErrorListener.js:23:11)
    at /Users/alenakhineika/www/js-runtime/node_modules/antlr4/error/ErrorListener.js:69:40
    at Array.map (native)
    at ProxyErrorListener.syntaxError (/Users/alenakhineika/www/js-runtime/node_modules/antlr4/error/ErrorListener.js:69:20)
    at ECMAScriptParser.Parser.notifyErrorListeners (/Users/alenakhineika/www/js-runtime/node_modules/antlr4/Parser.js:352:11)
    at DefaultErrorStrategy.reportNoViableAlternative (/Users/alenakhineika/www/js-runtime/node_modules/antlr4/error/ErrorStrategy.js:276:16)
    at DefaultErrorStrategy.reportError (/Users/alenakhineika/www/js-runtime/node_modules/antlr4/error/ErrorStrategy.js:137:14)
    at ECMAScriptParser.propertyAssignment (/Users/alenakhineika/www/js-runtime/lib/ECMAScriptParser.js:4616:27)
    at ECMAScriptParser.propertyNameAndValueList (/Users/alenakhineika/www/js-runtime/lib/ECMAScriptParser.js:4379:14)
    at ECMAScriptParser.objectLiteral (/Users/alenakhineika/www/js-runtime/lib/ECMAScriptParser.js:4292:18)
  code: 'E_SYNTAX_GENERIC',
  payload:
   { line: 1,
     column: 4,
     message: 'no viable alternative at input \'x 2}\'' } }
Alenas-MacBook-Pro:js-runtime alenakhineika$
```

Conclusion

If you are asked to write a compiler, don't hesitate to accept! It can be very interesting and probably significantly different from your routine programming tasks. We were working in this article only with the simplest nodes to get basic understanding of JavaScript compiler writing process using ANTLR. You can extend the functionality and add an input argument type validation, Extended JSON or BSON support to the grammar, apply a symbol table to enable recognition of such methods as toJSON(), toString(), getTimestamp(), etc. Actually, the possibilities are infinite.

At the time of writing this article, work on the MongoDB compiler is still in progress. It is likely that this code transformation approach will be changed over time, or more optimal solutions will appear. Then I will probably write a new article with more up-to-date information.

Now I'm very into writing a compiler and I would like to share the knowledge gained, as it may be helpful to someone else.

If you want to get deeper into this subject, I recommend the following resources for further reading:

- The language of languages by Matt Might;
- Compiler Construction by Niklaus Wirth;

- Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman;
- Tree structures processing and unified AST by Ivan Kochurkin, Positive Technologies;

Links to the references used:

- Super Tiny Compiler by James Kyle;
- Implementing a Simple Compiler on 25 Lines of JavaScript by Minko Gechev;
- How to write a simple interpreter in JavaScript by Peter Olson;
- Parsing in JavaScript: Tools and Libraries by Gabriele Tomassetti;
- The ANTLR Mega Tutorial by Gabriele Tomassetti;
- Antlr4 — Visitor vs Listener Pattern by Saumitra Srivastav.

Thanks to my Compass team and particularly Anna Herlihy for mentoring and contributing to the compiler writing, Alex Komyagin and Misha Tyulenev for article reviewing and recommendations on the structure, and Oksana Nalivaiko for the title artwork design.

Java Script Compilers Antlr Code Generation Parsing