Master Thesis

Syntactic language extensions for ECMAScript

Author:
Matthisk Heimensen
m@tthisk.nl

Supervisor:
dr. Tijs van der Storm
storm@cwi.nl

June 2015

Host organization: Centrum Wiskunde & Informatica http://www.cwi.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

UNIVERSITEIT VAN AMSTERDAM

Abstract

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Master of Science

Syntactic language extensions for ECMAScript

by Matthisk HEIMENSEN

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

Chapter 1

Introduction and Motivation

1.1 Context

JavaScript is an implementation of the ECMAScript specification. Since the release of the first version of the specification there have been four iterations (where version four was skipped), the current version of the specification is ECMAScript 5. In the near future a new version of the specification will leave the draft status and become the standard (ECMAScript 6). With a new specification come new syntactic language features. Before these features can be used in the JavaScript run-time environment of choice, the vendors of these environments have to implement the new standard. However it is possible to use the new standard today through the use of program transformations.

In this work we study the extension of programming languages, the new ECMAScript specification presents an opportunity to research these extensions and there transformations. Using the RASCAL meta-programming environment[1] we will implement the program transformations. Afterwards we can perform an analysis on the resulting transformation suite to uncover the *effectiveness* of our implementation.

"Program transformations find ubiquitous application in compiler construction to realize desugaring, optimizes, and code generators"[2] in our research we will focus on the desugaring of (new) language constructs.

"The aim of program transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher-level of abstraction, and increasing maintainability"[3]

1.2 Problem definition

Program transformations are faced by multiple problems, in what order do we run the transformation rules, how do we represent the program (e.g. abstract syntax) to perform transformations on, what guarantee do we have for the validity of our transformations, how can a transformation introduce new bindings. Some of these problems reoccur in every transformation that is created. These problems are identified as cross-cutting concerns. The goal of this thesis is to identify these concerns and separate them from the transformation code.

The capture avoidance is one such problem, it is mostly studied in the context of macro expansion[4–6]. And is often called the hygiene of transformations. When an identifier originating from the source program references to a synthesized declaration, the transformation is sub-hygienic and introduces variable capture. To prevent this problem we implement the *name-fix* algorithm[2] for our transformation suite.

1.3 Scope

1.3.1 Expected results

This thesis will have the following results:

- A taxonomy of ES6 language features specified by their needed transformations
- Language extension suite implementing (a subset of) the ES6 features, which is less verbose than current implementations (i.e. BabelJS[1] and Traceur[2]).
- Identification of cross-cutting concerns of our language extension suite
- A set of compatibility tests, testing our implementation against the specification document.
- Eclipse integration (syntax highlighting & declared at hyperlinks) for new language features

1.4 Research questions

The central research questions of this thesis are:

1. How can independent program transformations be categorized?
2. What are the cross-cutting concerns of any program transformation, and how can we take care of these concerns outside of the transformations themselves?
3. What is the advantage of using the RASCAL language workbench[1] for the creation of our transformation suite?

1.5 Outline

In Chapter 2 the background (program transformations, language workbench) is introduced. In Chapter 3 we give an overview of language extensions that are introduced and make a taxonomy of their transformations. In 4 we discuss the cross-cutting concerns of the transformation suite (i.e. variable capture). In Chapter **??** we evaluate our resulting transformation suite (against others). Finally, we conclude in Chapter **??**.

[1]http://babeljs.io
[2]https://github.com/google/traceur-compiler

Chapter 2

Background

2.1 ECMAScript

JavaScript is the programming language implemented from the ECMAScript specification. The specification is maintained by the World wide web consortium (w3c). In this section we will identify some of the key characteristics of the ECMAScript/JavaScript programming language.

Syntax One of the main concerns for program transformations is syntax. The syntax of the programming language is to be extended to support new language features/constructs. ECMAScript falls in the C-type family of syntax definitions. It uses curly braces to delimit blocks of code. A program exists of a list of *statements*. A *statement* can be one of the language control from constructs, a function, or an *expression*. Functions in JavaScript can be either defined as an *expression* or as a *statement*2.1.

Inhertiance & Object orientation JavaScript can be classified as an object oriented language but without *class* based inheritance model[1]. Instead of such a model JavaScript objects have a prototype chain that determine the inheritance. With prototypal inheritance it is possible to implement *class* based inheritance and more.

Everything is an object (with the exception of primitive data-types, e.g. numbers), with properties (even functions 2.1) and one special property called the *prototype*, this prototype references another object (with yet another prototype) until finally one object has prototype *null*. This chain of objects is called the *prototype chain*. When performing a lookup on a object for a certain key x, the JavaScript interpreter will look for property x on our object, if x is not found it will look at the prototype object of our object. This iteration will continue until either prototype *null* is found or an object with property x is found in the prototype chain.

The best analogy to understand the prototypal inheritance as compared to class based inheritance is that of a buildings blueprint. Every class in class based inheritance is the blueprint for a certain thing/object, when such a class is constructed to an object it is build from this blueprint. And when we call a function on this object it is retrieved from

[1]Most popular object-oriented languages do have *class* based inheritance (e.g. Java or C++)

the blueprint. Thus we have an object-class relation. In prototypal inheritance there is no blueprint (or class), but a fully build house (or object). Thus an object-object relation.

Without the use blueprints code will become inefficient rapidly once we start constructing many objects. To avoid this a constructed object has a prototype object which is also a constructed object. The prototype is only constructed once for all the objects that have this object as its prototype.

Scoping Most programming languages have some form of variable binding. These bindings are only bound in a certain context, such a context is often called a scope. There are two different ways to handle scope in a programming language, Lexical (or static) scoping and dynamic scoping. Lexical scoping is determined by the placing of a binding within the source program and its *lexical context*. Variables bound in a dynamic scope are determined through the program state.

In JavaScript bindings are determined through the use of lexical scoping, on a function level. This implies that variables bound within a function's context are available throughout the entire execution of the variable.

One exception on the lexical scoping is the binding of the *this* keyword. Similar to many pure object oriented languages JavaScript makes use of a *this* keyword. In most object oriented languages the current object is implicitly bound to *this* inside functions of that object. [2] JavaScript being object oriented, binds the *this* variable. In JavaScript the value of *this* can change between function calls. For this reason the *this* binding has dynamic scoping (we can not determine the value of *this* before runtime).

Typing JavaScript is a dynamically (duck) typed language without type annotations. Each object can be typecast without the use of special operator(s).

Functions in JavaScript are as explained above also an object. Because of this functions are automatically promoted to first-class citizens (i.e. they can be used anywhere and in anyway normal bindings can be used). This presents the possibility to supply functions as arguments to functions, and calling member functions of the function object. Every function object inherits from the *Function.prototype*[3] which contains functions that allow the overriding of this binding (see 2.1).

Another common pattern is JavaScript related to functions, is that of the immediately invoking function expression (IIFE). Functions in JavaScript can be defined either in a statement or in an expression. When a function is defined as an expression it is possible to immediately invoke the created function:

```
(function() {
  <Statement* body>
})()
```

LISTING 2.1: Immediately invoking function expression

[2]Sometimes *this* is called *self* (e.g. in the Ruby) programming language

[3]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/prototype`

Asynchronous JavaScript is a single-threaded language, this means that concurrency through the use of processes, threads, or other concurrency constructs is impossible. Many operations that have to wait on some form of IO can be asynchronous (non-blocking). Asynchronous programming is a whole subject of its own and falls outside the scope of this thesis.

2.2 Program Transformations

Eelco Visser defines the aim of a program transformation as follows:

“The aim of program transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher-level of abstraction, and increasing maintainability” ([3])

There are many different types of program transformations. Programs can be transformed from a source to a target language. Where both can be the same or different. One can be a higher level language and one a lower level language. In this thesis we will focus on transformations in the category rephrasing[3]. Here a program in a source language is transformed to a different program in the same language. The use case in our thesis is language extensions see section 2.3.

2.3 Language Extensions

”Language extensions augment a base language with additional language features. Many compilers first desugar a source program to a core language.”[2] For example the Haskell functional programming language defines for many constructs how they can be transformed to a kernel language[7]

Language extensions are closely related to other forms of program transformations. There are (syntax) macros[8], these are language extensions defined within the same source file and are expanded before compilation (examples SweeterJS[6], Syntax macros[9]). These ”define syntactic abstractions over code fragments”[10]. Macro systems have several limitations that make it difficult to implement a full language extension set, as needed for the transformation of ES6 features. The transformations need to be present in the same source file, macros are often restrained to a fixed invocation (through a macro identifier), and they often lack the expressiveness in their rewriting language[10]. There was an attempt to implement the ES6 spec through the use of syntax macros but the project[4] is abandoned.

Syntactic sugar is another relative of language extensions. As mentioned before there are several languages that before compilation are transformed to a core language (e.g. Haskell core language). Language features classified as syntactic sugar, are features that can be expressed in the core language. But often benefit from improved human readability when notated with aid of the syntactic sugar. For example the assignment expression $a+=b$ is syntactic sugar notation for $a=a+b$ of frequent occurrence in (c-style) programming languages.

[4]https://github.com/jlongster/es6-macros

2.4 Program Representation

?? Before programs can be transformed they need a structured representation. Seldom are program transformations performed on the merely the textual input. Here we discuss some program representations.

Parse Trees Parse trees represent a programs syntactic information in a tree. This tree includes layout information of the input program (e.g. white-space, comments). The parse tree is structured according to some context-free grammar, that defines how to parse textual input. The parse tree also deals with disambiguation of input and representation of parentheses. If the parser finds an ambiguity during the parse process the two (or more) alternatives are all inserted in the tree under an ambiguity node.

Abstract Syntax Trees or AST is created when layout information is removed from a parse tree and only the core constructs of the language grammar remain as nodes in the tree. White-space layout, comments, and things like parentheses (these are implicitly implied by the structure of the AST) are removed. An AST is often the input for compiler pipelines, program transformations, and refactorings.

Higher Order Syntax Trees (HOAS) To represent not just a program's sub expression relations but also its variable binding we can use a Higher Order Syntax Tree (HOAS)[**?**]. In a HOAS the variable bindings are made explicit (just as the sub expression relations are made explicit in an AST). Every variable has a binding-site and possibly uses throughout the rest of the tree. *"In addition to dealing with the problem of variable capture, HOAS provides higher-order matching which synthesizes new function for higher-order variables. One of the problems of higher-order mathcing is that there can be many matches for a pattern"*[3]

2.5 Parsing

Before a program is represented in on of the previously defined tree formats, it is represented in textual form. To transform the stream of input characters to a tree a program called a parser is used. There are many different types of parser techniques, and a large body of research[11**?** , 12]

The parsing of textual input often happens in two stages. First a *scanner* performs the lexical analysis. Which identifies the tokens of our syntax (e.g. keywords, identifiers). With this identification the parser only has to operate on these identified tokens. "The lexical syntax of a language is usually specified by regular expression"[10].

Scanners that do not have access to the parser, are also unaware of the context of lexical tokens. In case of JavaScript this can impose subtle difficulties in the implementation of a scanner. "due to ambiguities in how regular expression literals (such as /[0-9]*/) and the divide operator (/) should be lexed."[6]. For this reason traditional JavaScript parsers are often intertwined with their lexer. Disney et. al. avoid this problem by introducing a separate reader stage in the parser. However this problem can also be avoided by using scannerless parsing, these parsers preserve the lexical structure of the input characters, and make it possible to compose grammars[11].

2.6 Language Workbenches

The language workbench is a term popularized by Martin Fowler and we can formulate his definition as follows:

A language workbench makes it easy to build tools that match the best of modern IDEs, where the primary source of information is a persistent abstract representation. Its users can freely design new languages without a semantic barrier. With the result of language oriented programming becoming much more accessible.[13]

> Essentially the promise of language workbenches is that they provide the flexibility of external DSLs without a semantic barrier. Furthermore they make it easy to build tools that match the best of modern IDEs. The result makes language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many. ([13])

These workbenches reduce the awkwardness of language oriented programming[14] significantly, by giving programmers the the tools to define programming languages with an abstract representation.

The RASCAL[1] metaprogramming environment (mpl) is a language workbench. It allows programmers to define context-free grammars, generate parsers for these grammars. With these parsers programmers are able to define parse tree patterns. The RASCAL mpl uses Eclipse for IDE integration, which allows programmers to define interaction handles and syntax highlighting. All language extensions presented in this thesis are implemented using the RASCAL mpl.

Syntax definition ....

Concrete syntax A Rascal generated parser returns a parse tree, which is an ordered, rooted tree that represents the syntactic structure of a string according to the formal grammar used to specify the parser. Most transformation systems would implode this concrete syntax tree to an AST and perform program transformations on this tree. Rascal gives the possibility to perform program transformation directly on the concrete syntax tree through the use of concrete-syntax patterns. Presenting multiple advantages over an AST based solution:

- Preserving layout information encoded in the concrete-syntax tree
- Avoiding the use of a pretty printer to output the transformed AST to a textual representation

- Transformation code reads as rewrite rules, instead of a clutter of AST nodes. (Concrete syntax is syntax highlighted using the generated parser)

A concrete-syntax pattern may contain variables in the form of our grammars non-terminals, these variables are bound in the context of the current patterns use (e.g. functions body when used as a formal parameter). The following pattern matches un-named JavaScript functions according to the Saner JavaScript syntax definition[5]

```
(Expression)`function( <Params ps> ) { <Statement* body> }` := pt
```

In similar fashion new syntax trees can be constructed, using the variables bound from pattern matches.

```
(Expression)`function plus(x, y) { return x + y; }`
```

Rascal also supplies a shorthand notation to parse a string starting from the supplied non-terminal

```
[Expression]"1 + 2"
```

[5]https://goo.gl/B6HR22

Chapter 3

Transforming ECMAScript

3.1 Motivating Examples

The Babeljs compiler visits each AST node, functions inside of a specific transformation can announce themselves for the callback stack of a node visit. The callback to this function receives several arguments (current AST node, parent AST node, current Scope, and the file being transformed). The Scope class has functions presented in 3.1, if a node transformation wants to introduce a new variable say ref it calls $generateUidIdentifier$ with String ref as an argument on Scope and receives a name which is not bound in the current Scope. For this to work the visitor needs to have information of all the variables bound in the current scope. This makes the transformation more a full compiler than a set of transformation rules. For starters the transformation code needs to be aware of the Scope class and its functions.

```
generateUidIdentifier(name: string) {
  return t.identifier(this.generateUid(name));
}

generateUid(name: string) {
  name = t.toIdentifier(name).replace(/^_+/, "");

  var uid;
  var i = 0;
  do {
    uid = this._generateUid(name, i);
    i++;
  } while (this.hasBinding(uid) || this.hasGlobal(uid) || this.
 hasReference(uid));


  var program = this.getProgramParent();
  program.references[uid] = true;
  program.uids[uid] = true;

  return uid;
}

_generateUid(name, i) {
  var id = name;
  if (i > 1) id += i;
  return `_${id}`;
  }
```

Listing 3.1: Variable capture avoidance code from babeljs source[1]

The Traceur compiler totally ignores the problem of variable capture. In conformance with the case study conducted by Erdeweg et. al. [2] we were able to identify a transpiler that fails to address variable capture. With a simple example we can demonstrate the introduction of capture by Traceur 3.2

```
function f() {
  var $__0 = 0;
  var x = () => this;
}
```

Listing 3.2: Example input to Traceur[2]

```
function f() {
    var $__0 = this;
    var $__0 = 0;
    var x = (function() {
      return $__0;
    });
  }
```

Listing 3.3: Variable capture

3.2 Taxonomy of language features

Every language extension has several properties which can be identified and categorized along certain dimensions. In this chapter we present such a taxonomy for all the language extensions presented in the ECMAScript 6 specification[15].

3.2.1 Dimensions

The following dimensions are identified and used to categorize every language extension.

Category One of the rephrasing categories defined by Eelco Visser[3], rephrasings are program transformations that transform, where the source and target program language are the same. We identify the following categories:

Abstraction level Program transformations can be categorized by their abstraction level. There are four levels of abstraction (similar to those of macro expansions[9]), character-, token-, syntax-, or semantic-based. Character and token based transformations work on a program in textual representation. Syntactical transformations work on a program in its parsed representation (either as an AST or as a parse tree **??**). Next to the syntactic representation semantic transformations also have access to the static semantics of the input program (e.g. variable binding).

[1]https://goo.gl/BZKIvV
[2]https://goo.gl/Ds3xUn

TABLE 3.1: Taxonomy categories

Normalization	
Desugaring	is a form of normalization. With a desugaring language constructs (called syntactic sugar) are reduced to a core language.
Simplification	this is a more generic form of normalization in which parts of the program are transformed to a standard form (e.g. removing an if statement which always evaluates to false)
Weaving	
Optimization	These transformations help improve the run-time and/or space performance of a program
Specialization	
Inlining	
Fusion	
Other	
Refactoring	"is a disciplined technique for restructuring an existing body of code, alteringits internal structure without changing its external behaviour."[3]
Obfuscation	is a transformation that makes output code less (human) readable, while not changing any of the semantics.
Renovation	is a special form of refactoring, "to repair an error or to bring it up to date with respect to changed requirements"[3]

Purely semantic Is the extension purely in semantic, or does the extension introduce new syntax.

Scope Program transformations can happen within four different scopes:

When a program transformation matches on a sub-tree of the parse tree and only transforms this matched sub-tree it is a local-to-local transformation. If the transformation needs information outside the context of the matched sub-tree, but only transforms the matched sub-tree it is global-to-local. When a transformation has no additional context from its local sub-tree but does alter the entire parse tree it is called local-to-global.If the transformation transforms the input program in its entirety it is global-to-global.

Syntactically type preserving Program transformations performed on syntax elements (i.e. being syntax- or semantic based) can preserve the syntax type of their input element or alter it (e.g. is an expression transformed to an expression, or to a list of statements)

Introduction of bindings Are new bindings introduced in the transformed code as opposed to the original code.

Depending on bindings (i.e. run-time code) Will the transformed code rely on function calls not introduced by the transformation itself but provided separately from the transformation suite.

Compositional When a program transformation does not alter the containing context of the matched node, it is said to be compositional.

Preconditions What are the preconditions that have to be met before execution of a transformation rule, to ensure validness of our transformation (e.g. all sub-terms have to be analyzed and transformed)

Restrictions on sub-terms Does the language extensions impose restrictions on the sub-terms used inside of the language extension.

Analysis of sub-terms Are the non-terminals of our language extension analyzed and possibly transformed by the transformation rule.

Dependency on other extensions Can the language extensions be performed standalone or is there a dependency on one of the other extensions.

Backwards compatible Is the API of the transformed code compatible with the ECMAScript 6 specification (i.e. can we import a transformed module in ECMAScript 6 and use it properly).

Dividable Is it possible to identify smaller transformation rules inside this language extension, that can be performed independently from one another.

3.2.2 On the expressive power of ECMAScript 6

Matthias Felleisen presents a rigorous mathematical framework to define the expressiveness of programming languages (as compared to each other) in his paper "On the expressive power of programming languages"[16].

The formal specification presented by Felleisen falls outside of the scope of this thesis. But we can summaries his findings informally: If a construct in language A can be expressed in a construct in language B by only performing local transformations (local-to-local). Language A is no more expressive than language B. And *"By the definition of an expressible construct, programs in a less expressive language generally have a globally different structure from functionally equivalent programs in a more expressive language."*[16]

"Intuitively, a programming language is less expressive than another if the latter can express all the facilities the former can express in the language universe."[16]

How is this applied to the ECMAScript 6 standard. Every new language feature can be transformed to a construct with similar semantics in ECMAScript 5 with a local-to-local transformation. With the exception of the let-constB.0.0.12 extension.

3.3 Implementation

Each transformation is defined as a term-rewriting rule. It has a concrete syntax pattern which can match part of a parse-tree. The result is a concrete piece of syntax, using only constructs from the fundamental syntax definition (i.e. ECMAScript 5). The rewrite rules are exhaustively applied on the input parse-tree until no more rewrite rules match any sub-trees of the input (constraint solving). Application of rewrite rules to the parse tree is done bottom-up, because several rewrite rules (e.g. B.1) demand for successful completion that their sub-terms are already transformed.

3.3.1 cross-cutting concerns

Each transformation rule has to deal with similar issues, can we identify these issues and solve them in a standalone (language agnostic) fashion? Many problems with transformations originate with name binding in source and target language.

Table 3.2: ES6 features transformation dimensions (1/2)

	Arrow Functions	**Classes**	**Destructuring**	**Object literals**	**For of loop**	**Spread operator**
Category	D.	D.	D.	D.	D.	D.
Abstraction level	CfS.	CfS.	CfS.	CfS.	CfS.	CfS.
Scope	L2L	L2L	L2L	L2L	L2L	L2L
Syntactically type preserving	●	●	○	●	●	●
Introducing bindings	●	○	●	○	○	●
Depending on bindings	○	●	●	○	○	●
Compositional	●	●	●	●	●	●
Analysis of subterms	●	●	●	○	○	●
Constraints on subterms	○	●	○	○	○	○
Preconditions	●	●	○	○	○	○
Dependencies	○	○	●	○	●	○
Backwards compatible	●	●	●	●	●	●
Dividable	○	○	●	●	●	●

	Default parameters	**Rest parameters**	**Template Literals**	**Generators**	**Let Const**
Category	D.	D.	D.	D.	U.
Abstraction level	CfS.	CfS.	CfS.	CfS.	S.
Scope	L2L	L2L	L2L	L2L	L2L
Syntactically type preserving	●	●	●	●	●
Introducing bindings	○	○	○	○	○
Depending on bindings	○	○	○	●	○
Compositional	●	●	●	●	○
Analysis of subterms	○	○	○	●	●
Constraints on subterms	○	○	○	○	○
Preconditions	○	○	○	●	●
Dependencies	○	○	○	○	○
Backwards compatible	●	●	●	●	○
Dividable	○	○	○	○	●

Variable capture What happens when a transformation rule introduces a new binding. There are two possibilities either the binding-name is not yet bound in the source program, or the binding-name is already bound in the source program. In the second case our transformation code introduces *variable capture*. This can be illustrated with the use of a simple example.

```
var {x} = obj;
```

Listing 3.4: Input JavaScript

```
var _ref = obj;
var x = _ref.x;
```

Listing 3.5: Desugared JavaScript

For this example we look at destructuring variable declaration (see: B.0.0.3). During the transformation a new binding is created to store the object (named _ref).

If however the name _ref was already bound in our source program this would result in an illegal redeclaration of _ref. This problem is identified as variable capture and can arise for any piece of transformation code, were source and target language have name binding.

What are some possible solutions for this problem. Let each transformation deal with the avoidance of variable capture. Build full scope tree and supply this scope to our *desugaring* functions, generate new identifiers that are unique in the scope. Disadvantage of this solution is that programmers creating transformation code need also be capable of creating scope trees, and our transformer start to have more resemblance with full fledged compiler than term-rewriting rules.

The solution presented by Erdweg et. al.[17] is called *name-fix*, this is a standalone algorithm that does not interfere with transformation code. It identifies variable capture through the use of string origin[18] information stored in the parse tree.

Introducing (multiple) bindings As described above various desugarings have to introduce bindings for correct transformation. In most cases new bindings will be bound to some descriptive variable name (e.g. _ref). If a language extension is used multiple times within the same scope the same identifier will be declared multiple times. This is a similar problem to that of variable capture, but the capture now happens with binding all introduced by transformation code. And not from a transformation to source binding. Similar to the variable capture problem this problem could be solved uniquely for every transformation, or can be extracted and solved in a generic manner (i.e. agnostic from transformation code).

Supply each *desugar* function with a function called *generateUId* (generate unique identifier). When a transformation wants to generate a identifier (.e.g _ref), it calls this function (which has knowledge of all previously generated identifiers) and generates a unique identifier through the use of *gensym*?? algorithm.

Introducing new bindings Many transformations introduce new bindings to store some intermediate value needed for our transformation. These bindings should be introduced in the containing block-scope where our transformation is performed. With the use of annotations on the parse tree we can set declarations on our current matched node. A final pass over the tree will take care of setting declarations in the correct block scope.

Chapter 4

Results and Comparison

4.1 Implementation

TABLE 4.1: Lines of code - transformations

	Files	Lines of code
Babel	76	6547
Traceur	19	9881
Rascal	62	1368

TABLE 4.2: Lines of code - parser

	Files	Lines of code
Babel	22	3583
Traceur	15	6681
Rascal	12	555

Compatibility tests[1]

TABLE 4.3: Compatibility tests

	Total	Babel	Traceur	Rascal
Syntax	76	76	60	58
Bindings	19	15	15	12
Functions	62	54	50	34
Total	157	136 (87%)	125 (80%)	98 (62%)

[1] https://kangax.github.io/compat-table/es6/

Appendix A

Artifact Description

Summary We provide implementation of ECMAScript 6 language features syntax, and transformation rules in the Rascal meta-programming language (http://rascal-mpl.org). We use rascal's built-in support for syntax definition and parsing.

Content The code for language extensions is stored in *src/extensions* here we will discuss implementation status for each language feature:

Function extensions
Arrow functions (*src/extensions/arrow*)
Full functionality* (i.e. basic functionality and correct *this*,*arguments* binding)

Class declaration (*src/extensions/classes*)
Full functionality.* (i.e. basic support, support for extension, getter/setter/static methods, methods are non-enumerable)

Super (*src/extensions/classes*)
Full functionality.*

Generators (*src/extensions/generators*)
No support. Due to the time constraint nature of the project this feature has not been implemented.

Syntax extensions
Destructuring (*src/extensions/destructuring*)
Near full functionality. There is no support for unbound match:

```
var [,b] = [1,2]; // b == 2
```

This feature is not implemented because syntax definition for empty elements in a comma separated list fairly complex. Making the resulting concrete syntax even more complex. To avoid unneeded complexity we decided to refrain from implementation of this feature.

For of loops (*src/extensions/forof*)
Full functionality.

Binary & Octal literals (*src/extensions/literal*)
Full functionality.

Object literal (*src/extensions/object*)
Full functionality.

Rest Parameters (*src/extensions/parameters*)
Near full functionality. (i.e. no support for arguments object interaction outside of strict mode).

Default (*src/extensions/parameters*)
Basic functionality.

Spread operator (*src/extensions/spread*)
Near full functionality. (i.e. no support for spread operator with generators, because we do not implement generators)

Template Literals (*src/extensions/template*) Full functionality (i.e. tagged template literals and normal template literals)

Binding extensions
Let (*src/extensions/letconst*)
Full functionality. (i.e. support for renaming of function scoped clashes of block scoped names, no reference possible before definition, and possibility to throw run-time errors on redeclaration)

Const (*src/extensions/letconst*)
Full functionality. (i.e. same as let)

Modules (*src/extensions/modules*) No support.

* These features make use of the ES6 *new.target* property. This newly introduced property is an implicit parameters. Implicit parameters are those parameters that are set during every function executed but not explicitly passed by the caller (e.g. *arguments*). It is used to refer to the constructor function when a function is invoked with the use of the *new* keyword. Our implementation does not support the *new.target* property. It is possible to parse a new.target references but we have not implemented any desugaring code to transform the reference to ES5 code. The dynamic behavior of *new.target* makes the transformation complex to perform during a static (i.e. non-runtime) phase. A transformation would be global-to-global (see:B) (every function call has to be analyzed in the entire program).

The compatibility tests as discussed in **??** are stored in *input/compatibility* and can be invoked from the Rascal module at *src/test/Compatibility.rsc*

Our implementation of the *name-fix*[17] algorithm resides in *src/core/resolver*.

Getting the artifact The latest version of the artifact is available at `https://github.com/matthisk/rascal-sweetjs` as a git repository. The repository page includes information on how to run the artifact inside the Eclipse IDE[1].

[1]`http://eclipse.org`

Appendix B

Language feature categorization

B.0.0.1 Arrow Functions

Arrow functions[15, 14.2] are the new lambda-like function definitions inspired by Coffeescript and C# notation. The functions body knows no lexical this binding but instead uses the binding of its parent lexical scope.

TABLE B.1: Extension transformation dimensions

	Arrow Functions
Category	D.
Transformation level	CfS.
Scope	Global-to-local
Syntactically type preserving	●
Introducing bindings	●
Depending on bindings	○
Compositional	●
Analysis of sub-terms	●
Constraints on sub-terms	●
Preconditions	●
Dependencies	○
Backwards compatible	●
Dividable	○

Category The arrow function construct is eliminated by translating them into the fundamental function construct. Thus we speak of a syntactic sugar and a desugaring.

Scope To avoid a ReferenceError when an arrow function is used outside of the lexical scope of a function, the $_arguments$ identifier should reference $undefined$. But only when the arrow is placed outside of any functions lexical scope. Thus the transformation needs context information of the placement of an arrow function and the transformations scope is global-to-local.

Analysis of sub-terms References to the arguments, sup er and this keyword have to be identified in the ConciseBody of an ArrowFunction and renamed.

"An ArrowFunction does not define local bindings for **arguments**, **super**, **this**, or new.target. Any reference to arguments, super, or this within an ArrowFunction must resolve to a binding in a lexically enclosing environment." ([15, 14.2.16])

An arrow function is transformed to an ES5 function which is wrapped in a self-calling function which receives its lexical enclosing scopes **super**, **this**, and **arguments** variables. The enclosing lexical scope is not polluted with new variable declarations. Keywords references are prepended with an underscore, only references in the current scope (i.e. no deeper scope) are renamed.

```
var f = <Arrow Function>;
```

```
var f = (function(_super,_arguments,_this) {
  return <Desugared Arrow Function>
})(super,arguments,this);
```

Constraints on sub-terms Use of the **yield** keyword is not allowed in an arrow functions ConciseBody. Arrow functions can not be generators and deep continuations are avoided.

Preconditions Before an arrow function can be transformed its ConciseBody should have been transformed. This to prevent the incorrect renaming of sub-terms (**super**,**arguments**,**this**), because they still fall in the same scope depth.

```
var f = () => {
  () => this;
};
```

```
var f = (function(_this,_arguments) {
  return function() {
    () => _this;
  };
})(this,arguments);
```

Listing B.1: Incorrect renaming of this in nested arrow function

B.0.0.2 Classes

Class definitions[15, 14.5] are introduced in ECMAScript 6 as a new feature to standardize inheritance model. Underneath the prototypal inheritance model is still used to create class declarations.

Category The class construct is syntactic sugar for simulation of class based inheritance through the more fundamental prototypal inheritance system of the ECMAScript language.

Table B.2: Extension transformation dimensions

	Classes
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	●
Compositional	●
Analysis of sub-terms	●
Constraints on sub-terms	●
Preconditions	●
Dependencies	○
Backwards compatible	●
Dividable	○

Syntactically type preserving There are two types of class declarations, a direct declaration as a Statement, or an Expression declaration. They are both transformed to a construct of the same syntactical type.

Depending on bindings Several features of the class declaration as described in the specification demand some run-time. (e.g. the class methods being non enumerable)

Analysis of sub-terms References of **super** and a **super** call have to identified inside constructor or class methods. These are transformed to preserve the correct **this** binding.

Constraints on sub-terms The sub-terms of a class declaration are always executed in strict mode

"A ClassBody is always strict code." ([15, 14.5])

Preconditions Bodies of constructor and class methods should have been transformed before the class declaration itself is transformed. This to prevent incorrect transformation of the sub-term **super**.

B.0.0.3 Destructuring

Destructuring[15, 12.14.5] is a new language construct to extract values from object or arrays with a single assignment. It can be used in multiple places among which parameters, variable declaration, and expression assignment.

Table B.3: Extension transformation dimensions

	Destructuring
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	○
Introducing bindings	●
Depending on bindings	●
Compositional	●
Analysis of sub-terms	●
Constraints on sub-terms	○
Preconditions	○
Dependencies	●
Backwards compatible	●
Dividable	●

Category The destructuring language feature is eliminated by translating it into fundamental language concepts such as member access on objects and array member selection.

Syntactically type preserving The transformation is not syntactically type preserving in every situation. If the destructuring is used in a variable declaration the syntactic type is converted from $< Statement >$ to $< Statement* >$, because new bindings are introduced.

Dependencies Object destructuring supports the computed property notation of extended object literals (as discussed in section: B.0.0.4):

```
var qux = "key";
var { [qux] : a } = obj;
```

Destructuring thus has a dependency on extended object literal notation for this feature to work properly.

Dividable This language features consists of two types of destructuring, object destructurings, and array destructurings. These two different features can be transformed separately.

B.0.0.4 Extended object literals

Literal object notation receives three new features in the ECMAScript 6 standard[15, 12.2.5]. Shorthand property notation, shorthand method notation, and computed property names.

Table B.4: Extension transformation dimensions

	Object literals
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	○
Compositional	●
Analysis of sub-terms	○
Constraints on sub-terms	○
Preconditions	○
Dependencies	○
Backwards compatible	●
Dividable	●

Category The extended object literals are syntactic sugar in the purest form. There are direct rules to eliminate the introduced concepts to fundamental concepts of the ECMAScript 5 language. (e.g. shorthand property notation translate to non-shorthand notation)

Introducing bindings For full spec compliance run-time code has to be introduced to set computed property keys. However this can also be done through the member notation:

```
var obj = { [qux] : 0; }
```

```
var _obj;
var obj = ( _obj = {}, _obj[qux] = 0, _obj );
```

Dividable All three extensions of the object literal can be transformed separately.

B.0.0.5 For of loop

The ECMAScript 6 standard introduces iterators, and a shorthand for notation to loop over these iterators in the form of for of[15, 13.6.4]. Previous versions of the ECMAScript standard had default for loops, and for in loops (which iterate over all enumerable properties of an object).

Category This construct can be eliminated by transformation to a for-loop using an iterator variable, and selecting the bound variable for the current index from the array using this iterator variable.

Table B.5: Extension transformation dimensions

	For of loop
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	•
Introducing bindings	◦
Depending on bindings	◦
Compositional	•
Analysis of sub-terms	◦
Constraints on sub-terms	◦
Preconditions	◦
Dependencies	•
Backwards compatible	•
Dividable	•

Constraints on sub-terms No constraints any $< Statement >$ can be used within the body of a for-of loop.

```
syntax Statement
  = "for" "(" Declarator ForBinding "of" Expression ")" Statement;
```

Dependencies The for-of loop binding can be declared with the let declaratorB.0.0.12.

```
for( let x of arr ) <Statement>
```

B.0.0.6 Spread operator

ECMAScript 6 introduces a new unary operator named spread[15, 12.3.6.1]. This operator is used to expand an expression in places where multiple arguments (i.e. function calls) or multiple elements (i.e. array literals) are expected.

Category This unary operator can be expressed using fundamental concepts of the ECMAScript 5 language. In case the operator appears in a place where multiple arguments are expected, a call of member function *apply* on our function can be used to supply arguments as an array. In case it appears in a place where multiple elements are expected the prototype function *concat* of arrays can be used to interleave the supplied argument to the operator with the rest of the array.

Introducing bindings A binding has to be introduced to save the scope in which a function is applied, when the function call happens on an objectB.0.0.6

```
a.b.f(...args);
```

```
var _a$b;
(_a$b = a.b).f.apply(_a$b, args);
```

Table B.6: Extension transformation dimensions

	Spread operator
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	●
Depending on bindings	●
Compositional	●
Analysis of sub-terms	●
Constraints on sub-terms	○
Preconditions	○
Dependencies	○
Backwards compatible	●
Dividable	●

Depending on bindings For correct specification compliance the argument of the spread operator has to be transformed to a consumable array. This behavior can be simulated through a run-time function.

Analysis of sub-terms If there exists a function call of function f on object a where spread operator is used in arguments list. This function call has to be transformed to a call to function *apply* on function f on object a. Where the first parameter of *apply* is a and second is the argument of spread. The sub-terms of function calls on objects with spread operators thus have to be analyzed to determine the call scope of the called function.

```
a.f(...args);
```

```
a.f.apply(a,args);
```

Listing B.2: apply function with correct *this* scope

Dividable The operator can be used in two places (places of multiple arguments, and multiple elements), which can be transformed separately.

B.0.0.7 Default parameter

The ECMAScript 6 spec defines a way to give parameters default values[15, 9.2.12]. These values are used if the caller does not supply any value on the position of this argument. Any default value is evaluated in the scope of the function (i.e. *this* will resolve to the run-time context of the function the default parameter value is defined on).

Table B.7: Extension transformation dimensions

	Default parameters
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	○
Compositional	●
Analysis of sub-terms	○
Constraints on sub-terms	○
Preconditions	○
Dependencies	○
Backwards compatible	●
Dividable	○

Category This language feature can be eliminated by transformation to a fundamental concept of the ECMAScript 5 language. By binding the variable in the function body to either the default value (in case the parameter equals $undefined$) or to the value supplied by the parameter.

Introducing bindings This transformation does not introduce any bindings because the bindings already exist as formal parameters.

B.0.0.8 Rest parameter

The *BindingRestElement* defines a special parameter which binds to the remainder of the arguments supplied by the caller of the function.[15, 14.1]

Table B.8: Extension transformation dimensions

	Rest Parameter
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	○
Compositional	●
Analysis of sub-terms	○
Constraints on sub-terms	○
Preconditions	○
Dependencies	○
Backwards compatible	●
Dividable	○

Category

Syntactically type preserving $< Function >$ to $< Function >$

B.0.0.9 Template strings

Standard string literals in JavaScript have some limitations. The ECMAScript 6 specification introduces a template string literal to overcome some of these[15, 12.2.8]. Template string literals are delimited by the ‘ quotation mark, can span multiple lines and be interpolated by expressions: $\{< Expression >\}

Table B.9: Extension transformation dimensions

	Template Strings
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	○
Compositional	●
Analysis of sub-terms	○
Constraints on sub-terms	○
Preconditions	○
Dependencies	○
Backwards compatible	●
Dividable	○

Category Template strings are syntactic sugar for Strings concatenated with expressions and new-line characters.

Syntactically type preserving $< TemplateLiteral >$ to $< StringLiteral >$

B.0.0.10 Tail call optimization

JavaScript knows no optimization for recursive calls in the tail of a function, this results in the stack increasing with each new recursive call overflowing the stack as a result. The ECMAScript 6 specification introduces tail call optimization to overcome this problem[15, 14.6]

Category Optimization, this transformation improves the space performance of a program.

Abstraction level Tail call optimization is a semantic based transformation. To identify if the expression of a return statement is a recursive call to the current function we need name binding information.

Pure semantics This extension is purely semantic, no new syntax is introduced. To transform this extension to work in older JavaScript interpreters the tail call has to be removed and replaced by a *while* loop, to prevent the call stack from overflowing.

```
function fib(n,nn,res) {
  if( nn > 100 ) return res;
  return fib(nn,n + nn,res + [n,nn]);
}
```

Listing B.3: Function with tail recursion

```
function fib(arg0,arg1,arg2) {
  while (true) {
    var n = arg0, nn = arg1, res = arg2;

    if(nn > 100) return res;
    arg0 = nn;
    arg1 = n + nn;
    arg2 = res + [n,nn];
  }
}
```

Listing B.4: Semantically identical function, without tail recursion

Figure B.4 illustrates how a function with tail recursion can be transformed to a function with an infinite loop. Not filling the call stack with new return positions and thus executing correctly even when our upper limit (100) is increased.

B.0.0.11 Generators

The ECMAScript 6 standard introduces a new function type, generators[15, 14.4]. These are functions that can be exited and later re-entered, when leaving the function through *yielding* their context (i.e. variable bindings) is saved and restored upon re-entering the function. A generator function is declared through `function*` and returns a Generator object (which inherits from *Iterator*)

Category A generator can be implemented as a state machine function wrapped in a little run-time returning a Generator object. It does categorize as syntactic sugar, however the transformation is complex (because of identification of control flow of body statement, see Analysis of sub-terms).

Scope Including a run-time the transformation can be managed local-to-local.

Syntactically type preserving $< Statement >$ to $< Statement >$ $< Expression >$ to $< Expression >$

Table B.10: Extension transformation dimensions

	Generators
Category	D.
Abstraction level	CfS.
Scope	L2L
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	●
Compositional	●
Analysis of sub-terms	●
Constraints on sub-terms	○
Preconditions	●
Dependencies	○
Backwards compatible	●
Dividable	○

Analysis of sub-terms The control flow of the functions body has to be identified. The control flow graph can be interpreted as a state machine with several leaps, entry points, and exit points that result from the use of different control-flow constructs (e.g. *yielding*, *returning*, *breaking*, etc.). After transformation this state machine can be represented in a *switch* statement. All bindings of the function body have to be hoisted outside of this state machine, and only to be assigned through an assignment expression in the state machine.

Preconditions The $< GeneratorBody >$ has to be transformed before transformation of the Generator function starts.

Depending on bindings The state machine function is wrapped in a run-time function to initialize the Generator object returned from a generator function. The run-time makes sure the state-machine has correct scoping bindings on each entry[1]

B.0.0.12 Let and Const declarators

JavaScript's scoping happens according to the lexical scope of functions. Variable declarations are accessible in the entire scope of the executing function, this means no block scoping. The *let* and *const* declarator of the ECMAScript 6 specification change this. Variables declared with either one of these is lexically scoped to its block and not its executing function. The variables are not hoisted to the top of their block, they can only be used after they are declared. Code before the declaration of a *let* variable is called the temporal dead-zone. *const* declaration are identical to *let* declarations with the exception that reassignment to a *const* variable results in a syntax error. This aids developers in reasoning about the (re)binding of their const variables. Do not confuse this with immutability, members of a *const* declared object can still be altered by code with access to the binding.

[1] `http://github.com/facebook/regenerator/blob/master/runtime.js`

Table B.11: Extension transformation dimensions

	Let Const
Category	Undefined
Abstraction level	Semantic
Scope	Global-to-global
Syntactically type preserving	●
Introducing bindings	○
Depending on bindings	○
Compositional	○
Analysis of sub-terms	●
Constraints on sub-terms	○
Preconditions	●
Dependencies	○
Backwards compatible	○
Dividable	●

Category It is possible to eliminate *let* declarations without losing correct block scoping of bindings. We can illustrate this with the help of a simple example. 0 is bound to x, now we bind 1 to x in a nested block. Outside of this block 1 is still bound to x.

```
let x = 1;

{
  let x = 0;
  x == 0; // true
}

x == 0; // false
```

This behavior can be emulated in ECMAScript 6 with the use of immediately invoking function expressions (or IIFE), the function expression introduces a new function scope. And the rebinding of x is not ‘visible‘ outside of the new IIFE.

```
var x = 1;

(function() {
  var x = 0;
  x == 0; // true
})();

x == 0; // false
```

Thus *let* declarations can be desugared if every block is transformed to a *closure* with the help of IIFE's. This does however imply a serious performance overhead on the transformed program (instead of normal block execution the interpreter has to create a closure for every block). This also breaks the standard control-flow of our program when the block uses any control-flow statement keywords (e.g. *return*).

There is another way to transform *let* declarations, for this implementation we need full name analysis of a program. The previous example can also be transformed by renaming the rebinding of x:

```
1 var x = 1;
2
3 {
4   var x_1 = 0;
5   x_1 == 0; // true
6 }
7
8 x == 0; // false
```

The analysis has access to both the binding graph of the program with function-scoping and *var* declarations, and the binding graph with block scoping and *let* declarations. After the creation of these graphs an illegal reference can be defined as a reference that does appear in the function-scope binding graph, but does not appear in the block-scope binding graph. In the case of our example this means that x references 0 bound to x in block (at line 4) when analyzing function-scope bindings. But x does not reference the same binding when analyzing the block-scope binding graph. [2]

Scope The transformation needs to analyse the entire program (global) and make transformations to its entirety (global). Thus the scope of the transformation is global-to-global.

Compositional The transformation of *let* (or *const*) bindings is not compositional because bindings of these types can not be captured by names in sibling scopes

```
{
  let x = 0;
}
{
  let x = 1;
}
```

Once these *let* bindings are transformed to *var* bindings without renaming they would be declared in the same lexical scope. Whereas they are not at the moment. For a transformation to *var* bindings to work one of these bindings and its uses has to be renamed.

Introducing bindings The transformation introduces no new bindings, it converts *let* and *const* bindings to *var* bindings, and renames these bindings (and their references) where necessary.

Preconditions Other transformations need to be performed before the let const extension is transformed. To prevent this extension to have dependencies on other extensions. For example the for-of extension has a dependency on the let-const extension:

```
for( let x of arr ) <Statement>
```

If the for-of loop is transformed to a normal loop let-const needs not to be aware of the for-of loop as an extension, but just of the ECMAScript 5 manner to declare variables inside loops.

[2]More examples: http://raganwald.com/2015/05/30/de-stijl.html

Dividable Variables declared through let and const can be transformed to ECMAScript 5 code separately.

Bibliography

[1] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL : a Domain Specific Language for Source Code Analysis and Manipulation.

[2] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-Avoiding and Hygienic Program Transformations (incl. Proofs). 8586:1–29, April 2014. URL http://link.springer.com/10.1007/978-3-662-44202-9http://arxiv.org/abs/1404.5770.

[3] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, December 2001. ISSN 15710661. doi: 10.1016/S1571-0661(04)00270-1. URL http://linkinghub.elsevier.com/retrieve/pii/S1571066104002701.

[4] David Herman and Mitchell Wand. A Theory of Typed Hygienic Macros. *Proceedings of the 17th European Symposium on Programming*, 4960:48, 2010. URL http://www.springerlink.com/index/V03K0734Q7217R35.pdf.

[5] David Herman and Mitchell Wand. A theory of hygienic macros. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4960 LNCS, pages 48–62, 2008. ISBN 3540787380. doi: 10.1007/978-3-540-78739-6_4. URL http://www.springerlink.com/index/V03K0734Q7217R35.pdf.

[6] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript. In *Proceedings of the 10th ACM Symposium on Dynamic languages - DLS '14*, pages 35–44, New York, New York, USA, 2014. ACM Press. ISBN 9781450332118. doi: 10.1145/2661088.2661097. URL http://dl.acm.org/citation.cfm?doid=2661088.2661097.

[7] SL Peyton Jones, RJM Hughes, L Augustsson, D Barton, B Boutel, W Burton, J Fasel, K Hammond, R Hinze, P Hudak, T Johnsson, MP Jones, J Launchbury, E Meijer, J Peterson, A Reid, C Runciman, and PL Wadler. Report on the programming language haskell 98, February 1999. URL http://research.microsoft.com/apps/pubs/default.aspx?id=67041.

[8] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, 1966. ISSN 00010782. doi: 10.1145/365876.365879.

[9] Daniel Weise and Roger Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6):156–165, 1993. ISSN 03621340. doi: 10.1145/173262.155105.

[10] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. *Proceedings of the 19th*

ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04, pages 365–383, 2004. ISSN 0362-1340. doi: 10.1145/1035292.1029007.

[11] Eelco Visser. Scannerless Generalized-LR Parsing. (P9707), 1997.

[12] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, 1989. ISSN 03621340. doi: 10.1145/74818.74833.

[13] Martin Fowler. Language workbenches: The killer-app for domain specific languages. *Accessed online from: http://www. martinfowler. com/articles/languageWorkbench. html*, pages 1–27, 2005. doi: http://www.martinfowler.com/articles/languageWorkbench.html. URL http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf.

[14] M P Ward. Language Oriented Programming. *Software-Concepts and Tools*, 15(4): 1–22, 1994.

[15] Ecmascript 2015 language specification. final draft 38, Ecma International, apr 2015.

[16] Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, pages 131 – 151, 1990. ISSN 01676423. doi: 10.1016/0167-6423(91)90036-W.

[17] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. *ACM SIGPLAN Notices*, 49(3):3–12, March 2014. ISSN 03621340. doi: 10.1145/2637365.2517210. URL http://dl.acm.org/citation.cfm?doid=2637365.2517210.

[18] Pablo Inostroza, Tijs van der Storm, and Sebastian Erdweg. Tracing Program Transformations with String Origins. pages 154–169. 2014. doi: 10.1007/978-3-319-08789-4_12. URL http://link.springer.com/10.1007/978-3-319-08789-4_12.