HOCHSCHULE DER MEDIEN

Bachelor Thesis

Building Compiled Language Extensions for JavaScript

Author
Jonathan Immanuel Brachthäuser

Examiners
Prof. Walter Kriha
Prof. Dr. Ansgar Gerlicher
Andreas Stiegler, M. Sc.

Stuttgart Media University (HdM) — February 27, 2012

Integrity Statement

Herein I declare that this Bachelor Thesis was created entirely by myself. I only used the sources specifically stated in this document. Thoughts used, either by meaning or quoted, were marked as such.

Stuttgart, February 27, 2012   Jonathan Immanuel Brachthäuser

Acknowledgements

This thesis would be impossible without the support of some great people.

Marlen, I love you. Thank you for trying to understand what I am doing and especially for learning Lisp! Thank you, beloved family, for always supporting me unconditionally.

Many thanks to Stephan Soller, Andreas Stiegler and Patrick Bader for all those intense and vivid discussions that helped my stay on track. I have truly learned a lot from you guys, even in times I did not want to. Patrick pointed my nose at OMeta which has been a great inspiration for this thesis. By the way, AST guessing is an awesome game to play.

Alessandro Warth did a great work with his language OMeta. Thank you, it is so much fun to work with.

Even if writing a thesis can be rough some time it is less of a burden when working in a comforting environment. Thank you Prof. Zimmermann for your trust and for tilting at the windmills of bureaucracy.

I also have to thank Claus Gittinger for showing me that building Interpreters (and compilers in extension) is no voodoo.

Last but not least I want to thank Prof. Walter Kriha and Prof. Dr. Ansgar Gerlicher for giving me the chance and freedom to do research on such an amazing topic.

Abstract

The web is moving in an enormous speed. New standards like HTML5, CSS3 and the 6th edition of the ECMAScript are under development. Browser vendors have improved the performance of their JavaScript engines a lot which now serve as Petri dishes for the development of new languages and JavaScript derivatives. The term “transpiler” has been coined to subsume the cross-compilation from one edition of ECMAScript to another.

Set in this volatile environment, the goal of this thesis is to provide a foundation to compose individual extensions. Thus, a custom language derivative can be built which fits the individual likings. It is not scope of this thesis to design such a language but to provide a framework supporting the creation of JavaScript language extensions.

This thesis targets at intermediate JavaScript developers who are interested in creating their own language. No expert knowledge in building compilers is required. The thesis is designed as a guide and hence necessary theoretical knowledge of compilers and parser generators is briefly discussed in the first few chapters.

OMeta/JS is chosen as parser generator in oder to experiment with language extensions in the most flexible and convenient way. Both a JavaScript parser and a translator are implemented in a way that allows an easy extension. JsonML is chosen to internally represent the abstract syntax tree. The creation of the tree is accomplished by the means of node-constructors. A generic walker grammar is provided in order to easily traverse the syntax tree.

The design of the framework aims at the reuse of the single components. Hence, the re-factored implementation of OMeta/JS can be used as a standalone package for Node.js. The interdependencies between the different modules are clearly separated to allow reasonable maintainability.

A fictional language EJS is introduced to demonstrate the possibilities of the framework at hand. For this purpose four example extensions are discussed, each with increasing complexity. The last extension finally introduces a class based object orientation.

Many improvements of the framework are imaginable. Yet, the architecture renders itself to be a good foundation to quickly develop JavaScript language extensions.

Contents

Chapter 1

Introduction

Contents

For almost every developer there are times, when the question arises: “Why, tell me why, can’t I simply do it like in x?”

In this saying, x can be replaced by any programming language that might have one additional feature to that language one is currently working with. This absent feature can be a complex one, unique to the language. However, these features often are of syntactical nature - in other words: just the *way to express* the things we want to happen. With every new programming language we have learned, we also possibly gain more frustration because there is no programming language that can fit all shoes. So most of the time we will be missing certain capabilities or ways to express our algorithms in a delightful manner.

There are languages that make us feel more locked-in than others do. On the other hand, some existing languages are full of beautiful features, waiting to be discovered[1]. Additionally there is a third category of languages, providing us with all the necessary tools to change the language itself where it doesn’t fit our needs. A good example for this group is Lisp, where every programmer can create his own set of tools and redefine almost anything, according to his personal likings.

Nevertheless, thsese shortcomings usually exist for a purpose. Sometimes “prosaic elegance” has been traded off against performance, security motives or internal consistency. So there are many good reasons that underly possible answers to the introductory question. Likely none of them will really comfort us in that kind of situation. We just have to keep in mind that languages are not born out of a singularity, somewhere in vacuo. Instead, languages are settled in a given environment as children of hard requirements

[1]According to my personal experiences, Ruby is a good example for this languages

that have a large impact on design decisions. Occasionally, mostly due to improvements of the underlying hardware, some of these requirements become obsolete with time passing, but others still apply.

1.1 JavaScript

JavaScript has been developed as language within the browser. With the magnificent gain of importance the Internet has received in the last decade, JavaScript has become one of the world's most used programming languages. It is used both by amateur web-developers with little programming-knowledge and professionals. The initial scope was to be a client-side scripting-language in order to allow interactivity on a webpage, after it has been downloaded by the client's browser. The bad reputation of being slow and insecure JavaScript has carried away from the early days is now covered with the success of buzzwords like Web 2.0 and AJAX. Meanwhile, many browser vendors did an outstanding work under the hood and significantly improved the performance and security of their JavaScript-interpreters. Currently, almost each and every site in the web is using JavaScript to add more or less comprehensive interactivity and visual effects. Dozens of large productivity web-applications like e-mail clients, image-editing and complete office-solutions emerge, built from tip to toe solely on JavaScript. Today, the only web technology providing client-side scripting which had a similar impact is Flash. But the web is moving fast - the upcoming HTML5 standard, in combination with some JavaScript APIs, is on the way to replace Flash as prominent tool for complex animations and the embedding of videos.

1.1.1 Outside of the Browser

However, the browser isn't the only environment of JavaScript anymore. The language is used for scripting purpose by many desktop applications. Gnome Shell, part of the window-manager of the Gnome 3 Desktop[2], makes intense use of JavaScript for window-positioning and compositing. The graphical user interface of Mozilla Thunderbird[3], an open source mail-client, is completely written in JavaScript and XUL. In consequence of the large diversity of mobile-devices, many mobile applications are currently built using web-standards to achieve a better compatibility across the different platforms. With WebOS[4] there even is an operating system for mobile-phones and embedded-devices which allows to develop native applications with web-technologies. But this evolution is not restricted on the mobile-market. Microsoft announced the release of Windows 8, an operating system optimized to be used with a touchscreen. Windows 8 also picks up the trend of so-called *apps*, small single purpose applications with an easy to use interface, implemented with HTML, CSS and JavaScript.

Programs written in JavaScript can even be found on server-side. With the dramatic performance improvements of JavaScript interpreters like Mozilla's Spidermonkey or Google's V8, those engines are more and more used as standalone interpreters for server-side scripting. Currently, the most famous environment for this very purpose is

[2]Gnome 3 is a graphical desktop environment atop of operating-systems like Linux (`http://www.gnome.org/gnome-3/`)
[3]`https://www.mozilla.org/en-US/thunderbird`
[4]`https://developer.palm.com`

Node.js, which uses V8 at it's core and wraps important operating-system functionality like file-handling and network-sockets in JavaScript-APIs.

Many of these interfaces are specified by a project, called CommonJS. It provides a modular, uniform API, that can be implemented by the different libraries and frameworks. Equipped with such an API, the underlying base-library of an application could easily be exchanged without the need to re-factor large portions of code. Furthermore, developers could use the same API in many of their projects, regardless if these are web-applications or scripts running on Node.js. The team of CommonJS aims to create the kind of comprehensive standard-library that most languages already have, but JavaScript is still missing.

1.1.2 JavaScript as a Standard

> "I worked there for almost a month, in May switched to client group, spent 10 days prototyping the core language. When Marc saw that, he said: this is it, it's Emacs, we're done, and then immediately pushed to get it shipped."
> - Brendan Eich

Originally *Mocha*, with all of it's flaws and it's beauty, has been created by Brendan Eich in about ten days. He has been hired by Netscape with the promise "to bring Scheme to the browser" but subsequently received the condition that it has to look like Java [6]. All unfortunate decisions aside, Brendan Eich managed to combine a Java-like syntax with Scheme's philosophy of functions as first-class citizens and Self's prototypal inheritance. The first implementation has been released under the name *LiveScript*, built in Netscape Navigator. A renaming to JavaScript followed, that has been the reason for a lot of confusion ever since. Many developers who are unfamiliar with the language believe JavaScript to be just the dynamic scripting brother of Java. Against this assumption and besides it's syntactical nature, JavaScript is an independent, functional programming language. In fact it has more in common with Lisp and Scheme then it has with Java. In 1996 Netscape handed JavaScript over to the ECMA International in order to establish a new industry standard. Since that day, the ECMA-262 standard specifies the language, furthermore known as *ECMAScript*. So many different names and implementations sometimes make it difficult to talk about "JavaScript". Hence in the remainder of this paper I refer to the current edition 5.1 of the ECMAScript standard (abbr. ES5) by saying "JavaScript".

Although it is called a "standard", the language itself is not frozen. The technical committee TC-39 is constantly polishing ECMAScript and trying to solve problems which are the result from historic miss-decisions. They are also trying to weave in knowledge and experiences, gathered from the vital community of JavaScript-developers and engine-implementors to make JavaScript a living language which evolves with time. After having problems while working on the 4th Edition the term "Harmony" has been coined. It is a metaphor for handling new ideas in a constructive manner and for working together open-minded at the next standard: ECMAScript Edition 6 - sometimes also referred to as ES.Next. This process addresses not only the members of the committee but also includes ideas and opinions from the JavaScript-community.

In an interview[5] Eich points out how important user testing is to support the standardization process. Sample implementations of the new language early can be used by developers to play around with, and therefore help to find bugs and misconceptions in the specification before the new standard is released. Especially usability issues may be detected that way. The experience, gathered this way, can flow back into the upcoming standard.

> “We can’t "do science" to decide what features in the enormous feature-vector hyperspace to standardize” - Brendan Eich

As a matter of fact, implementation and standardization go hand in hand. It is an iterative approach between the committee, implementors and users, which does not only apply to JavaScript-standardization but also to HTML, CSS and many other specifications. Considering the work of WhatWG on HTML5, browser-vendors often are some steps ahead of the committee, doing field-research with their implementations. It’s like an evolution of the standard: only the fittest, proven to work stay. Nevertheless, many developers and managers think of standardization as a three-step process: “First specify, then implement and finally use”.

1.1.3 Problems and Solutions

The environment in which JavaScript is being used changes almost every day. Hence, assumptions made at the very moment of language design are not valid anymore. As Douglas Crockford describes in [3] there are some misconceptions in the JavaScript language design often making it difficult to write clean and robust code. The most strikingly example is the shared global namespace which is used to “link” the different embedded scripts. On the other hand, some unique functionality like prototypal inheritance is covered under a verbose syntax. Last but not least, the exact result of a JavaScript program is highly dependant on the interpreter in which it is being executed. The combination of different browsers and different versions results in a matrix of numerous interpreters with diverse capabilities. This makes it very complicated to harmonize the result of a computation across multiple platforms. There are some strategies addressing those widespread problems:

1. Design patterns provide ready-to-serve solutions for certain semantic problems. The revealing module pattern (See chapter 7.1 on page 77) for instance is used to encapsulate the implementation of a module by only revealing those variables and functions which form the public interface while hiding all others.

2. Libraries are collections of functions and objects. There is a vast number of libraries each tailored to serve a special purpose. They may be grouped into three categories, though many can be classified in multiple categories at the same time:

   (a) Base libraries normalize the behavior of different browsers in order to simplify cross-platform development. One famous example is jQuery[6].

[5] `http://www.aminutewithbrendan.com/pages/20110805`
[6] `http://jquery.com/`

   (b) High-level language features like classical class-based inheritance or powerful collections can be implemented as part of a functional library. Ext.JS offers an extensive object orientation[7] while underscore.js provides methods for improved collection handling like `each`, `map` and `reduce`[8].

   (c) Frameworks simplify the development of larger applications by providing a structural foundation programmers can start to build on. The underlying principle is the *inversion of control*. Most frameworks also implement the *model view controller* pattern. Ext.JS and Sproutcore[9] both can be counted to this group.

3. Despite the dynamic behavior of JavaScript (For further information see [22]) there are tools for static analysis which provide mechanisms to analyze JavaScript code at compile time and detect possible problems and inconsistencies before execution. Two prominent examples for JavaScript code analysis are JSLint[10] and DoctorJS[11].

4. Special subsets of the JavaScript language enforce security on embedded third party scripts by either filtering and rewriting (Maffeis and Taly have done some research on this topic [14]) or by compiling to a capability safe subset of JavaScript (Also see [13]). A good example for this approach is Caja[12].

5. Compiled language extensions and JavaScript language derivatives introduce a new syntactical frontend for JavaScript and encapsulate the syntactic overhead, which is often created by common implementation patterns. Examples for this category are CoffeeScript[13], Move[14], Kaffeine[15] and JS11[16].

Since all of the solutions are directed at a partial subproblem, mostly a combination is chosen to match the requirements of a specific project. But there are limitations of what libraries and design patterns can do to improve the language. In JavaScript these boundaries often are of syntactical nature, because most missing features, like a class-system, can be simply reimplemented. In contrast an unaesthetic syntax (for semantically simple tasks often in combination with unnecessary complex constructs) cannot be circumvented that easy. The only solution is to adapt the language's frontend: The syntax.

Building on this reasoning I am focusing on the last solution above and present a way to create *custom compiled language extensions* with JavaScript as their target language.

1.1.4 Compiled Language Extensions

Supported by the incredible performance gain of JavaScript interpreters in the last decade more and more projects compile to JavaScript in order to provide an alternate syntax or make existing software-components usable in the web. Thus JavaScript seems

[7] `http://sencha.com`
[8] `http://documentcloud.github.com/underscore/`
[9] `http://sproutcore.com/`
[10] `http://jslint.com`
[11] `http://doctorjs.org`
[12] `http://code.google.com/p/google-caja`
[13] `http://coffeescript.org`
[14] `http://movelang.org`
[15] `http://weepy.github.com/kaffeine`
[16] `http://js11.org`

to become the “assembly of the web” (Eric Meyer). The list of languages with JavaScript as it’s compilation target is long [2]. There are compilers for existing languages like Lisp / Scheme, Smalltalk, Haskell, Ruby, Python, Java, Scala, C# and much more. With EMScripten (See [30]) there even is a compiler to translate LLVM (abbr. for Low Level Virtual Machine) assembly to JavaScript. This allows the compilation of all languages for which a LLVM frontend exists (These include amongst others C, C++, Objective-C, D and Fortran).

On the other hand new languages like CoffeeScript (including many derivatives), JS11, Kaffeine, Mochiscript[17], Jack[18] and Move are emerging. CoffeeScript offers an indentation based programming style similar to Python. JS11 claims itself to be a “compact version of JavaScript” and for instance reduces the need for semicolons and the function-keyword. Kaffeine and Mochiscript are supersets of JavaScript and add optional language features like a shorthand for function declaration or a pseudo-classical object orientation system.

They all share the goal of simplifying JavaScript-development by providing an individual syntactical frontend to the programmer. Some of those source-to-source compilers are written in JavaScript itself and thus can be executed inside of a browser environment. We will concentrate on this category of compilers out of two reasons:

1. They allow a developer to deliver code in a language alien to the browser while the client is able to execute it.
2. When developing extensions for JavaScript we apparently have gained expertise in this language before hand. This makes it reasonable to reduce the number of utilized languages and stay within JavaScript throughout the development of the compiler.

1.2 Scope of this Thesis

It is *not* the goal of this thesis to design yet another programming language that compiles to JavaScript. Instead, a framework will be presented that allows the creation of *custom extensions* to JavaScript in order to *a*) experiment with these extensions and *b*) create a tailored language for individual usage.

My personal motivation is to create individual extensions in order to make working with JavaScript more comfortable for me. The existing JavaScript derivatives introduce a lot of interesting features. Most of the time this is an opt-in into the whole package which also includes undesired features or behavior.

The purpose of this thesis is to be able to compose only those wanted features.

In order to achieve this goal the following chapter 2 roughly runs through the whole process of compilation. Chapter 3 offers a short look into the different parsing strategies to present *parsing expression grammars* as a tool of choice. Based on this decisions chapter 4 compares four different parser generators implemented in JavaScript. Due to it’s outstanding extensibility mechanisms OMeta/JS is chosen. As a result an introduction into OMeta/JS is given in chapter 5.

[17] `https://github.com/jeffsu/mochiscript`
[18] `https://github.com/creationix/jack`

Building on this foundation, in chapter 6 a five step solution is shown to create a syntactical extension for JavaScript. In the same chapter the resulting architecture is presented as well. The subsequent chapter 7 demonstrates the use of the framework by implementing four different example extensions. Finally a conclusion and an outlook is given in chapter 8.

Chapter 2

The Compilation Process

Contents

Generally speaking, the translation of one language or dialect (*source language*) to another language (*target language*) is divided into multiple phases. Each of these phases being

1. lexical analysis,
2. syntactical analysis and tree creation,
3. multiple tree transformations and finally
4. code generation

is shortly described in the remainder of this chapter. A full detailed explanation of all internal workings can be found at [1] and [18]. Figure 2.1 illustrates the various phases of the compilation pipeline. It is common practice that each step of the pipeline is performed by a decoupled module of the compiler or even by a separate program specialized on that very purpose.

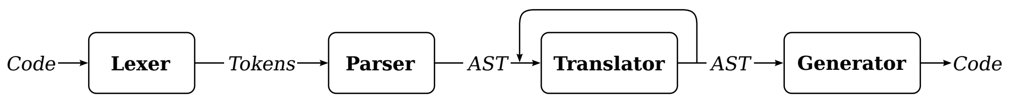


Figure 2.1: *The compilation pipeline*

The source language is analyzed by a lexer and afterwards by a parser. An abstract syntax tree is created, translated and finally the code in the target language is generated.

Nevertheless, the structure of the compiler presented here differs a little from the one described in [1, chapter 1]. Since we are not targeting machine code as final result of our compilation process we use abstract syntax trees (abbr. AST) as intermediate representation. Consequently, all semantical analysis and optimization of the schematic compiler described below are performed directly on the those trees. The same applies to the final code generation which operates on the AST as a result of the previous phase. When discussing the various steps of compilation in the remainder of this paper, the above pipeline is used as reference architecture.

2.1 Lexical Analysis

The input which is fed into the compiler is the source code written in the source language (For instance a JavaScript derivate). It is given as a string, which also can be seen as a *stream of characters*.

In the first phase a lexical analyzer or *lexer* (also called "tokenizer" or "scanner") scans the stream of characters and partitions it into a *stream of tokens*. Each *token* embraces one or more input characters into a unit of syntactical information. It can be represented as an object with a type and a value like the following JavaScript-object:

```
{ type: "ID", value: "foo" }
```

Regular expressions are often used to recognize the sequence of characters needed to form a token. Given the following regular expressions

```
VAR    = /var/
EQ     = /=/
ID     = /[a-zA-Z]+/
NUMBER = /[0-9]+(\.[0-9]+)?/
```

a lexer could transform the character stream

```
['v','a','r',' ','f','o','o',' ','=',' ','4'] // as string: "var foo = 4"
```

into the resulting token stream:

```
[{ type: "VAR", value: undefined }, { type: "ID", value: "foo" },
 { type: "EQ", value: undefined }, { type: "NUMBER", value: "4" }]
```

In this example the lexer simply skips over the whitespaces. If whitespaces are requried as important syntactical information or in order to preserve them during compilation, those could be saved in dedicated tokens like:

```
{ type: WHITESPACE, value: " " }
```

$$\begin{aligned} Program &\rightarrow Declaration \\ Declaration &\rightarrow \textbf{var id eq } Expr \\ Expr &\rightarrow \textbf{number} \\ Expr &\rightarrow \textbf{id} \end{aligned}$$

Figure 2.2: *Production rules needed to recognize an input like `var foo = 4`*

2.2 Syntactical Analysis

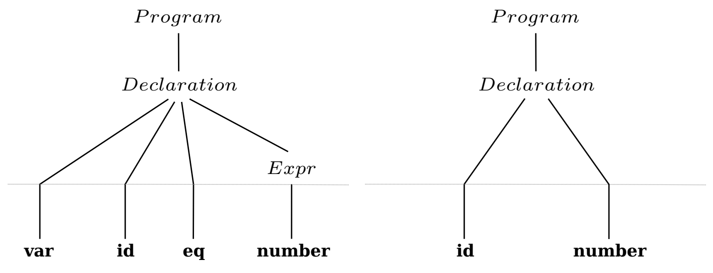
The stream of tokens created by the lexer serves as input for the second phase - the *syntactical analysis*. In that phase a *parser* analyzes the given input for syntactical structures and constructs an *abstract syntax tree* (abbr. AST). The parser utilizes rules related to the production rules of a context-free grammar in order to check whether a given input matches the specified language or not.

The grammar

$$G = (\{Program, Declaration, Expr\}, \{\textbf{var}, \textbf{eq}, \textbf{id}, \textbf{number}\}, P, Program)$$

with P being the set of rules from figure 2.2 could be used to recognize the input described above.

In order to distinguish between terminals and nonterminals the following convention is used: Words with a starting uppercase letter like $Program$ describe nonterminals, lowercase bold words like **number** indicate that a token (terminal) with the appropriate type is expected as next item within the input stream. Starting with the nonterminal $Program$ all occurring nonterminals on the right hand side of a production are replaced according to the existing rules. It should be noted that we use terminals and tokens synonymously since from the parser's point of view tokens are the smallest unit of information.



(a) *A parse tree as a result from parsing `var foo = 4`*   **(b)** *An abstract syntax tree*

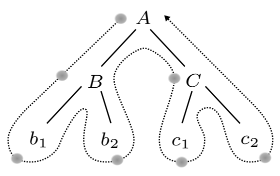
Figure 2.3: *Parse trees*

Figure 2.3a depicts the *parse tree* resulting from the input "`var foo = 4`". With this illustration all intermediate steps performed by the parser become visible. This is often too much information. Most of the time only the concentrated syntactic essence of a program is needed for further processing. To achieve this, *semantic actions* are introduced. With semantic actions it is possible to manipulate the results of a production in order to shape the parse tree during it's creation (Examples of semantic actions can be found in chapter 5.1). This way a condensed AST as seen in figure 2.3b can be created.

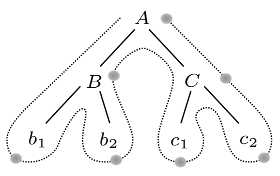
The purpose of an AST is to describe the syntactical structure of a program. Since all of the upcoming transformation passes are performed on various versions of the AST, it can be seen as the *"intermediate representation"* [18] of our compilation-process. Furthermore, it is important to specify the exact structure of the AST as it serves as an interface between the different phases.

2.3 Tree Transformations

The third phase - *semantical analysis and transformation* - consists of traversing and transforming the AST. These transformations are fulfilled by a *translator*. In contrast to the previous phases there are various translations that can be performed multiple times until the AST matches the desired format. Usually those translations are implemented using the visitor design pattern [10, p. 331] or by a traversing program called "walker". This name pictures the behavior of the program which has to recursively "walk" (or traverse) all nodes of an AST in order to translate them. According to the visitor design pattern every "visit" of a node in the AST requires a function to be implemented for each corresponding node-type. In the remainder these functions are also called *handlers*, since it is their purpose to handle a specific node type. Depending on when the translation of the respective child-nodes is performed, the walking strategy is said to be top-down (preorder traversal) or bottom-up (postorder traversal) as seen in figure 2.4. The sequence of traversal on the left hand side is $A, B, b_1, b_2, C, c_1, c_2$ whereas on the right hand side the children are visited first $b_1, b_2, B, c_1, c_2, C, A$.



(a) *Top-down traversal*



(b) *Bottom-up traversal*

Figure 2.4: *Top-down / preorder and bottom-up / postorder traversal of a tree*

In general, the tree is traversed for mainly two reasons. Firstly, the tree can be processed to statically analyze the programs structure. For example variable declarations and their usage may be tracked in order to mangle variable names or to analyze and control the access to global variables. Secondly, these nodes may be modified while visiting them resulting in a transformation. These transformations can be used to translate the format of single nodes, sub-trees or the AST as a whole from one language (or dialect) to another. On the other hand, conventions like "Always declare variables at the top of the scope" can be forced that way.

2.4 Code Generation

The final step of each compiler is the generation of code in the target language (e.g. Assembler or as in our case JavaScript). While most compilers have machine specific code as their target we focus on translating one high-level language into another. Furthermore, it is possible to compile into the same language as the source language. This may sound surprising at first. However, depending on the transformations and optimizations performed in the previous phases a generation of code in the source language can be reasonable. One common example for source to source compilation within the same language are compressors. They aim to minify the overall size of the program, in order to save bandwidth, by removing unnecessary whitespaces and by shortening variable names among many other optimizations.

Nevertheless, the most common use case is to output code in a language different to the source language. Since we are focusing on language extensions for JavaScript the output of our compiler is always JavaScript itself, whereas the input is a derivative of JavaScript. The overall transformation can be expressed as

$$JavaScript' \rightarrow AST_{JS'} \rightarrow AST_{JS} \rightarrow JavaScript$$

with $JavaScript'$ representing any JavaScript dialect and with $AST_{JS'}$ being the corresponding abstract syntax tree. This tree is transformed afterwards to match the format of a generic JavaScript tree AST_{JS} which is finally translated back to $JavaScript$-source code.

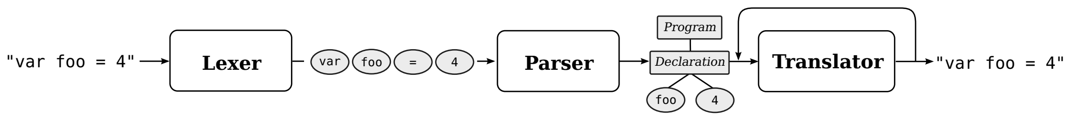


Figure 2.5: *Overview of the compilation pipeline for `var foo = 4`*

Generally speaking, the transformation of an AST to source code is just a special category of AST translations as described in the previous section. The only difference is that after this final transformation no other traversal can be performed without repeated parsing of the generated source. Based on this insight it is possible to further simplify the compilation process by merging the code generation with the previous phase. Figure 2.5 illustrates the pipeline for the compilation process of input “`var foo = 4`”. The input stream is split up into several tokens which feed into the parser. The parser uses a grammar to translate the token stream into an abstract syntax tree. The tree is then processed by zero or more translators before a final transformation emits the desired code in the target language.

2.5 Summary

In this chapter we have seen how the compilation process can be broken down into several phases. Starting with the lexical analysis a stream of characters is recognized and transformed into a stream of tokens. The result in turn is passed to the next phase:

the syntactical analysis. The language which can be recognized by a parser is composed of the different rules. These rules can be compared to the productions of a context free grammar. After syntactically analyzing the token stream the parser creates an abstract syntax tree. Multiple translation passes follow which traverse the abstract syntax tree in order to analyze and transform it before the final transformation (or code generation) can be applied.

Chapter 3

A little more about Parsers

Contents

In section 2.2 we have seen how parsers are used to analyze the token stream in order to create an abstract syntax tree. To be able to compare the different available parsers we have to get a little more into detail about how parsers can be built.

Writing parsers is preceded most of the time by designing a grammar for the language which shall be parsed. For this purpose a context free grammar (abbr. CFG) or the Backus-Naur form (abbr. BNF) is often used as notational foundation. Commonly, further subsets of the class of CFGs are created to be able to efficiently implement a parser or automatically generate it by using *parser generators*. For instance $LL(k)$-grammars describe the subclass of those context free grammars which can be used to create a top-down parser with a fixed amount of lookahead. In contrast LR-grammars form the class of grammars that can be used to generate bottom-up parsers. We may group parsers based on such grammars into categories

1. whether they create the parse tree *top-down* or *bottom-up*, or
2. whether they are implemented using a *recursive decent* or *table based* approach.

A combination of the possible options results in the matrix as seen in table 3.1.

It is possible to implement a parser as a *recursive decent parser*, or by using a *table driven* model (often in combination with a state-machine). Most of the time parsers following the recursive decent approach are also of the category *top-down* (LL)[1].

[1]Pepper showed in [19] that, after a grammar has been transformed into the third normal form, it can be used to create a recursive decent parser which produces the same output as a classic LR-parser.

	top-down	bottom-up
recursive decent	LL	transformation + LL
table based	non recursive LL	LR

Table 3.1: *Ways to build parsers*

Bottom-up parsers (LR) usually offer a less restrictive way to express grammars than top-down parsers do. For instance it is possible to use left-recursive rules without the need to refactor the grammar. Nevertheless they also come with a much more complicated implementation which is nearly impossible to be accomplished by hand. The same applies to non recursive top-down implementations (in addition they are rarely used in practice).

Although most of the time we use parser generators to automatically create parsers from grammars, it is always useful to be able to easily understand the generated source code. Recursive decent parsers implicitly meet this criterion since every nonterminal maps to a function in the parser implementation. Using a debugger the programmer always is able to step through the parsing-process and directly see what parsing-functions are being applied to match the given input. Furthermore, dividing the parser into modular units according to the nonterminals can simplify the reuse of parts of the language in other parsers. Since this simplicity appears to be of major importance (and as we will see many of the restrictions of the top-down approach can be bypassed), in the remainder I will focus on recursive decent parsers in general and parsing expression grammars (abbr. PEG) in special.

Example

In this section small parser is constructed which is able to recognize a subset of the Lisp language. The context-free grammar

$$G = (\{Program, List, ListItem, Atom\}, \{\mathbf{(}, \mathbf{)}, \mathbf{.}, \mathbf{id}, \mathbf{number}, \mathbf{eos}\}, P, Program)$$

with P being the rules as seen in figure 3.1 describes the language which recognizes words like `(3 . 4)` and `(add . (4 . (5 . nil)))`. The token **eos** depicts the end of the input stream.

3.1 Recursive Decent Parsers

The first and most intuitive attempt to build a parser by hand is to implement it as a *recursive decent parser*. Even if the parser can be built by hand it is always helpful to outline the grammar of the language like in figure 3.1. A recursive decent parser can be implemented by simply mapping all nonterminals to parsing-functions. Here the grammar's productions are part of the function's implementation. We assume that the lexer provides the two functions `peek` and `consume`. `peek()` allows to gather the next token in the input stream without altering the stream while `consume(type)` in turn

$$
\begin{aligned}
Program &\rightarrow List\ \textbf{eos} \\
Program &\rightarrow Atom\ \textbf{eos} \\
List &\rightarrow \textbf{(}\ ListItem\ \textbf{.}\ ListItem\ \textbf{)} \\
ListItem &\rightarrow List \\
ListItem &\rightarrow Atom \\
Atom &\rightarrow \textbf{id} \\
Atom &\rightarrow \textbf{number}
\end{aligned}
$$

Figure 3.1: *Production rules describing a subset of Lisp*

would remove the next token. If the optional `type` cannot be matched by the lexer an error will be thrown.

```
function Program() {
  var result, next = peek();
  if(/* next predicts list */) {
    result = List();
  } else if(/* next predicts atom */) {
    result = Atom();
  } else {
    throw "Expected list or atom";
  }
  consume('eos');
  return result;
}
```

Figure 3.2: *Example implementation of the production rule* $Program$

In the implementation of $Program$, seen in figure 3.2, it gets visible that each of the choices is mapped to an `if`-branch trying to predict the appropriate matching-function by looking at the next token. If the condition is met successfully, the associated sub-rule is executed. The parser works it's way *top down* through the grammar recursively resolving all nonterminals. It is said to be a *predictive parser* [1, p. 64-68], because it uses a fixed amount of lookahead tokens to predict the further flow of control.

```
function List() {
  var first, rest;
  consume('(');
  first = ListItem();
  consume('.');
  rest = ListItem();
  consume(')');
  return [first, rest];
}
```

Figure 3.3: *Example implementation of the production rule* $List$

Since the parser processes the input stream from the **L**eft to the right and the recursive resolution creates a **L**eftmost derivation of the grammar, the parser is also said to be of the category $LL(k)$ [1, p. 222-233]. Here k indicates the fixed amount of lookahead required for prediction. In our case just one token is used for this purpose, thus the parser is of the category $LL(1)$. Sometimes more than one lookahead is required to make decisions. The easiest solution is to use a *circular buffer*, which can be implemented in the lexer. This buffer can store the next k tokens of the stream in an array of length k utilizing a modulo operation [18, p. 45-48].

Figure 3.3 depicts the sample implementation of the nonterminal $List$. The function recursively calls `ListItem` and finally returns an array containing the captured results. The implementation of all other functions follows likewise and can be found in appendix A.3. Parsing the input string "`(add . (5 . (4 . nil)))`" with the final parser will result into the array `["add", [5, [4, "nil"]]]`.

While recursive decent parsers require a modest effort when being implemented by hand they also exhibit some problems. For a CFG grammar to be $LL(k)$ it requires some restrictions:

- No left recursion
- No ambiguities in alternative productions

Invoking a rule which calls itself directly or indirectly without consuming any input can lead to an endless recursion. This limitation can be circumvented by rewriting the grammar from left-recursion to right-recursion [1]. For instance the grammar

$$G = (\{List\}, \{\textbf{item}, \textbf{,}\}, P, List)$$

with P being the production rules

$$List \rightarrow List \ \textbf{,} \ \textbf{item}$$
$$List \rightarrow \textbf{item}$$

can be rewritten to be right-recursive:

$$List \rightarrow \textbf{item}$$
$$List \rightarrow \textbf{item} \ \textbf{,} \ List$$

Most rewriting is more complicated than this example because usually the associativity and precedence of operators has to be preserved [1, p. 193]. Even if rewriting solves the problem of endless recursion, it often obscures the grammar and adds additional complexity for implementors and readers of the grammar.

Due to the fact that predictive parsers only use a fixed amount of lookahead no ambiguities in the productions of a nonterminal are allowed.

In other words, the sequence of k upcoming symbols for two productions has to be disjoint to decide which path to take. In consequence, only one of the alternatives can derive the empty string. Otherwise the parser cannot decide which empty string it should accept. Finally, if one alternative production derives the empty string the other may not start with the same sequence of k terminals that is used right after the nonterminal which is derived.

Example

The simplified production rules describing a JavaScript for-statement

$$\begin{aligned} ForStmt &\rightarrow \textbf{for (} Expr \textbf{ ; } Expr \textbf{ ; } Expr \textbf{)} \ Stmt \\ ForStmt &\rightarrow \textbf{for (} VarDecl \textbf{ in } Expr \textbf{)} \ Stmt \end{aligned}$$

are not $LL(1)$. Neither they are $LL(2)$ since at least three tokens have to be inspected until a decision can be made. In general, two rules that start with the same combination of grammar symbols have to be rewritten using *left factoring* (See [1, p. 214]). The rewrite rules applied to the example above result in

$$\begin{aligned} ForStmt &\rightarrow \textbf{for (} \ ForRest \\ ForRest &\rightarrow Expr \textbf{ ; } Expr \textbf{ ; } Expr \textbf{)} \ Stmt \\ ForStmt &\rightarrow VarDecl \textbf{ in } Expr \textbf{)} \ Stmt \end{aligned}$$

which in turn might be rewritten into the more elegant solution:

$$\begin{aligned} ForStmt &\rightarrow \textbf{for (} \ ForHeader \textbf{)} \ Stmt \\ ForHeader &\rightarrow Expr \textbf{ ; } Expr \textbf{ ; } Expr \\ ForHeader &\rightarrow VarDecl \textbf{ in } Expr \end{aligned}$$

Every production now begins with another terminal or nonterminal. Depending on the exact implementation of $Expr$ and $VarDecl$ (which may cause further ambiguity) a decision can be made right after looking at the first token of lookahead.

3.2 Extending Recursive Decent Parsers

While recursive decent parsers offer a fast implementation which is easy to understand they also come with certain limitations. This section deals with extensions, that can be used to bypass some of them.

3.2.1 Backtracking

As we have seen, grammars sometimes can be rewritten to be compatible for predictive parsing. However, predictive parsers fail when language-constructs cannot be recognized by solely inspecting a fixed amount of lookahead.

A solution to this problem is *backtracking*. Here, all available options are speculatively tried one after another. If an option fails the parser has to rewind the input-stream to the last position with a successful, match before it can start over and try the next option. The first successful alternative is accepted, therefore causing the options to appear *ordered*.

There are two common ways to implement backtracking parsers. The first one is to maintain a stack of stream positions. This stack may be compared to a memory of performed commands in order to be able to "undo" them. Entering a rule, the current position is pushed onto the stack. If the rule succeeds the new position is accepted and the stored position is popped off the stack and discarded. In case of failure the position also is popped off the stack and used to restore the previous state of the parser. Another way to implement backtracking is to utilize the automatic propagation of exceptions through the call-stack in order to catch failed trials and restore the previous state.

$$\begin{aligned} Statement &\rightarrow Definition \\ Statement &\rightarrow Declaration \\ Definition &\rightarrow \textbf{var id =}\ Expression \\ Declaration &\rightarrow \textbf{var id} \end{aligned}$$

Figure 3.4: *Grammar describing variable declarations and definitions*

The unlimited lookahead of backtracking parsers comes with an exponential performance penalty. Considering the grammar as seen in figure 3.4 and the input "`var foo`", the parser will speculatively match the first alternative $Definition$ until it reaches **=**. At this point the option fails and the second one $Declaration$ is tried. Both **var** and **id** are matched a second time, even if they have been previously parsed with success at the same position. The following example (figure 3.5) illustrates a possible implementation of $Statement$ using exception handling for backtracking.

```
function Statement() {
  var result, before = current_pos();

  // try alternative speculative
  try { result = Definition() }
  catch(e) {
    set_pos(before);
    try { result = Declaration() }
    catch(e) {
      throw "Expected Definition or Declaration at position " + before;
    }
  }
  return result;
}
```

Figure 3.5: *Backtracking implementation for production* $Statement$

In this example `current_pos` and `set_pos` are again functions provided by the lexer to control the position within the input stream.

3.2.2 Memoization

Two solutions came up addressing the problem of exponential parsing time since it is not acceptable in many applications:

1. Only use backtracking in those cases an unlimited lookahead is truly required
2. Use memoization to restore the guarantee for linear parse time

While ANTLR offers a semantic predicate to trigger backtracking for a special rule [18, p. 51] by hand - other parser generators like OMeta/JS [27] choose the way of memoizing the results. Bryan Ford [8] rediscovered the concept of *memoization*, originated in functional programming, for parsing and named it *packrat parsing*. Every parsing result (failure or success) at a certain position is temporarily stored and therefore only computed once. This methodology guarantees a linear parsing time while still offering full backtracking support and therefore allows not only arbitrary but *unlimited lookahead*. However, the storage cost for memoizing grows proportional to the size of the given input. Considering the amount of memory in modern computers it appears reasonable to pay this price. In addition, experiments [20] have shown that in practical use it is sufficient to memoize the last two recent results of a rule in order to reduce about 99% of the redundant calls.

```
var memo = {};
function Definition() {
  var pos = current_pos(), mem = memo[pos], result;

  // Has already been matched -> skip ahead and reuse
  if(!mem && mem.success) {
    set_pos(mem.end);
    return mem.result;
  }
  // Result already has been an error
  else if(!mem)
    throw mem.error;

  try {
    result = ... // Actually match the production of Definition
  } catch(e) {
    memo[pos] = { success: false, error: e }
    throw e;
  }

  // Success: save for next time
  memo[pos] = { success: true, result: result, end: current_pos() };
  return result;
}
```

Figure 3.6: *Memoization implementation for rule* $Definition$

A pseudo implementation of the rule $Definition$ in a memoizing backtracking parser can be seen in figure 3.6. When invoking the rule `Definition` it is checked whether the computation has been already performed at this very position. In this case the previous result is being reused regardless of success or error. Otherwise it is tried to match the productions of $Definition$. The result (again success or error) is cached in the memoization-object `memo` to be reused in subsequent processing.

3.2.3 Left Recursion

We have seen that left recursion can be eliminated by rewriting the underlying grammar. This often results in drawbacks regarding readability and complexity. Warth et al. showed in [28] that packrat parsers can support left recursion directly by modifying the memoization-algorithm. This makes it possible to use left-recursion in grammars without the need to rewrite them manually or automatically.

3.2.4 Parsing Expression Grammars

All grammars that appeared up to this point have been context free grammars (abbr. CFG). In CFGs choices are unordered. This often leads to ambiguities when parse-trees are created using those grammars (See [1, p. 203]). As seen in the previous section, backtracking parsers implicitly reduce this ambiguities by matching the choices one after another until one succeeds. With parsing expression grammars [9] (abbr. PEG) Ford introduced a new class of formal languages that embraces ordering by making every option a *prioritized choice*. In difference to CFGs, parsing expression grammars focus on *recognizing* languages rather than *generating* them. The syntax of PEGs is a combination of CFG and regular expressions (abbr. RE) and looks similar to the Extended Backus-Naur Form (abbr. EBNF). Likewise a PEG is built up of *terminals* and *nonterminals* which have to be resolved through the use of *parsing expressions*. Non-technically speaking, parsing expressions may also be called *rules*. The components from which every PEG is built can be found in table 3.2.

' '	String Literal
[ ]	Character class as known from regular expressions
.	Match any character
($expr$)	Group to force precedence
$expr$?	Optional occurrence of $expr$
$expr*$	Zero or many subsequent $expr$
$expr$+	Many $expr$, at least one occurence
&$expr$	Positive lookahead without consuming
!$expr$	Negative lookahead without consuming
$expr_a$ $expr_b$	Sequence of expressions, which have to match in this order
$expr_a$ / $expr_b$	Ordered choice. If $expr_a$ does not match $expr_b$ is tried

Table 3.2: *Operators for parsing expression grammars*

The PEG rules as in figure 3.7 specify the same language as it is described by the context free grammar of figure 3.1.

Lexers are usually described using REs and parsers using CFGs. Yet, in this example it gets obvious that PEGs allow to describe both of them in a single grammar. Some nonterminals like Id and $Space$ simply carry out the work of the lexer.

Ford points out that PEGs are both, more expressive than $LL(k)$ grammars and powerful enough to recognize all $LR(k)$-languages. Finally, PEGs are by design easy to implement as packrat parsers and therefore can benefit from backtracking, memoization and the left-recursion optimizations as described above.

$$\begin{aligned}
Program &\leftarrow (List\ /\ \ Atom)\ Eos \\
List &\leftarrow\ '('\ ListItem\ Space?\ '.'\ Space?\ ListItem\ ')' \\
ListItem &\leftarrow (List\ /\ \ Atom) \\
Atom &\leftarrow Id\ /\ \ Number \\
Id &\leftarrow [a-zA-Z\_]+ \\
Number &\leftarrow [0-9]+\ \ ('.'\ [0-9]+)? \\
Space &\leftarrow\ '\ '\ /\ '\backslash n' \\
Eos &\leftarrow\ !.
\end{aligned}$$

Figure 3.7: *Parsing expression grammar that recognizes a subset of Lisp*

3.2.5 Semantic Predicates

Recursive decent as well as table-driven parsers reach their limits when it comes to decisions depending on context. For this purpose *semantic predicates* [17] are introduced acting as a guard for the continuation of rule-matching. Semantic predicates are tests which have to return a boolean value. They are implemented in the same language as the parser (the *host language*) and can be used to differentiate between otherwise ambiguous rules by either including context-information or computations in the decision. A semantic predicate can be used for instance to decide whether the currently parsed ECMAScript 5 code is in strict mode context or not. The following example grammar is akin to the specification [5].

$$\begin{aligned}
&AssignExpr \rightarrow ?(\texttt{strict\_mode})\ AssignableExpr\ \texttt{=}\ Expr \\
&AssignExpr \rightarrow LeftExpr\ \texttt{=}\ Expr
\end{aligned}$$

The semantic predicate indicated by $?(\ldots)$ guards the test for $AssignExpr$ and all following. If it evaluates to `true`, the left hand side expression has to be assignable. This excludes for instance `eval` and `arguments` which cannot be reassigned in ES5 strict mode.

3.3 Summary

In this chapter we have seen that parsers can be created to derive a grammar top-down (LL) or reducing it bottom-up (LR). We limited our analysis on top-down parsers only, based on the insight that LR parsers show a large complexity in the matter of implementation, making debugging of the generated code too difficult. The recursive decent approach has been sketched as it is the most important implementation of top-down parsers. We have seen that top-down parsers can be grouped into predicting parsers and backtracking parsers. While predicting parsers are always using a fixed lookahead to foresee the next decision, backtracking parsers are utilizing the trial and error strategy to mimic an unlimited lookahead. The major downside of backtracking being exponential runtime behavior could be reduced to linear time by memoizing intermediate results. In addition, further convenience could be obtained by allowing direct and indirect left-recursion in packrat parsers.

Finally, semantic predicates have been introduced to break the barrier of context free grammars. With semantic predicates it gets possible for the parser to select productions depending on the current context and thus behave context aware.

Chapter 4

Existing Parser Generators

Contents

The use of grammars to automatically generate parsers introduces a level of abstraction that makes it more easy to concentrate on the language itself and “what the parser should do” instead of focusing on the implementation details. Extensions to the language, as well as to the parser, can be implemented on this abstraction level, since only the pure language essence gets visible. In the following, we will notice that this complete separation of specification and implementation cannot be achieved at all times. Nevertheless, parser generators are a good starting point for language creation and extension.

There are many existing parser-generators, designed for various purposes, as it may be seen in the comprehensive lists [2, 21, 11, 29]. The selection can be reduced a lot by focusing on the parser-generators with JavaScript both as their target and implementation language. Table 4.1 shows the available parser generators grouped by the particular class of grammar: bottom-up[1], parsing expression grammars and parser combinators. *Parser combinators* are a special form of parser generators, since they generate the parser not once from a grammar but built it at runtime recursively from smaller parsers. To achieve this they make use of functional programming paradigms such as higher order functions and monads. This concept is quite powerful but removes the abstraction from the grammar, because the implementation of the parser is also the description of the language[2].

[1]Even if jison has a LL-mode it is categorized as bottom-up, since this is the major target of this tool
[2]For instance the rule $A \rightarrow B\ C*$ is denoted as `var A = seq(B, many(C))`

Therefore, as noticed in chapter 3 we will focus on those tools only which use PEG to describe the language since PEG-parsers offer a reasonable combination of

- easy to understand source code,
- acceptable performance (improved with memoization),
- flexibility in writing grammars with no ambiguities and unlimited lookahead and
- the unification of lexer and parser.

This leaves us with the four alternatives Canopy, Language.js, OMeta/JS and PEG.js which are compared in the remainder of this chapter.

Parser Generator	Grammar Class	URL
JS/CC	$LALR(1)$	`http://jscc.jmksf.com`
jison	$LALR(1)$, $LR(0)$, $LR(1)$, $SLR(1)$, $LL(1)$	`http://zaach.github.com/jison/`
Canopy	PEG	`https://github.com/jcoglan/canopy`
Language.js	PEG	`http://languagejs.com/`
OMeta/JS	Extended PEG	`http://tinlizzie.org/ometa/`
PEG.js	PEG	`http://pegjs.majda.cz/`
jsparse	Parser Combinators	`https://github.com/doublec/jsparse`
ReParse	Parser Combinators	`https://github.com/weaver/ReParse`
P4JS	Parser Combinators	`https://github.com/asmyczek/p4js`

Table 4.1: *List of Parser generators with JavaScript target written in JavaScript*

4.1 Comparison of PEG-Parser Generators

Each of the four parser generators has been tested using a simple Lisp-grammar similar to figure 5.8. All four grammars can be found in appendix A.1. The performance of the created parsers has not been tested, because it is not of major concern when experimenting with language extensions. Even in a production environment the performance is not critical most of the time since the compilation, in most cases, will be performed only once during deployment.

All four generators are written in JavaScript and emit parsers also using JavaScript as implementation language. Therefore they all can be used within a browser-environment to compile grammars on the client side. In fact, PEG.js[3] and OMeta/JS[4] both offer an

[3] `http://pegjs.majda.cz/online`
[4] `http://www.tinlizzie.org/ometa-js`

interactive playground to write and test grammars. While the PEG.js online generator is split up into a three step process (write a grammar, provide input to match against grammar, download compiled grammar) the OMeta/JS online version follows a Smalltalk-like workspace approach. The latter makes it more difficult for developers unfamiliar with OMeta/JS to learn the language itself and how to use the parser generator.

A summarized comparison of all four parsers can be seen in table 4.2 on page 30.

4.1.1 Concepts

Canopy and Language.js both can be seen as simple implementations of a packrat parser generator for PEG grammars. Each of the two offers one extra concept that makes it unique.

Canopy allows expressions to be *annotated with types*. In Canopy's terminology a "type" can be compared to a mixin or a module. These collections of methods may be included into an object in order to enhance it. For example an array-node could be equipped with functionality to print the array's children.

Language.js offers a concept to add error messages which don't interrupt the parsing process when they occur. Instead of terminating they generate a *graceful warning*. For instance an omitted semicolon could be automatically be inserted and produce a syntax-warning.

OMeta/JS and PEG.js both offer a full range of extended PEG features. This includes *semantic predicates* as guards for further rule matching and *semantic actions* to affect the output of a rule or influence the state of the parser. Semantic actions also can be used to flexibly shape the abstract syntax tree, which is emitted by the parser.

Parsing expression grammars unite the lexical and syntactical analysis in one grammar. OMeta takes this concept even a step further by claiming itself to be applicable in every stage of compilation. This includes the traversal, matching and conversion of the AST. Hence, not only streams of characters can be matched by an OMeta grammar but every stream of generic objects. This decision leads to the consequence that no syntax for character ranges like $[a - z]$ can be found in OMeta. Luckily, this disadvantage can be circumvented by utilizing the outstanding concept of *parametrized rules*[5].

PEG.js keeps the syntax close to classic parsing expression grammars. The syntax of OMeta however differs slightly from PEG, since it introduces some novel concepts like *grammar inheritance* and *higher order rules* which had to be embedded into the language design.

4.1.2 Documentation

Most of Canopy's documentation covers PEG operators only. The dependency to another project is not explained in detail, making the setup a burden. Additionally I did not get the proclaimed key feature of Canopy, the type annotations, to work within reasonable time. Language.js offers no documentation at all, as a result I was not able to use the advertised graceful warnings. The most comprehensive documentation of OMeta/JS can

[5]Thus the range $[a - z]$ can be written as `range('a','z')`

be found in [27] but is neither complete nor up to date. For instance no information about the setup process or component dependencies can be found. Also, new operators have been introduced in the meantime which are not part of the original thesis. Nevertheless, the theoretical foundation of the major concepts is explained very well, allowing the developer to understand how OMeta works basically. The PEG.js documentation provides a short explanation about the installation and every feature of PEG.js, resulting in the easiest installation.

4.1.3 Error Reporting

When developing grammars, error reporting is important in order to be able to find flawed parts in grammars as fast as possible. Secondly, the generated parser should create meaningful errors to support the user. The implementation of smart error reporting in PEG-parsers is not as easy as in bottom-up parsers, which detects errors at the moment they occur. PEG-parsers try alternatives speculatively, hence an error in one alternative production does not necessarily result in complete failure. Only one of the four parsers at hand provides a reasonable error reporting, which is PEG.js. It captures all attempts to match options and uses them for error reporting in case there is no viable alternative:

```
SyntaxError: Expected "(", ".", "/", ";", character class or whitespace but "=" found.
```

PEG.js also detects direct or indirect left-recursion in the grammar and reports it as an error. It is noteworthy that OMeta/JS doesn't only recognize but also *allows left-recursion*. This is achieved by slightly modifying the memoization algorithm as seen in [28].

The idea of Language.js to offer error-recovery utilities in the grammar seems good at first glance. Nevertheless, the recovery operator `%` never worked in the experiment, since the input has always been accepted completely unparsed if an error occurred. Canopy and OMeta/JS both report the position at which the match failed. As additional information Canopy reports the last expected option. This can only serve as a small hint, where the matching process stopped.

4.1.4 Extensibility

Since all four solutions implement parsing expression grammars, they already provide a basic extensibility in terms of grammar abstraction. Rules can easily be copied from one grammar to another, without bothering about actual implementation. But this strategy results in the exact same problems as copy and paste mostly does:

1. Changes of the original grammar don't have influence on the derived grammar.
2. Duplicated rules lead to
   (a) larger and untidy grammars,
   (b) maintenance overhead.

PEG does not offer a way to build modular grammars and reuse rules of foreign grammars. Nevertheless, Canopy bundles all rules for a grammar inside a module which can be compared to mixins[6]. This may simplify the composition of two different grammars into one parser. Even if this modular composition of grammars is not an explicit target of Canopy, it may be achieved with just a few adaptations of the source code. If a rule is specified in two grammars one of these rules simply overrides the other. Hence, only disjoint grammars can be composed or an overriding order has to be clearly specified. This strategy however does not allow to invoke overridden rules nor does it implement a true grammar inheritance.

These concepts are part of OMeta/JS where one grammar can inherit from another and is still able to explicitly call overridden rules using a dedicated *super-call* operator. Additionally grammars don't have to inherit from one another to be composed. OMeta/JS allows the invocation of *foreign-rules* resulting in the input stream to be borrowed by the foreign grammar. When the foreign-rule has been matched the control-flow returns back to the original grammar.

4.1.5 Conclusion

OMeta/JS is not flawless and hence some downsides became visible in the comparison. The setup process and the rich feature list are insufficiently documented. In addition, it cannot compete with the more matured error-reporting of PEG.js. Nevertheless, since we focus on extending a language, the reuse of grammars in order to extend them is of highest precedence. The valuable concepts of OMeta/JS address this very purpose. Additionally, OMeta/JS implements most of the extensions a recursive decent parser has to offer. It is packed with memoization and automatic left recursion elimination. The support for higher order rules somehow makes it comparable with a parser combinator. Furthermore it supports the inheritance of grammars and therefore is the best pick for parsers which are easy to extend. Despite of the missing extension mechanisms PEG.js appears to be a solid, matured parser generator and probably would have been the tool of choice in another context.

[6]Mixins are a restricted form of multi-inheritance. They allow to mix in modular behavior to an existing class without the drawbacks of inheritance

	Canopy	Language.js	OMeta/JS	PEG.js
Implementation Language	JavaScript	JavaScript	JavaScript	JavaScript
Target Language	JavaScript	JavaScript	JavaScript	JavaScript
Grammar	PEG	PEG	OMeta	PEG
Parsing Algorithm	Packrat	Packrat	Modified Packrat	Packrat
Dependencies (Compiletime)	js.class	-	-	-
Dependencies (Runtime)	js.class	-	Parent-grammar	-
Documentation	✓	✗	✓	✓
Error Reporting (Compiletime)	✗	✓	✓	✔
Error Reporting (Runtime)	✓	✗	✓	✔
Readable Code Output	✓	✗	✓	✔
Left-recursion	✗	✗	✔	✓
Semantic Predicates	✗	✗	✔	✔
Semantic Actions	✗	✗	✔	✔
Grammar Reuse	✓	✗	✔	✗
Influence on AST-Format	✓	✗	✔	✔

✔	supported feature
✓	partially supported feature with limited usability
✗	feature which is either not supported or could not be used in reasonable time

Table 4.2: *Comparison of PEG-Parser Generators*

Chapter 5

OMeta

Contents

Writing parsers by hand can be quite tedious and error-prone, especially when implementing a language specification that still evolves or when experimenting with novel language features. Small changes in a grammar may result in complex changes of the parser. Thus, parser-generators are often used to automate this process. As we have seen in chapter 2 on page 9, a compiler commonly consists not only of a parser but also of a lexer, several translators and finally a code-generator - all of which being created with different tools or frameworks and maybe even using different languages. Alessandro Warth created OMeta to unify all of those tools in order to flatten the learning curve and to make experimenting with languages more easy [27].

The goal of this chapter is to provide a solid understanding of how to work with OMeta and specially OMeta/JS. Of course Warth's thesis [27] is a great source for background information and this chapter may be seen as a restructured, updated and enriched form of [27, chapter 2].

OMeta is a general purpose *pattern matching language* based on parsing expression grammars (abbr. PEG). As we have seen PEGs unite the flexibility of CFGs and REs and thereby remove the separation between the process of lexical analysis and parsing. They usually operate on characters as terminals and hence can only be used to match strings.

OMeta circumvents this limitation by allowing every object of the host-language to be a terminal, thus making it possible to use OMeta in almost every step of the compilation process. It also offers many extensions to PEG like *parametrized rules, higher order rules* and *grammar inheritance* described in the remainder of this chapter.

Working with OMeta can be split into three single steps:

1. Write your grammar in OMeta-language
2. Set up the OMeta-compiler and compile the grammar
3. Use the resulting grammar-object to match and translate input-streams

OMeta uses *memoization* to increase performance and therefore reduces the drawbacks resulting from backtracking (Also see [8] and chapter 3.2.2 on page 20). In addition, it allows the use of *left-recursive rules* by modifying the memoization algorithm (See [28] and [27, chapter 3]).

OMeta/JS

Being a generic language for grammar-description, OMeta has been implemented in many different *host languages*. In the remainder we will describe Alessandro Warth's reference implementation written in JavaScript (called OMeta/JS[1]), as it is close to the one used in the remainder of this thesis. Some examples of this chapter are taken from the `ES5Parser` presented in section 6.1.2 on page 51 (The parser itself is based on the JavaScript-parser delivered with OMeta/JS[2]).

In order to get an idea of how an OMeta grammar looks like, figure 5.1 shows a grammar with three simple rules, each *separated by a comma*. This simplified grammar matches JavaScript identifiers like `foobar`, `$1` and `_global`, always starting with rule `identifier`.

```
ometa ID {
  identifier = nameFirst namePart*,
  nameFirst  = letter | '$' | '_',
  namePart   = nameFirst | digit
}
```

Figure 5.1: *Sample grammar which can be used to match JavaScript-identifiers*

Here the structure of every OMeta gets visible. Since OMeta/JS is a combination of the OMeta-language and JavaScript the keyword `ometa` is used to announce that a following section is written in OMeta. After the introductory keyword the *name* of the OMeta grammar is expected before it's implementation can take place inside of the following block. Compiling this grammar to JavaScript results in a JavaScript-object `ID` containing

[1]The source of Alessandro's OMeta/JS implementation is available at github `https://github.com/alexwarth/ometa-js`

[2]Information about OMeta and OMeta/JS as well as an interactive Workspace can be found at `http://www.tinlizzie.org/ometa/`

three methods to match the specified rules. Since no parent-grammar has been specified OMeta assumes that it's base grammar `OMeta` should be the parent. Hence, a prototypal link to an object representing this base grammar is added as depicted in figure 5.2. Here we can get a quick idea of how OMeta/JS models the inheritance of different grammars by using the prototype-chain.

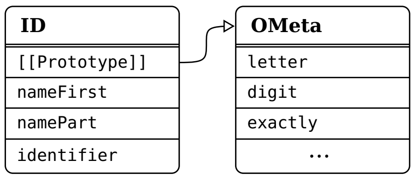


Figure 5.2: *The two grammar objects `ID` and `OMeta`*

5.1 Writing Grammars

The most important part of implementing a parser by the means of a parser generator is to write a grammar. To accomplish this task we will start off by taking a look at the different tools and syntax elements provided by OMeta.

5.1.1 Differences to PEG

OMeta supports almost all default operators that can be found in PEG. Nevertheless, new features have been introduced which conflict syntactically with existing operators. Table 5.1 illustrates the differences to the syntax as it is known from PEG (Also see original table 3.2 on page 22).

$expr_a \mid expr_b$	A pipe is used instead of a slash to express prioritized choice
`anything`	Instead of a single dot the rule `anything` is used to consume the next input without matching it
$\sim expr$	Negative lookahead uses tilde-character instead of an exclamation mark
`"rule"`	Shorthand for application of the rule `token("rule")`

Table 5.1: *Syntactical differences from OMeta to original PEG*

Character classes as they are known from regular expressions and adopted by PEG do not have an equivalent syntax in OMeta. In order to allow the pattern matching of lists (as it can be seen in section 5.1.2 on the following page), brackets had to be reserved and thus could not be used to match character classes. Regardless of the missing syntax it is still possible in OMeta/JS to implement a character range by using parametrized rules. For example the class `[a-z]` can be matched by the rule `range('a', 'z')`[3].

The use of basic PEG operators within OMeta is demonstrated in figure 5.3.

[3]The implementation of `range` can be found in appendix B.2 on page 107

```
ometa Numbers {
  number     = decimal,
  decimal    = '-'? decimalInt+ ('.' digit+)? expPart?
             | '-'? ('.' digit+) expPart?,
  decimalInt = '0' | (~'0' digit) digit*,
  expPart    = ('e' | 'E') ('+' | '-')? digit+
}
```

Figure 5.3: *A sample grammar to match decimal numbers, starting with rule* `number`

As already known from PEG, the lookahead operators assure whether the next input-token does (& positive) or does not (~ negative) match the given expression. It is important to note that thereby *no input is consumed*. In this example it gets visible how the negative lookahead operator is used to exclude the character `'0'` which would otherwise be matched by the rule `digit`. The starting point for this grammar is the rule `number`. The grammar matches all allowed decimal numbers like `-3`, `4.7`, `.6` and `6.18e-1`.

5.1.2 Pattern Matching

In contrast to PEG which only allows to match a stream of characters, OMeta is able to match a stream of *arbitrary host-language objects[27]*. There are quite a few types of objects in JavaScript for which OMeta provides a dedicated syntax as it can be seen in table 5.2.

`'c'`	Single characters can be matched using the character literal notation.
“*string*”	Matches a sequence of characters (Please note, that the string is delimited by two backticks on the left and two single quotes on the right hand side)
`1337`	Numbers can be matched natively
`[char 'o' 'o']`	The list notation allows matching a sequence of arbitrary objects inside of a list.

Table 5.2: *Syntax for matching different types of objects*

Like with PEG, the most basic terminals a parser may recognize are single characters and sequences of characters. When matching a string like “`var foo = 4`” OMeta destructs this string into it’s single characters in order to form a stream:

```
['v', 'a', 'r', ' ', 'f', 'o', 'o', ' ', '=', ' ', '4']
```

Here it gets visible why every single character has to be matched separately. If a sequence of characters like `var` is expected this has to be denoted explicitly by using the character sequence notation “`var`”. In fact, this notation is semantically equivalent to `'v' 'a' 'r'`.

In contrast to parsers, a translator has to work on structures. For this task OMeta provides a notation that can be used to match lists. Consider the task of converting a prefix notation, as it is used by Lisp, to be infix. Given the following input

```
['+', 5, ['-', 3, 8]]
```

a grammar has to recursively match the contents of the lists. Like already said, this can be performed by using the list-notation as seen in figure 5.4.

```
ometa PreToInfix {
  list     = [operator:op content:first content:second] -> [first, op, second],
  content  = list | number,
  operator = '+' | '-' | '*' | '/'
}
```

Figure 5.4: *OMeta/JS grammar to convert prefix to infix notation*

Please note that in contrast to JavaScript arrays, the elements inside of the list-notation are separated by *whitespaces* and *not commas*. To transform the output of rule `list`, semantic actions are used which will be presented in subsequent sections.

Generally speaking, every JavaScript object may be matched by utilizing predicates. For example the rule

```
expressions = anything:n ?(n.name == expr)
```

can be used to match objects like `{ name: "expr", contents: [] }`. Until now, there is no special pattern matching syntax for generic objects.

5.1.3 Semantic Predicates

Since OMeta/JS is an aggregation of OMeta and JavaScript, we can use JavaScript inside of *semantic predicates* to refine the matching process. The host-language expression inside of a semantic predicate should evaluate to a boolean value. If the resulting value is *falsy*[4], the matching of the current rule is assumed to be failed and therefore aborted, whereas a truthy value leads to a continuation of the matching-process. Figure 5.5 illustrates a grammar using predicates to differ between even and odd digits in order to match numbers like `381496`. It gets visible that the prefix-operator `?` is followed by a JavaScript expression which may, but don't necessarily has to be wrapped in parenthesis.

Of course the function `even` could have also been inlined in the semantic predicates like:

```
even = digit:d ?(parseInt(d) % 2 === 0)
```

In OMeta, the result of the last expression within a rule always is used as result of the rule. Since the semantic predicate returns a boolean value, the result of the rules `even` and `odd` is this boolean value and not the digit itself. To bypass this problem the capture operator `<...>` is used that records all input which is matched by the enclosed rules.

[4]To discriminate between the boolean value `false` and all other values that behave equally when used in conditions the terminology *falsy* (or likewise *truthy*) is used (See Douglas Crockford [3]). Falsy values include for example `false`, `undefined`, `null` the empty string `""` and `0`.

```
function even(digit) {
  return parseInt(digit) % 2 === 0;
}
ometa EvenOdd {
  even   = digit:d ?even(d),
  odd    = digit:d ?( !even(d) ),
  number = <(even odd)+ even?
           | even
           >:n -> parseInt(n)
}
```

Figure 5.5: *A grammar to match numbers with alternating even and odd digits*

Another solution to this problem would have been to add a semantic action at the end of each rule:

```
even = digit:d ?even(d)      -> d,
odd  = digit:d ?( !even(d) ) -> d,
```

Semantic predicates also can be used to include context information in matching decisions. For example it might be checked whether a variable, to which a value is about to be bound, has been declared before hand.

5.1.4 Semantic Actions

Usually, it is the job of a grammar to decide whether or not an input can be matched using the given rules. Although this is a useful information, we often need to work with the recognized input in order to extract information or to modify it. For example a stream of strings may be transformed into an intermediate representation like an abstract syntax tree. The other way around, an existing AST can be used as input to a translator to be converted to code again. For this purpose the output of each rule may be transformed using so called *semantic actions*.

There are three different ways to express semantic actions in OMeta. The first one, which is mostly used to transform the output of a rule, is denoted by the arrow-operator `->`. It may appear after each expression and is delimited by either a comma (end of rule), pipe-character (end of choice) or closing curly brace (end of grammar). Hence, it's precedence is higher than a choice, but lower than a sequence. If a programmer wants to define a semantic action to manipulate the output of a choice-expression as a whole and not for a individual option, the choice has to be wrapped in parenthesis. The implementation of a semantic action can be any *expression* of the host language.

Again, if no semantic action is given for a rule, the result of the last applied expression is used without any transformation.

The grammar in figure 5.6 is a enhancement of the grammar as seen in figure 5.3. Semantic actions are used in rule `decimal` on the right-hand side of every choice to call the JavaScript function `parseFloat(n)`. But where does the identifier `n` come from and to which value is it bound? The *capture operator*, denoted by `< ... >` captures the input used to match the inner expressions. It is very useful if we want to work with the consumed input independently of any transformations performed in the descendant rules.

```
ometa Numbers {
  number     = decimal,
  decimal    = <'-'? decimalInt+ ('.' digit+)? expPart?>:n -> parseFloat(n)
             | <'-'? ('.' digit+) expPart?>:n              -> parseFloat(n),
  decimalInt = '0' | (~'0' digit) digit*,
  expPart    = ('e' | 'E') ('+' | '-')? digit+
}
```

Figure 5.6: *A grammar to match decimals, using semantic expressions*

Using the property assignment operator `lhsExpr:id` the result of evaluating the left-hand side expression is bound to the identifier, which can be accessed in every associated host-language code like semantic actions, predicates and calls to parametrized rules.

When trying to find out in which scope an identifier can be used, we have to recall that every rule is compiled to it's own JavaScript function. Thus, every variable defined by the assignment operator can be accessed only within the corresponding rule.

expr`-> host_expr`	semantic action, transforms the result of *expr*, using an expression in the host language
`{ host_expr }`	semantic action, equivalent to the above except that it is delimited by `}`
`!host_expr`	semantic action, equivalent to the above. Commonly used in combination with parenthesis like `!( host_expr )`
`?host_expr`	semantic predicate, boolean expression evaluated while matching
rule`(expr)`	parametrized rules, `expr` is prepended to the input stream before applying *rule*
`^`*rule*	super-call operator, applies *rule* of the parent grammar object
Foreign`.`*rule*	call of a rule, residing in a foreign grammar
`<`*expr*`>`	capture-operator, memorizes and returns all input consumed by *expr*
`@<`*expr*`>`	index-capture-operator, returns an object that contains the indexes bordering the consumed input (e.g. `{ fromIdx: 3, toIdx: 7 }`)
expr`:id`	assignment operator, binds the result of *expr* to a rule-local variable `id`

Table 5.3: *Summary of OMeta syntax, additional to PEG operators*

Table 5.3 gives an overview over the different syntactical extensions OMeta offers. The first three entries of the table all represent semantic actions. They only differ in their syntax. Semantic actions are executed during the matching process according to their position within a rule. The side-effects created by semantic actions are not automatically undone by OMeta if a rule does not match in the end. The behavior of creating side-effects is visually emphasized by the exclamation prefix notation.

5.1.5 Parametrized Rules

OMeta adds even more flexibility to the grammar by allowing the use of arguments on rules, so called *parametrized rules*. Those rules behave basically the same as the one without arguments. The passed arguments are simply prepended to the input stream, before the rule is matched. Consequently, parametrized rules also support pattern matching on their parameters. Thus, the notation `rule :a :b` is only shorthand for `rule anything:a anything:b`.

Figure 5.7 shows an extension to the above grammar `EvenOdd`. Instead of defining multiple rules, one for each digit-type, there is only one parametrized rule.

```
ometa EvenOdd {
  even   :yes = digit:d ?(yes === even(digit)),
  number      = <(even(true) even(false))+ even(true)?
                | even(true)
                >:n -> parseInt(n)
}
```

Figure 5.7: *Grammar using parametrized rules*

It is important to point out that the call-arguments of parametrized rules can be any valid expressions of the host language. The result of the expression is than bound to the parameter of the invoked parametrized rule. In the first call to `even` the JavaScript value `true` is bound to the parameter `yes` and therefore further can be used in all locations where host-language is allowed.

Another example for parametrized rules is the built-in function `token(tok)`. As previously stated, OMeta can be used as “one shoe fit’s it all“ solution for the diverse compilation stages. The `token` method helps to combine “scannerful” and “scannerless” parsing[27]. The stage of lexical analysis, usually performed by a *lexer*, can be included in the parser-grammar as seen in figure 5.8.

Starting with rule `list` the grammar can be used to parse simple Lisp-like lists. The given input “`(plus 4 (minus 8 6)`” results in the tree consisting of objects, as seen in

```
ometa Lisp {
  // Lexer
  identifier = <letter+>:id      -> { type: "Id", value: id },
  number     = <digit+>:num      -> { type: "Number", value: parseInt(num) },
  punctuator = '(' | ')' |'.' | ',',

  token :tt  = spaces ( punctuator:t             ?(t == tt)      -> t
                      | (identifier | number):t  ?(t.type == tt) -> t
                      ),

  // Parser
  list       = token("(") (atom | list)+:cs token(")") -> { type: "List", content: cs },
  atom       = token("Id") | token("Number")
}
```

Figure 5.8: *Lexical analysis inside a parsing-grammar*

figure 5.9. Every object has one property `type` to specify it's kind. Additionally identifier and numbers save their values in the property `value`. Lists in turn store the contained list items in the property `content`.

```
{ type: "List", content: [
  { type: "Id", value: "plus" },
  { type: "Number", value: 4 },
  { type: "List", content: [
    { type: "Id", value: "minus" },
    { type: "Number", value: 8 },
    { type: "Number", value: 6 }]
  }]
}
```

Figure 5.9: *Result of parsing the input "`(plus 4 (minus 8 6)`"*

As it gets clearly visible the lexer is included directly in the parser grammar. Every time a token needs to be scanned the method `token` is invoked, providing the required type of token as a string. OMeta provides a special syntax for this kind of invocation since it is used pretty often. Instead of writing `token("Id")` the programmer might simply use a shorthand syntax `"Id"`. At first glance this might easily be mixed up with the matching of strings. Hence, it is important to keep in mind that strings are broken down to character sequences and therefore the syntax `"string"` has to be used.

Using the shorthand notation for `token` the parser rules may be rewritten as

```
list      = "(" (atom | list)+:cs ")" -> { type: "List", content: cs },
atom      = "Id" | "Number"
```

which is much easier to read and write.

5.1.6 Higher-Order Rules

Among the methods of the OMeta base grammar the rule `apply(rule_name)` can be found, which expects `rule_name` to be a string and invokes the rule in place. Therefore, a call to `apply("myrule")` is identical to `myrule`. Equipped with `apply` and parametrized rules it is possible to create *higher order rules* by passing rule-names as arguments. The higher order rule itself can in turn make use of `apply`. In OMeta some built-in functions are implemented that way. In appendix B.2 a pseudo implementation of the base grammar with all of it's built-in rules can be found. For example let's analyze `listOf(rule, sep)` that can be used to match a list of items. The internal implementation is close to:

```
listOf :rule :sep = apply(rule):f (token(sep) apply(rule)):r*  -> [f].concat(r)
                  | empty                                      -> []
```

Each item has to match `rule` and is delimited by the provided separator. Here we can see how the given rule is applied at all positions where a matching item is expected. Considering the grammar of figure 5.6, a call to `listOf(#decimal, ',')` could match an input string like "`1.5, 4, -8`". The usage of the dubious literal `#decimal` as first argument will be explained in the following section.

5.1.7 It’s all about Context: OMeta or JavaScript?

To write comprehensive grammars in OMeta it is necessary to distinguish between the two languages we are working with. Firstly the OMeta language and secondly the underlying host-language: JavaScript. Outside of a grammar definition only host-language code is valid. For example we are not able to write OMeta rules outside of a grammar.

```
// here only JavaScript can be written
ometa Grammar {
  // only OMeta is allowed right here
  rule   :a :b = { ... } otherRule !( ... )  -> ...,  // semantic action
  otherRule    = rule:c ?( ... ),                     // semantic predicate
  start        = rule(..., ...) "rule" apply(...)     // parametrized rule
}
// again: just JavaScript is allowed
```

Figure 5.10: *Allowed usage of JavaScript within a OMeta grammar*

The other way around, OMeta is our primary language inside of a grammar definition as it is illustrated in figure 5.10. Here we can see that host-language code is valid outside of a grammar (`... ometa Grammar {} ...`), inside of semantic predicates (`?(...)`), inside of semantic actions (`{...}`, `!(...)`and `-> ...`) and inside the call of parametrized rules (`rule(...)`). At every occurrence of an ellipsis we might implement a JavaScript expression[5].

However, there are some ambiguous notations regarding strings. For example, as we have seen, the notation “`attention`” in OMeta-language context is *not a string.* It is equivalent to calling the parametrized function `token` and passing the JavaScript string value `attention` as first argument. In contrast, appearing in host-language context “`attention`” represents a string. In order to prevent this confusing usage of double quoted string, the word literal (e.g. `#singleWord`) has been introduced to host-language context. Table 5.4 provides an overview of string literals and their semantics depending on the context of use.

	#	" "	’ ’	“ ”
OMeta	-	`token(...)`	char	char-sequence
JavaScript	single word string	string	string	-

Table 5.4: *Semantics of string-literals depending on the context*

All host-language sections inside of a grammar are compiled into individual functions which are called in the context of the grammar-object. Due to this fact, the binding of `this` in these sections is always the grammar-object itself.

[5]In the first case all JavaScript statements are also allowed

5.1.8 Grammar Inheritance

One of the most important features in OMeta is the reuse and composition of grammars. Grammars can make use of other grammars in two ways. Firstly, a grammar can inherit from another. This is expressed by using the inheritance operator `<:` followed by the grammar to inherit from. If no parent is given, the grammar implicitly inherits from the OMeta base grammar which is stored in the object `OMeta`. Thus writing `grammar Numbers {}` and `grammar Numbers <: OMeta {}` is equivalent. Of course the parent grammar needs to be compiled first before it can be extended.

For example let's extend the number grammar to additionally allow hexadecimal numbers to be matched. The implementation of this extension can be seen in figure 5.11.

```
ometa HexNumbers <: Numbers {
  range :from :to = char:x ?(from <= x && x <= to)             -> x,
  hexDigit        = digit | range('a', 'f') | range('A', 'F'),
  hex             = ''0x'' <hexDigit+>:ds                      -> parseInt(ds, 16),
  number          = hex | ^number
}
```

Figure 5.11: *Grammar that matches decimal and hexadecimal numbers*

The function `range` is introduced to check for character ranges. The implementation is identical to the one in appendix B.2. It is realized as a parametrized rule expecting two parameters - the lower as well as the upper boundary. Rule `hex` indirectly uses this function to match an arbitrary number of hex digits and returns the decimal value[6]. The last rule `number` matches either the rule `hex` or `^number`. The latter is a *super-call* to the parent-grammar applying rule `number`. In general, if a rule isn't defined in the grammar, the lookup automatically continues recursively with the parent-grammar. Given the situation that a rule with the exact same name is defined in the child grammar, just like `number`, this rule is preferred and shadows the implementation of it's parent. This behavior is similar to the one found in classical object orientation. Nevertheless, it is still possible to access the parent-rule by using the super-call operator. Figure 5.12 illustrates how OMeta/JS uses the prototypal chain to realize the inheritance of the different grammars.

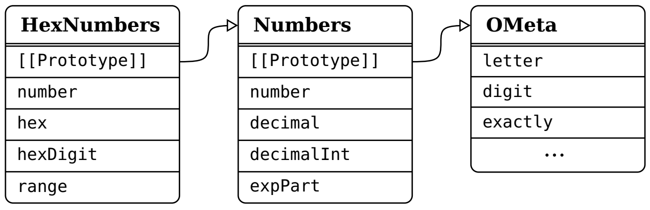


Figure 5.12: *Grammar inheritance in OMeta/JS*

Another example for using this inheritance-mechanism is to create debugging rules:

```
log :rule = ^pos:p <apply(rule)>:t !console.log("pos "+p+":", t) -> t,
next      = ^pos:p &anything:t     !console.log("pos "+p+":", t)
```

[6]The second argument of the JavaScript function call `parseInt(ds, 16)` is the radix parameter, specifying that the hexadecimal system should be used for parsing

The first rule `log` is a higher order rule expecting the rule name to apply. It can be used to log the position and input consumed by a special rule. The second rule `next` is a little easier to understand. A positive lookahead is used to log the position and the upcoming element of the input-stream without consuming it.

Since at the end of the inheritance-chain every grammar implicitly extends OMeta, it is important to know which rules are provided by this special grammar-object. For this purpose a pseudo implementation of all rules the base object offers can be found in appendix appendix B.2.

5.1.9 Foreign Rule Invocation

Building on top of existing grammars, the mechanism of inheritance is a big advance to the classical way of combining two grammars: Copying both grammars into one file and hope there are no name-clashes. But single inheritance fails when we want to include two or more grammars into a new one. This is when it comes to *foreign rules*.

Given the example we want to implement a syntax highlighter that automatically detects SQL strings within another language (for instance JavaScript). Equipped with the two grammars `JavaScript` and `SQL` this task can be accomplished pretty easy as it gets visible in figure 5.13.

```
ometa Highlighter <: JavaScript {
  string = ’"’ SQL.statement:c ’"’ -> { type: "SQLString", content: c }
         | ^string
}
```

Figure 5.13: *Example usage of foreign rules to embed SQL within JavaScript strings*

In this example we are extending the rule `string` to also match SQL strings. If the contents of the string cannot be recognized by the foreign rule `SQL.statement`, the rule falls back to the parent implementation of grammar `JavaScript`. This example illustrates how rules of other grammar objects can just be applied as if they where part of the current grammar. Nevertheless, in contrast to grammar inheritance, applying foreign rules results in a change of contexts. The input stream is just borrowed by the foreign rule and handed back when the matching has been finished [27]. Returning the flow of control is performed anyway, independent of success or error.

This procedure can be compared to switch the track for matching and continue on this track as far as we can. After the matching on that track is finished we change the lane again and return to the original grammar. Of course, just like every own rule, the track can always be a dead end.

Figure 5.14 illustrates the dependencies of the different grammars involved in the previous example.

Again, it is an important requirement that all grammar objects have to be loaded in the same environment before they may be used for inheritance or foreign rule invocation.

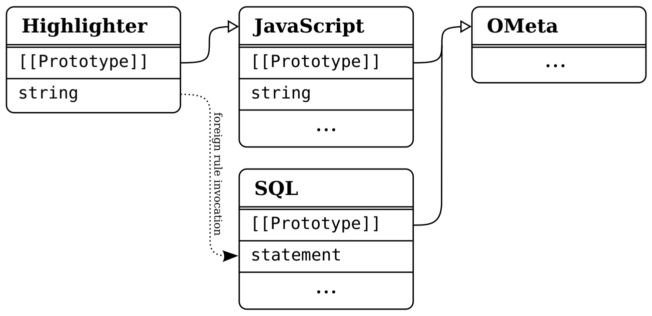


Figure 5.14: *Using foreign rules and grammar inheritance*

5.2 Using OMeta/JS

In the previous section we have learned how to write sophisticated OMeta/JS grammars. In order to be able to use the grammars in combination with the reference implementation[7] we have to do some preparations. The first step is to load all files required to compile the grammars. A list of those files, together with a short description, can be found in appendix B.1 on page 106.

Due to the large amount of files it appears reasonable to concatenate them to one file, which we may call `ometajs.js` in the remainder of this section. After all, in order to use OMeta/JS the most important three objects implemented in those files are:

`OMeta` The base grammar object every grammar inherits from.

`BSOMetaJSParser` This grammar object can be used to parse OMeta/JS grammars, starting with the rule `topLevel`.

`BSOMetaJSTranslator` This grammar object can be used to compile the tree, produced by `BSOMetaJSParser`, to JavaScript code. The starting rule for this grammar is `trans`.

Since OMeta/JS is implemented in JavaScript we may use it inside of a browser environment. In the following, we will set up OMeta in a few steps. For this purpose, we create a html file called `ometa.html` with the contents of figure 5.15.

```
<!DOCTYPE html>
<html>
  <head>
    <title>OMeta/JS</title>
    <script src="ometajs.js"></script>
    <script language="OMetaJS" id="grammar"> ... </script>
    <script>
      ...
    </script>
  </head>
  <body></body>
</html>
```

Figure 5.15: *Contents of `ometa.html`*

The necessary files to compile and execute OMeta/JS grammars are included in the first script-tag. Inside of the second script-tag with attribute `language` set to `OMetaJS` we

[7]`http://github.com/alexwarth/ometa-js`

may now add any OMeta/JS grammar like the Numbers grammar, as seen in figure 5.6. The setup of the compilation process, as described in the following, takes place within the third script-tag. First of all we need to retrieve the textual source of our grammar definition. This can be easily achieved by requesting the script-tag and reading property innerHTML.

```
var source = document.getElementById("grammar").innerHTML;
```

The next step is to parse the source, using BSOmetaJSParser which is already loaded into the global namespace by including ometajs.js. Like every OMeta object the parser provides the two methods match and matchAll. At this point only the latter one is of significance.

```
matchAll(input, rule, args?, failure?)
```

The function requires at least two arguments. The first argument, representing the input which is about to be matched by the grammar, has to be a streamable object. This only applies to strings and arrays by default. The second argument rule specifies the starting point of the matching process. The remaining arguments are optional. If the starting rule is a parametrized rule, the required arguments can be prepended to the input stream by providing an array as third argument args. Finally an optional callback function failure can be registered to handle errors.

```
var tree = BSOMetaJSParser.matchAll(source, 'topLevel');
```

The result of the matching process is an OMeta/JS language parse-tree, representing our grammar definition. In order to receive valid JavaScript code we need to translate this tree using the BSOMetaJSTranslator object and the method match. The required arguments of match are exactly the same like the ones of matchAll, with the exception that any JavaScript object may be provided as input[8]. In this case we are matching the syntax tree resulting from the previous step.

```
var grammar = BSOMetaJSTranslator.match(tree, 'trans');
```

After applying the above line, the variable grammar contains a textual representation of JavaScript code. To bring it to life and in order to actually use our grammar object we have to evaluate the JavaScript-string.

```
eval(grammar);
```

This introduces a new variable in the global scope named identical to the compiled grammar. In this case the variable Numbers will contain the desired grammar object.

5.2.1 Usage of OMeta Grammar Objects

In the previous we learned how to parse, translate and evaluate our grammar. The result is a new grammar-object introduced within the global scope. Compiling the above

[8]For instance the OMeta grammar `ometa Four { n = 4 }` will successfully match the number four by applying `Four.match(4, 'n')`

grammar will result in a Numbers object which has a prototypal link to the OMeta object. The usage of this grammar object is equivalent to the above usage of BSOMetaJSParser. A matching process for instance could look like:

```
Numbers.matchAll('1.534e-2', 'decimal')
```

The result of this expression is the numerical value 0.01534. We also might add a handler to gather more information about possible failures. In this case we just have to add a callback function as fourth argument. Currently the third one, awaiting arguments to pass to the specified rule “decimal”, is not needed and hence set to undefined.

```
Numbers.matchAll('1.5f', 'decimal', undefined, function(grammar, pos) {
  ...
})
```

The first argument provided to the callback function is the grammar object itself. The second argument indicates the position at which the error occurred.

Generally speaking, the method matchAll is used to match input which is treated as a stream, while in contrast the method match is used to recognize single objects.

5.2.2 Stateful Pattern Matching

By adding semantic actions OMeta allows not only to manipulate the results of expressions (for instance in order to create the syntax tree), but also to trigger side effects during the process of matching. This can be really useful, for instance if we want to gather information such as the occurrence of strings in order to collect them in a string table. An example of how this task can be achieved is illustrated in Figure 5.16.

```
ometa SomeParser {
  ...
  string = '"' <(~'"' char)+>:cs '"' !this.collect(cs):i -> { type: "String", id: i }
  ...
}
Parser.initialize = function() { this.strings = []; }
Parser.collect = function(string) {
  var i = this.strings.indexOf(string);
  if(i === -1)
    return this.strings.push(string) - 1;
  else
    return i;
}
```

Figure 5.16: *Using stateful pattern matching to create a string table*

In this example SomeParser makes use of semantic actions like !this.collect to push all found strings in a shared string table. Each string is only stored once - duplicates are filtered. Each string is finally replaced with an AST node containing the id (the position of the string inside of the collection), not the value itself. The callback function initialize is registered in order to prepare the parser instance before the matching can start.

5.3 Summary

OMeta/JS rendered itself to be an elegant solution for the different steps of compilation. It allows to match not only streams of characters, but also arbitrary host objects. To provide this functionality, the OMeta language shows some differences compared to common parsing expression grammars. Additionally, it equips the developer with features like left-recursion, semantic predicates, semantic actions, grammar inheritance and foreign rule invocation. We have seen how parametrized rules can be combined with the rule `apply` to create higher order rules. Nevertheless, there are some pitfalls like the difference between host-language and OMeta context. Moreover, the subtle distinction between the various string-literals is not quite easy and requires some attention.

Due to the fact that setting up OMeta requires the inclusion of many files, we have concatenated them all into one single file. This allows to work with OMeta grammars more easily. Yet, the various dependencies between the different files can be improved futher.

In the next chapter we will see how this can take place and how OMeta/JS can be used more conveniently.

Chapter 6

Extending JavaScript

Contents

In the previous chapters we made the following assumption that *a*) we are extending a language by compilation rather than introducing new design patterns or using libraries; *b*) we want to profit by the flexibilities of implementing the compiler itself in JavaScript; *c*) we use a parser generator to benefit from the abstraction a grammar can provide; *d*) we concentrate on PEG-parsers since their implementation is a mapping from grammar rules to code which is simple to understand and *e*) we choose OMeta/JS as parser generator because it offers unique extensibility mechanisms such as grammar inheritance.

Building on top of those decisions, in this chapter we are facing the problem of extending the syntax of JavaScript. For illustration a fictional language called EJS (abbr. for Extended JS or Example JS) is introduced which forms a superset of JavaScript. The goal of EJS is to hide some commonly used, verbose design patterns and frequent inconveniences behind an extended syntax. The language extension may be specified by an individual programmer or a group of developers in order to create a tailor-made domain specific language and consequently simplify every-day's work.

This chapter describes one way of how such an extension can be created within five steps without going into detailed implementations of the fictional language itself. The resulting architecture is than presented by inspecting the different packages and grammars within.

Subsequently, chapter 7 demonstrates by example how an implementation of selected EJS syntax-elements can take place utilizing the architecture as it is presented.

6.1 Five Steps to Create a Language Extension

The environment in which the language extension will be used can have major influences on design. Thus the exact context of application needs to be clarified first. Naturally two environments appear reasonable since we decided to implement the extension in JavaScript.

Most obviously, the extension could be implemented in a way that makes it possible to deliver the EJS source-code to the client's *browser* where it is compiled and evaluated. While this seems to be a good approach for development and testing as it responses immediately to modifications, it also generates a lot of overhead. Firstly, not only the source-code of the application has to be transferred but also the code for compilation. Secondly, on every load the compilation process consumes a repeated amount of time which may not be acceptable in production usage.

Both downsides can be circumvented by compiling the EJS-code once to JavaScript during deployment and only deliver the resulting JavaScript code to the client. Since most developers already use some kind of build-process[1] to build and deploy their JavaScript applications (for instance to include unit-testing, script concatenation and compression into the deployment) the compilation step is expected to be seamlessly integrated into an existing workflow. For the execution of JavaScript outside of the browser a *console-runtime environment* like Node.js is required.

Generally speaking, both environments for compilation and execution, browser and Node.js, can be targeted at the same time, because we chose JavaScript as language of implementation. Nevertheless, each individual one demands the architecture to meet certain criteria. For instance there are different requirements for file-size, performance and the way modules are linked. The latter approach of separating compile-time from run-time appears to be more promising since it does not create additional client-side overhead. In consequence we will concentrate on this strategy. After all, the final architecture can, with some effort, be translated to work in the browser as well.[2]

Notes on Node.js

Node.js is a command line interpreter based on the V8 Engine which is originated in the browser Google Chrome. Additionally it implements large portions of the CommonJS API-specification. The goal of CommonJS[3] is to provide a universal API in order to establish a standard library for JavaScript, which at the current time does not exist. Implementors of the various libraries, frameworks and runtime platforms can choose to

[1] There are different tools existing for specific environments like ant, make, rake, jake or capistrano

[2] The process of rewriting may be shorted by using a tool called node-browerify, which automatically allows the use of node-packages within the browser `https://github.com/substack/node-browserify`.

[3] `http://www.commonjs.org/`

support different levels of the API in order to make the user code transportable. Most important, Node.js supports the Modules/1.0 API which allows to encapsulate different parts of the code in modules. Every file represents a self-contained module. Instead of linking the modules via the global namespace, the required dependencies have to be explicitly stated. This is illustrated in the following example using the two modules `my` and `other`.

```
// file: my.js
var private = "World";
module.exports = {
  say_hello: function() { return "Hello " + private }
}

// file: other.js
var my = require('./my');
my.say_hello(); //=> "Hello World"
console.log(private) //=> undefined
```

The separation of modules is not as hermetic as one might expect and thus manipulation of the global namespace is still possible from within a module. For instance an assignment to a not previously declared variable still is visible in all modules. Hence, the implementation of modules in Node.js behaves similar to the pseudo code provided below[4]:

```
var m = new Module();
(function(exports, require, module) {
  // implementation
})(m.exports, m.require, m);
```

The loading and initialization of a module is only performed once. Afterwards it is cached in order to be reused by all subsequent calls to `require` which request the very same module.

6.1.1 First Step: Set up the Environment

With the decision to use Node.js as primary target environment, the first step is to prepare OMeta in order to make it compatible to be used with Node.js. As seen in the previous section, part of the philosophy of Node.js is to encapsulate the different components in isolated modules. In chapter 5 we also have seen that OMeta uses the global environment to link the different required components which leads to many interdependencies from one file to another. For instance the file `lib.js` contains a function `objectThatDelegatesTo` which behaves similar to the built-in ES5 method `Object.create` and is used in almost every other file of the OMeta implementation. Moreover it also is used by every compiled grammar leading to further dependency problems. To assure encapsulation and, in the broader sense, prevent interdependencies not clearly visible between the modules, some alternations have to be carried out:

1. OMeta has to be slightly modified to use the `module.exports` mechanism as it is used by Node.js,

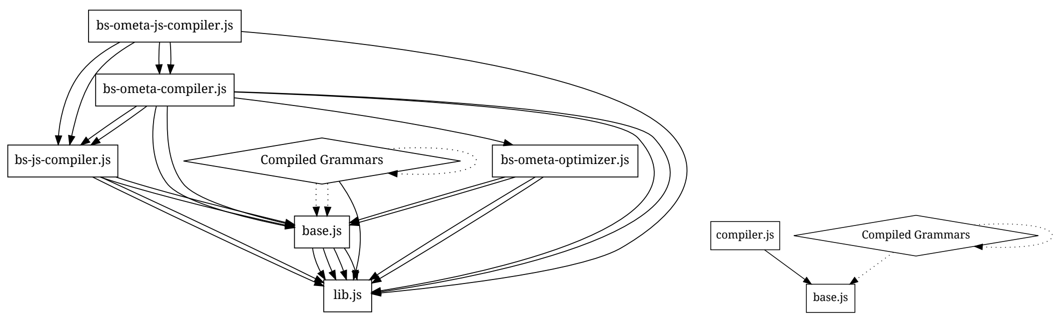
[4]The full implementation of the module-system within Node.js can be seen at `https://github.com/joyent/node/blob/master/lib/module.js`

2. The pollution of the global namespace has to be reduced:
   (a) No linking through the global namespace - instead use well defined interfaces and the export/require mechanism,
   (b) No introduction of new global objects,
   (c) No modification of global objects, such as `Object`, `Array` and `String`.

While there are some forks of OMeta/JS which mostly address the first target to make OMeta compatible with Node.js, the second one is often left behind, since it requires the larger effort of completely refactoring OMeta/JS. Nevertheless, after analyzing the dependencies a total rewrite of OMeta/JS has been performed to reduce the pollution of the global namespace. In addition, no alternation of existing global objects is required anymore. In order to achieve this the code has been restructured into two strictly separated components:

OMeta-Compiler is used to compile OMeta-grammars to JavaScript which than can be executed using the runtime

OMeta-Base represents the runtime required by the compiler and every other compiled grammar, since it is always the root of the grammar inheritance chain.



(a) *Before refactoring* **(b)** *After refactoring*

Figure 6.1: *Dependencies of the components within OMeta/JS*

The dependencies can be seen in figure 6.1. The graph on the left-hand-side (figure 6.1a) shows the usage of global variables before any refactoring has been performed. Every edge represents one global variable defined by the target and used by the source. The dotted edges of the compiled grammars symbolize a possible usage, depending on the underlying grammar. The self-reference indicates that a grammar may use another compiled grammar as parent grammar or by foreign rule invocation. By compositing the files needed for compilation into one module many dependencies could simply be restricted on the single module `compiler.js` as seen in the graph on the right-hand-side in figure 6.1b. In a second step a refactoring of the runtime and the compiler allowed to reduce the dependencies of one compiled grammar to only *a*) another compiled grammar or *b*) the runtime class `OMeta`. It is noteworthy that the compiler itself can also be seen as a simple compiled grammar.

While the original version of OMeta/JS added six methods to `String` and four methods to `Array`, it has been accomplished that neither the global object `String` nor `Array` had to be modified.

Additionally, the file-extension `.ojs` has been registered with Node.js in order to be able to compile grammars on the fly and use them as modules. This can be seen in the following example:

```
// file: parser.ojs
ometa Parser { ... }
module.exports = Parser

// file: usage.js
require('ometa'),
var Parser = require('parser.ojs');
Parser.matchAll("input string", "start")
```

The grammar specifying the example-parser is implemented in the file `parser.ojs`. Files containing OMeta/JS grammars are treated the same way as Node.js modules. This applies to both aspects implementation as well as usage. Consequently, it is possible to use the function `require` inside of a grammar to load dependencies. Also the `module.exports` mechanism can be used to specify the interface to the module. After loading OMeta/JS it is possible to directly require such a grammar file. The grammar is instantly being compiled, evaluated and the resulting parser object as specified in `module.exports` can be used right away.

6.1.2 Second Step: Write an ES5 Grammar

After successfully setting up OMeta/JS to be used in combination with Node.js the next logical step is to write a parser which recognizes JavaScript as it is specified in ES5 [5]. To be able to extend JavaScript with new syntax elements in later steps it is necessary to have a complete compilation life-cycle to build on. By using the mechanisms of grammar inheritance it is then possible to create new language-elements and insert them into the existing foundation. The life-cycle starts with the process of parsing, which is described by a grammar and mostly consists of two phases. At first, the given input has to be recognized before the single matched parts can be processed in a second step. In this section we are focusing on the former while in subsequent sections it is discussed how the intermediate representation is created and further processed.

OMeta/JS already comes with a JavaScript grammar[5] which renders itself as a good starting point, although it does not fully meet our requirements. As such the grammar should provide

1. full ECMAScript 5 compatibility and
2. a good extensibility; hence it should be
   (a) easy to understand and
   (b) modular.

The JavaScript parser at least should be able to parse every code which complies with the ECMAScript 5 specification to guarantee a compatibility to existing JavaScript code. The

[5]This grammar can be found in the original OMeta/JS implementation under the filename `bs-js-compiler.txt`

implementation included in OMeta/JS only covers a subset of JavaScript. For instance the use of the comma operator (`foo, bar`) and instantiation of properties (`new foo.Bar()`) are not possible. Newer syntax elements like property-getters and setters are also not supported.

Secondly, the ES5 grammar should be written in a way that allows an easy extension in future steps. To support a programmer in reusing rules it is considered helpful to choose rule-names akin to the specification, since the developer might be already familiar with the terminology. Additionally, rules should be expressed as simple as possible. When designing the grammar it is important to pay attention at modularity and the later reuse of rules. Changes such as a new statement or operator should only affect a small part of the grammar. Also the *principal of locality* should be applicable. Rules that are dependent should always be localized nearby if possible. This results in *clusters of rules* which exhibit certain characteristics of modules. They can be extracted without affecting other rules and most of the time offer one or more entry rules similar to an module-interface.

An early attempt of implementation has shown that the most complex rules, both for initial implementation and further extension, are the ones to allow an almost arbitrary combination of new-expressions, member-expressions and call-expressions. Figure 6.2 illustrates this problem by providing an extract of the specification (See [5, section 11.2]) directly translated to OMeta.

```
newExpr    = memberExpr
           | "new" newExpr,

memberExpr = primExpr
           | funcExpr
           | memberExpr '[' expr ']'
           | memberExpr '.' name
           | "new" memberExpr '(' args ')',

callExpr   = memberExpr '(' args ')'
           | callExpr '(' args ')'
           | callExpr '[' expr ']'
           | callExpr '.' name
```

Figure 6.2: *Ometa grammar representing an extract of the ES5 specification*

It gets visible that some redundancy has been introduced by the technical committee to be able to combine the different postfix operators (i.e. `expr[]`, `expr()` and `expr.name`) and prefix operators (i.e. `new expr`). At the same time the left-associativity of the operations is preserved. It becomes clearly visible that an extension based on this grammar leads to complexity which may be hard to master. The solution presented here uses OMeta's parametrized rules to solve both problems. Firstly, it removes the redundancy and secondly it allows an easy extension without the need to rewrite the participated rules.

In order to achieve this, we introduce a new type of expression, the *access-expression* as it can be seen in figure 6.3. It's only purpose is to be prefixed (for example by the new-expression) or to be postfixed. While the prefix extension is the same as in the specification (except that `new` only appears once) the postfixes can be specified each in a separate rule and than be added to `accessExpr`. Adding a novel syntax for a postfix operator such as `expr<...>` to an inheriting grammar can be easily accomplished now:

```
// prefixes
newExpr       = "new" newExpr
              | accessExpr,

// helper to add postfixes
accessExpr    = accessExpr:p callExpr(p)
              | accessExpr:p memberExpr(p)
              | primExpr,

// postfixes
callExpr   :p = '(' args ')',

memberExpr :p = '[' expr ']'
              | '.' name
```

Figure 6.3: *Restructured version of the ES5 extract*

```
accessExpr   = accessExpr:p angleExpr(p)
             | ^accessExpr,

angleExpr :p = '<' expr '>'
```

Passing the base-expression as argument `p` to the postfix expression has the advantage of being able to maintain the left-associativity later-on when creating AST-nodes.

Furthermore, when implementing a grammar which may be extended later on it is important to keep it as simple as possible. In the implementation at hand this simplification sometimes had to be performed at the cost of loosing some early errors, since the grammar allows at a *minimum* to detect ES5 code. For instance the redundancy of rules induced by $ExpressionNoIn$ (See [5, section 11.14]) has been removed by allowing the false positive parsing of `for(foo in bar; foo < bar; foo++) {}`. It is noteworthy that it is possible to implement the behavior as specified in [5] by using parametrized rules with a boolean parameter `noIn` to indicate whether the `in` operator is allowed or not. But again, this adds an undesired and not acceptable layer of complexity.

6.1.3 Third Step: Specify the Format of the AST

Since OMeta/JS supports semantic actions we are able to almost freely specify the format of the AST. For every rule that has successfully recognized a certain part of the input it is possible for us to compose the result as it is returned by the rule. The following example illustrates how the recognition of the input is performed on the left-hand-side of the grammar, while the processing of the output is placed on the right-hand-side.

```
stmt = "if" "(" expr:c ")" stmt:t "else" stmt:f   -> IfStmt(c, t, f)
```

If no intermediate steps such as analyzis or optimization are needed, the semantic actions also could be used to perform a syntax-directed translation (see [1, section 2.3]) which does not necessarily require an abstract syntax tree. This strategy can be applied for easy transformations only, since it does not allow to analyze the programs structure in order to create the correct output. For instance it could be difficult to implement an implicit return for the last statement of a function that way[6].

[6]An example implementation of functions expressions with implicit returns can be found in 7.3 on page 82

Hence, in order to support an arbitrary count of translations an AST is used as intermediate representation. As already seen in section 2.3 on page 12, the AST is mostly traversed for two purposes. Firstly to statically analyze the structure of a program and secondly to modify it. Furthermore, generating code can be seen as special modification of the AST which discards the AST structure to return the final result. The format of the AST is characterized by *a*) the structure of the data and *b*) the naming of the different types of nodes and their interface.

For the most part, the former affects only the implementation of tree-walkers and should be transparent to developers. Whereas the second point is most important for grammar writing developers, since they have to create and traverse the AST and therefore always have to be aware of the node-naming. Figure 6.4 illustrates how the format of the AST can serve as an protocol-like interface between the different stages of the compilation.

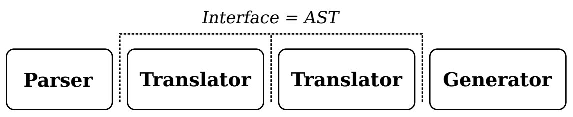


Figure 6.4: *AST-format as interfaces between compilation stages*

The data-structure of the AST has to be consistent over the different stages of compilation. The naming in contrast can vary, since each compilation step may add new node-types or remove node-types to concentrate on a smaller set. Each stage requires the AST format to meet certain criteria. On the one hand the nodes, from which the parser constructs the AST, should be created easily without much redundancy of code. On the other hand traversal and modification of the tree should be as simple as possible.

In the remainder three different data-structures will be compared. Afterwards, the traversal of the AST is discussed to finally explain how ASTs are assembled from nodes and translated back to code.

```
// original source code
var foo = 4;

// S-Expression AST
["VarDecl", "var"
  ["VarBinding", "foo",
    ["Number", 4]]

// Object-Notation AST
{ type: "VarDecl", kind: "var", declarations: [
  { type: "VarBinding", id: "foo", init:
    { type: "Number", value: 4 }}]}

// JsonML AST
["VarDecl", { kind: "var" },
  ["VarBinding", { id: "foo" },
    ["Number", { value: 4 }]]]
```

Figure 6.5: *Example of three AST-formats representing* `var foo = 4`

To allow serialization and a seamless integration into JavaScript programs[7], all three AST-formats as seen in figure 6.5 are formatted in JSON[8]. For reasons of readability

[7]This also includes OMeta/JS grammars, since JavaScript code can be embedded in every grammar
[8]For details on JSON see RFC 4627, July 2006 (`http://www.ietf.org/rfc/rfc4627`)

property-names are not quoted as strings in this example.

As it can be seen, the *S-Expression* notation is the most easy to read and to write manually, since it only consists of nested arrays. All information of a node is simply stored as entry inside of an array. The first element always specifies the type of a node, whereas all following elements are considered to be child-nodes. This data-structure is used within the OMeta/JS implementation and is widely known, since Lisp programs are described that way. Yet the traversal of such an AST in OMeta/JS can be troublesome. One has to exactly know which elements of the array are child-nodes and thus need to be traversed recursively and which are just simple information and hence don't need to be traversed. In addition, the position of an object within the node often also implies semantic information about the relation of the object to the node. For example `var node = [“IfStmt”, true, false]` is not the same as `var node = [“IfStmt”, false, true]`, if we assume that the first child represents the *test-expression* (`node[1]`) and the second one the *true-statement* (`node[2]`) which is evaluated if the test is truthy. Consequently, adding new information to a node of this type always results in also adapting all translators it is used within. This also makes it difficult to add meta-information like the position in the source code, whitespace information or comments.

	S-Expressions	Object-Notation	JsonML
Easy to read and write	✓	✗	✓
No implicit semantics	✗	✓	✓
Easy access to information	✗	✓	✓
Easy to traverse	✓	✗	✓
Can contain meta-information	✗	✓	✓

Table 6.1: *Comparison of the different AST data formats*

The *Object-Notation* in this example is inspired by the SpiderMonkey Parser API [16] which describes interfaces to the different node-types as they are created by the SpiderMonkey JavaScript engine. The API is build into Mozilla Firefox 7.0 and can be used to inspect and manipulate JavaScript programs. In contrast to S-Expressions the object-notation allows to directly access certain children by name. For example applying `node.test` on an if-statement will return the associated test-expression. The position of child-nodes does not imply semantic informations, since this is explicitly expressed as property name. As a result this data-structure appears to be more robust to changes and meta information can be added and removed without affecting the remaining behavior. Nevertheless, there is one big downside of this structure: OMeta/JS currently does not support a way to elegantly match generic objects and their properties. Details of this problem and a possible solution is presented in section B.3. Since object pattern matching is not yet implemented, the object-notation is still considered highly problematic. In addition it can be difficult to write a correct syntax tree by hand as well as to read a complex AST.

The *JsonML* format[9] has been developed to convert any XML into JSON-format and vice versa without losses. It builds on the same foundation as S-Expression, but adds an optional attributes object as first child. This object can be used to store any values, which are not considered to be AST-nodes themselves. This small structural change makes it a lot easier to automatically traverse the children of a node since every child has to be an AST-node. In addition, all attributes can be accessed by name and therefore can be added or modified with the same simplicity as nodes of the Object-Notation. Depending on the magnitude of used attributes, the readability can be ranked between S-Expressions and Object-Notation. The JsonML format shares the same problems with S-Expressions when it comes to semantic assumptions derived from an element's position. Nevertheless, those problems occur a little less often, since some information can be extracted to explicitly named attributes.

In summary, the JsonML format renders itself to be a reasonable compromise of the two other alternatives featuring the individual strengths and diminishing their shortcomings. On top, this data-structure appears to be the most easy to traverse, since it distinguishes AST-nodes and other attributes in a well-defined manner. These advantages make JsonML the natural AST-format for our requirements.

6.1.4 Fourth Step: Traverse the AST and Generate Code

Having chosen JsonML as the data-structure, the next decision to make is how the AST should be traversed. When implementing a translation there basically are two different strategies. The traversal of the tree can either be achieved by

1. using a grammar or
2. implementing a standalone walker utilizing the visitor pattern.

While the first approach can be implemented in OMeta itself, the second one requires a new piece of software, the walker or visitor. Both strategies have in common that they require at least one rule to be implemented for each individual node-type which needs to be translated.

After an experimental implementation of both approaches I decided to pursue the first one. It appears to be more consequent to implement both phases *parsing* and *translation* in OMeta instead of making use of a totally different tool-chain for translation than for parsing. Additionally, the complete `ES5Translator` implementation written in OMeta is just about 60 lines of code, whereas the visitor based implementation was about three times that size.

There are two reasons for the increased complexity of the visitor based implementation. Firstly, a translator written in OMeta allows to specify not only the node-type, but any arbitrary pattern that can be matched before a certain semantic action is performed. Using imperative programming some effort is needed to mimic these pattern matching capabilities. For example in order to differentiate between the two structures

```
["TryStmt", {}, try_block, catch_expr, catch_block, finally_block?]
```

[9]Further information on JsonML can be found at `http://jsonml.org/`

and

```
["TryStmt", {}, try_block, finally_block]
```

some conditional branches and testing conditions have to be written by hand which are otherwise generated automatically from the grammar. This causes the grammar-based approach to appear more structured at first sight, while the visitor implementation quickly becomes a diffuse mixture of matching and translation inside of the different rule implementations.

Secondly, as seen in chapter 3 the layer of abstraction introduced by a grammar makes it more easy for the developer to instantly recognize important semantics which otherwise are hidden under the verbose syntax necessary in a manual implementation[10]. Hence, redundant code can be reduced at a great level by automatically generating it from a grammar and consequently making both reading and writing the code less of a burden.

Nevertheless, in both approaches it is possible for one translator to inherit behavior from another one. Written in OMeta, the translator can use the mechanism of grammar inheritance. The rules of a visitor implementation in turn can be reused by utilizing JavaScript's prototypal inheritance.

The OMeta/JS Translator

In the following we will see how the implementation of an OMeta translator can take place. Each translation of an abstract syntax tree consists of the four steps *a*) Find the right translation rule for the current node; *b*) traverse the children recursively; *c*) combine the results with additional information like node-attributes or context and finally *d*) produce a result and return it. The implementation of the translator which is presented here is mainly inspired by the original translator as it is delivered with OMeta/JS. Nevertheless, some changes had to be applied in order to make it suitable. Firstly, the compatibility with JsonML had to be achieved, since this has been chosen as data structure of the AST. Secondly, all node types as they are created by the `ES5Parser` have to be recognized and successfully translated.

In order to find the appropriate rule to apply for the current node the grammar uses the node's type string. According to the JsonML format the type string always can be found as first element of the array which represents the AST-node. Considering the example-node `[“VarDecl”, { kind: “var”}, ...]` the correct rule to apply is `VarDecl`. The second entry contains the attribute object which does not need to be translated. All following elements are considered to be child-nodes and can explicitly be traversed by applying the rule `walk` to each child. Likewise, if many child-nodes are expected, `walk+` can be called which results in an array containing the traversed children. Based on this insights an intuitive implementation to translate the AST (as seen in figure 6.5) back to JavaScript code can be found in figure 6.6.

This way of implementation has the major downside of being completely aware of the underlying data structure. Additionally only one rule is specified which makes the grammar difficult to extend in future steps. To solve both problems the final implementation makes use of a parent `JsonMLWalker`-grammar which brings along all basic functionality

[10]Interesting thoughts about problem solving by the means of abstraction can be found in [23]

```
ometa Translator {
  walk = ['Number'     :attr]                -> attr.value
       | ['VarDecl'    :attr walk+:bindings] -> ["var ", bindings.join(', ')].join('')
       | ['VarBinding' :attr walk:init]      -> [attr.name, "=",init].join('')
       | ['VarBinding' :attr]                -> attr.name
}
```

Figure 6.6: *Implementation of a translator for variable declarations based on OMeta/JS pattern matching facilities*

to traverse the AST-nodes. In theory, this makes it more easy to exchange the underlying data-structure by just inheriting from another generic translator, although this may only work for array-based data-structures such as S-Expressions and JsonML. This limitation derives from the fact that the matching of the node-structure and traversal of the children is still performed by explicitly pattern matching on the array.

```
ometa Translator <: JsonMLWalker {
  Number     :n = empty            -> n.value(),
  VarDecl    :n = walk+:bindings   -> ["var ", bindings.join(', ')].join(''),
  VarBinding :n = walk:init        -> [n.name(), "=",init].join('')
                | empty            -> n.name()
}
Translator.force_rules = true
```

Figure 6.7: *Implementation of a translator for variable declarations utilizing a parent `JsonMLWalker`-grammar*

The grammar as seen in figure 6.6 can be rewritten to make use of the `JsonMLWalker`. The result of this refactoring can be seen in figure 6.7 with `n` not being the attribute-object (since the grammar is decoupled from JsonML) but the node object itself. The inherited rule `walk` still is the starting point for this grammar. The configuration `force_rules` can be used to define how to handle the traversal of nodes without a specified rule. If set to `false`, the node is left untranslated but it's child-nodes are traversed recursively. If set to `true`, an error is thrown.

We may notice that the code is visually separated in three columns. The first one shows the node-types, the second one represents the structure of each node (and also recursively specifies the traversal of the children by applying `walk`) and the third one indicates the transformations to perform on the nodes.

Furthermore, the node implements an interface to access it's attributes (for instance `n.value()`) and some additional methods. This methods can be used within semantic predicates to further confine the matching of the node such as `?n.is('kind', 'var')`. An implementation comparable to the one above but written in JavaScript and based on the visitor pattern can be found in appendix A.2.

Node Constructors

To simplify the process of translation all nodes have the same structure regardless if there are attributes for a special type of nodes or not. The structure is always as follows:

```
[node-type, attributes, children ...]
```

Nodes with the same structure and behavior are grouped together as node-types. They may be compared to classes as known from classical object orientation. Instances of a node-type are created by node constructors[11], even though the data structure is simple enough to be written by hand. The use of constructors offers two advantages:

1. The underlying data-structure may be changed without the need to refactor the parsers and translators.
2. Mistypings such as `[“VarDeccl”, {}, ...]` are reported as early errors, since `VarDeccl(...)` is not a function.

The constructors expect a set of parameters describing the configuration of the node to be assembled. By default each of the given arguments is appended as individual child to the node. This behavior can be customized as it is demonstrated in the following.

The node constructors themselves are created by a higher order function which can be compared to a factory or a meta-constructor. In order to create a node constructor two obligatory and one optional parameter can be specified as seen in figure 6.8.

```
function NodeType(node-type, attributes, constructor?)
```

Figure 6.8: *Signature of the node constructor factory*

The first parameter is expected to be a string specifying the type of nodes to create the constructor for. The second parameter has to be an object and describes the set of attributes and their default values that every node of this type should be equipped with. For every given attribute a combined getter and setter method is added to the node-instances. These attribute-accessors can be used to access and manipulate the attributes of the node without having to be aware of the attributes-object or the data-structure. The last and optional parameter can be used to specify a callback which is invoked right after the construction of a node has been finished and the attribute-methods have been added. The callback itself receives all arguments that are supplied during the constructor call. The binding of `this` is set to the newly created node. It can be used to override the default behavior and explicitly process the arguments in order to add the child-nodes by hand. For example a constructor for number literals nodes can be created as seen in figure 6.9 on the following page.

In this example `this.value` is the generated accessor method for the attribute `value`. Likewise, using the method `type` the type of the number-node which by default has the value `“decimal”` can be changed. The callback function is used to set the value-attribute instead of appending the arguments as child-nodes.

Additional to the attribute accessors every node is equipped with methods which can be used in semantic predicates to refine the process of matching. Some of the most important methods can be found in table 6.2 on the next page.

Utilizing this methods the grammar of figure 6.7 can be enhanced to differentiate between hexadecimal and decimal numbers. The rewritten rule `Number` can be seen in figure 6.10 on page 61.

[11]The SpiderMonkey API offers almost the same concept but calls it “Builder objects” [16]

```
var Number = NodeType("NumberLiteral", {
  type: "decimal",
  value: undefined
}, function(value) {
  this.value(value);
});

var number_7 = Number(7);
console.log(number_7);         //=> ["NumberLiteral", { type: "decimal", value: 7 }]
console.log(number_7.type());  //=> "decimal"
console.log(number_7.value()); //=> 7

number_7.value(42);
console.log(number_7.value())  //=> 42

var number_fa = Number(0xfa).type('hex')
console.log(number_fa);        //=> ["NumberLiteral", { type: "hex", value: 250 }]
```

Figure 6.9: *Creating a constructor for nodes with the type* `Number`

`n.hasType(type)`	Checks whether node `n` has the provided type.
`n.has(attr)`	Returns true if the given attribute can be found in node `n`
`n.is(attr, val)`	Compares the value of `attr` in `n` with the one provided
`n.not(attr, val)`	Same as `n.is` but negates the result

Table 6.2: *Excerpt of methods which are added to each node*

Limitations of the AST

The implementation of the AST and it's traversal as presented here isn't flawless and comes with certain limitations. For instance the process of matching a subtree and the translation of child-nodes are unified. This makes a traversal of the child-nodes difficult when only trying to match subtrees or collect information. Especially the latter can be achieved more easily by using a special query language like the one provided by jQuery in order to find elements in the DOM. For example it may be quite tedious to write a grammar that collects all variable declarations for each function but not the ones of nested functions. The same task can be achieved very easily using a jQuery like approach:

```
$('VarDecl').each(function(decl) { decl.parent('Function').add(decl) })
```

As a future work it is possible that these querying capabilities might be added as methods to the nodes. In consequence they could be used inside of semantic actions and semantic predicates and therefore combine the strength of both abstractions: recursive grammars and iterative filters.

```
  Number     :n = ?n.is('type', 'hex')     -> ("0x" + n.value().toString(16))
                | empty                    -> n.value()
```

Figure 6.10: *Using node-methods to refine the matching of rule* `Number`

Generating Code

The `JsonMLWalker` as it is presented above can be used to translate one abstract syntax tree into another. It also can be used to generate JavaScript code - the target language of our compilation process. The code in turn can be executed by any JavaScript interpreter or browser. Of course, parsing JavaScript to receive an AST and translating it back to the same code does not make any sense if there aren't intermediate steps like optimization or compression. But since we are planning for extension, this step is necessary to take. Figure 6.7 already illustrated partially how the generation of code can be achieved. Instead of modifying the nodes or creating new ones, semantic actions are used to generate a string value for each node. The semantic action is applied after the child nodes have been translated. Therefore, every node just has to merge the parts which are recursively produced by the child nodes. Finally, the root node can return the complete code which should be semantically the same as the input. For reasons of simplicity we accept that input and output are not identical down to a single character. This differences are the result of lost information like whitespaces and comments which could be preserved with some additional effort.

6.1.5 Fifth Step: Start Extending

In the previous steps everything has been prearranged to finally extend JavaScript. To summarize, a JavaScript parser has been written that makes use of node constructors in order to create an abstract syntax tree with JsonML as the underlying data structure. Afterwards, the design of a tree-visitor has been discussed which takes the AST as input and translates it back to JavaScript code. With this utilities at hand it is now possible for us to put together the pieces and start extending the syntactical frontend of JavaScript. In general, there are four ways to extend the language. Each characterized by both increasing complexity and power of expressiveness.

The first and easiest way to implement an extension is to use syntax directed translations as seen in figure 6.11.

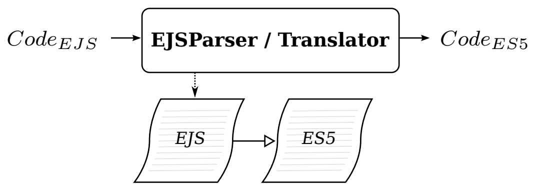


Figure 6.11: *Extending the language using syntax directed translations*

A necessary prerequisite is that the `ES5Parser` does not create an AST as output but directly translates the code back into a string. This causes the parser to be a translator (and code generator) at the same time. An extension could inherit from this parser grammar and add new custom rules which themselves have to produce valid JavaScript code. Following this approach, a transformation from EJS to JavaScript can be performed in one single step which may be denoted as $Code_{EJS} \rightarrow Code_{ES5}$. Despite of it's simplicity this strategy is not applicable due to our decision to use an AST as intermediate representation.

As discussed above, the usage of an AST allows us to add a wide range of optimization and transformation steps in between before the code is translated back to JavaScript. Additionally, with an intermediate representation it gets possible to realize more complex extensions since the code can be analyzed more easily.

The second approach embraces this concept as seen in figure 6.12. For notational convenience we may also write $Code_{EJS} \rightarrow AST_{ES5} \rightarrow Code_{ES5}$.

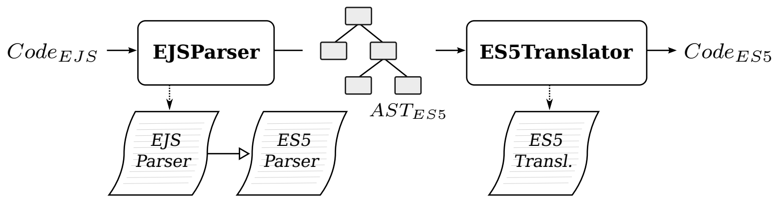


Figure 6.12: *Extending the language by creating an ES5 AST*

Following this approach, the `EJSParser` still inherits from `ES5Parser` but now parses the input string and reduces all syntactic sugar to create a valid AST_{ES5}. This abstract syntax tree can be further optimized and finally translated to code by the already existing `ES5Translator`. In consequence, the already existing tool-chain based on the AST_{ES5} can be reused. The flexibility of this two step process is sufficient to be able to create most extensions.

Nevertheless, it is possible to introduce a separate AST format for the extended language. The new AST format AST_{EJS} does not differ in structure but adds new node types for novel syntax elements, hence the different subscript. Using a separate format for the syntax tree defers the responsibility of converting EJS to ES5 from the parser to the translator. The parser only has to recognize the new syntax but does not need to "understand" it's semantics. The second task of the extension, that is converting the tree to valid JavaScript, is now part of the translator. Figure 6.13 shows the third way to extend a language, which we may also refer to as $Code_{EJS} \rightarrow AST_{EJS} \rightarrow Code_{ES5}$.

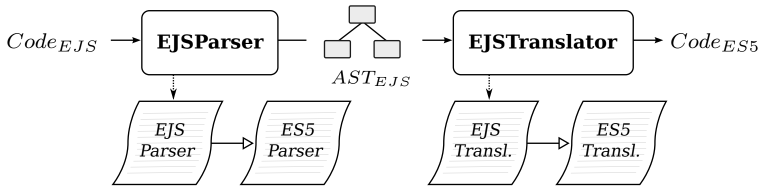


Figure 6.13: *Extending the language by creating an intermediate AST_{EJS} format*

A new translator, called `EJSTranslator` inherits from `ES5Translator` in order to directly generate the string representation. With this approach we gain the possibility of analyzing the high level AST (including the new syntactic elements). This can be rated positive, since it is always valuable for a later analysis to preserve as much high level information as possible. At the same time, this approach is similar to the first one, since we loose the compatibility to tools which are comfortable with theAST_{ES5} format.

The most powerful but also most the complex approach takes the concept of deferring the conversion to the translator one step further. The solution presented in figure 6.14 is similar to the second approach in a way that it allows the complete reuse of the ES5 backend. To achieve this an additional step of translation has to be introduced which converts all EJS node-types into appropriate ES5 subtrees. This last approach further is referred to as $Code_{EJS} \rightarrow AST_{EJS} \rightarrow AST_{ES5} \rightarrow Code_{ES5}$.

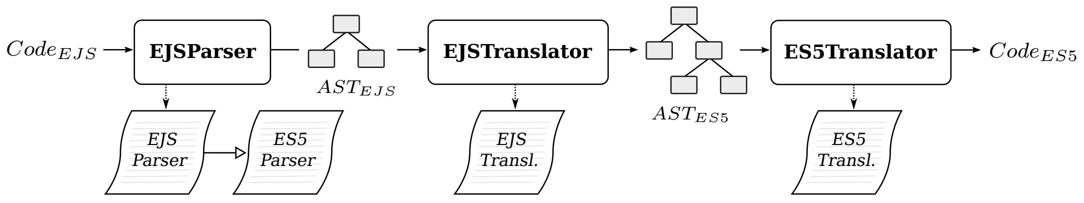


Figure 6.14: *Extending the language by additionally translating AST_{EJS} to AST_{ES5}*

As already said, the two step process most of the times is sufficient. Since we are only adding “syntactic sugar” to the language new elements can simply be mapped to JavaScript. Nevertheless, sometimes it is more convenient to create new node types and delay the translation to a dedicated step. This consequently separates the recognition of code from the further processing by the price of an additional translation-pass. Also analysis can take place at two different levels (AST_{EJS} and AST_{ES5}). As a result, the approaches two and four are supported by the architecture and a programmer implementing an extension may decide which one to use.

Example Extension

Let’s place ourselves in the situation that we want to implement a web-server based on an event loop (if we are more attracted by games we may also implement a game loop). In JavaScript, as in most languages based on C-style syntax we may utilize either of the statements `for(;;){...}` (pronounced “for ever”) or `while(true){...}`. Both statements will repeatedly execute the statements included in their bodies until the end of all days. However, both statements do not directly express what they are supposed to do - create an endless *loop*. As a consequence we will add a small extension to the JavaScript language and add a `loop`-statement.

Since this extension does not require further steps of analysis or an extensive translation process we will follow the two step extension approach. Hence, the first thing to do is to create a new parser which inherits from `ES5Parser` as seen in figure 6.15.

The extension can be broken down to the steps *a*) extend the existing rule `keyword` and add `loop` as a new keyword; *b*) create a new rule `loop` which matches the loop keyword and any subsequent statement; *c*) use a semantic action to create a new valid ES5 AST-node and thereby desugar the `loop`-statement into `for(;;)` to finally *d*) extend the rule `stmt` to register `loop` as novel statement.

The resulting parser can now be used in combination with the `ES5Translator` to compile any code containing the loop-statement.

```
ometa EJSParser <: ES5Parser {
  keyword = ‘‘loop’’ | ^keyword,
  loop    = "loop" stmt:s           -> ForStmt(undefined, undefined, undefined, s),
  stmt    = loop | ^stmt
}
```

Figure 6.15: *Adding `loop`-statements to JavaScript*

```
var es5_tree = ejs.parse("loop { process_events(); }");
var result   = es5.translate(es5_tree);
console.log(result); //=> "for(;;) { process_events(); }"
```

Figure 6.16: *Using the EJSParser to make use of the new `loop`-statement*

The usage can be seen in figure 6.16 with the parser being part of the ejs-package and the translator being included in the es5-package.

6.2 The Architecture

In the previous section we have seen how an extension to JavaScript can be created by following five steps. This section will give some more information about the architecture resulting from that process.

To gain an impression of the architecture the first view that will be discussed is the "package view". Following the principle of *separation of concerns* all modules, whether they are grammars or not, are grouped into packages. Each one defines a clean interface. This encapsulation ensures a better maintenance and allows changes of the implementation within a package without affecting other packages.

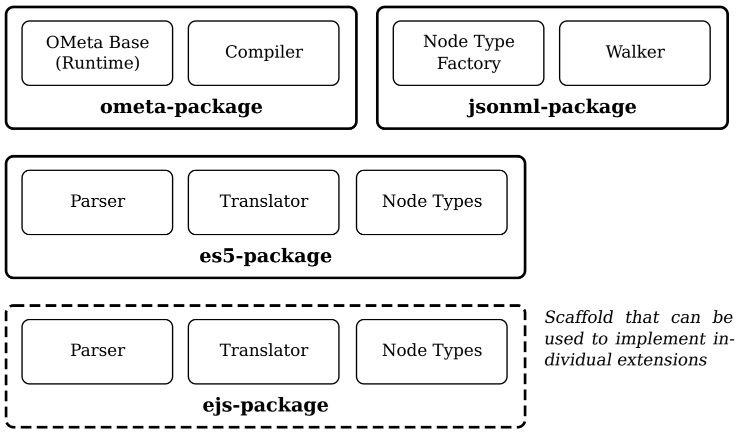


Figure 6.17: *Packages from the architectural point-of-view*

The packages can be grouped into three levels of implementation as seen in figure 6.17. The first level consists of the ometa-package and the jsonml-package. Those two packages can be distributed independently of the rest and may be used as foundation for every compiler-project that is based on OMeta/JS with JsonML as data structure. Moreover, the ometa-package also can be used standalone if no pre-configured JsonML-translator or node-constructors are necessary.

The second level is the result of "Second Step: Write an ES5 Grammar" and "Fourth Step: Traverse the AST and Generate Code". The es5-package depends on the first level in many ways, which can be seen more detailed in the following subsection 6.3. Hence it only can be used in combination with both packages ometa and jsonml.

The extension of JavaScript finally starts with the third level containing the ejs-package. The package builds on es5 and contains the results from "Fifth Step: Start Extending" in order to provide language extensions. It is the third level at which a developer starts implementing a new language-extension. The ejs-package presented here only serves as example and is not necessarily part of an individual implementation.

6.2.1 The OMeta Package

The *ometa-package* basically consists of two components. First, a compiler which is needed to compile an OMeta/JS grammar to JavaScript. The second component is the `OMeta` base class every OMeta parser inherits from. The base is sometimes also referred to as "OMeta-runtime", since every OMeta parser requires it at runtime to be executed.

Method	Description
`.compile(grammar)`	Expects the arguments `grammar` to be a string, describing an OMeta/JS grammar and returns compiled JavaScript code
`.run(compiled-source)`	Executes code as it is returned by `.compile` and returns what has been specified as `module.exports`

Table 6.3: *Interface of the ometa-package*

```
// All required dependencies
var ParentGrammar = require('./parent_grammar.ojs'),
    tools         = require('tools'),
    ...;

// Implementation of the grammar itself
ometa Grammar <: ParentGrammar {
  ...
}

// Specification of the interface
module.exports = {
  parse: function(input) { return Grammar.matchAll(input, 'start'); }
}
```

Figure 6.18: *Usage pattern of standalone grammar modules*

The ometa-package presents an interface which can be used by all other packages to *compile* and *evaluate* OMeta/JS grammars (Table 6.3). In addition, the `.ojs` extension is registered with Node.js. This allows to use OMeta/JS grammar-files the same way as common Node.js packages (Also see subsection 6.1.1). To avoid multiple, redundant compilation passes over the same grammar-module the compiled results are cached in temporary files. The standalone usage of grammar-modules keeps the amount of files within a package at a minimum, since no additional file is required to control the life-cycle of a parser-instance. Instead, akin to the usual implementation of a Node.js module, the dependencies and interfaces to a grammar are expressed in the grammar itself. In consequence, two or more grammars can be grouped within one module and therefore be encapsulated behind one specified interface. During the implementation of the framework the pattern emerged as it can be seen in figure 6.18. It is used within all standalone grammars.

One may notice that there is no way for a grammar which is placed inside of another module to inherit from `Grammar` because only the method `parse` is included in the interface. To support an extension it is possible to rewrite the interface definition as follows.

```
Grammar.parse = function(input) { return Grammar.matchAll(input, 'start'); }
module.exports = Grammar
```

6.2.2 The JsonML Package

The second package that can be found in the first level is the *jsonml-package*. The package provides all functionality that is required to work with an AST based on the JsonML data-structure.

Method / Property	Description
`.factory(type, attr, constr?)`	Factory function to create constructor functions, which can be used to instantiate the respective node-type.
`.walker`	Instance of the walker from which an OMeta/JS grammar might inherit.

Table 6.4: *Interface of the jsonml-package*

Firstly, it contains a factory which can be used to create node-constructors for each node type. These constructors in turn are mostly used by parsers in order to create node-instances and thereby build the abstract syntax tree.

Secondly, the package provides a walker-implementation that is written in OMeta/JS. All subsequent translators inherit from this grammar in order to reuse the traversal functionality provided by the walker. Obviously, the jsonml-package depends on the ometa-package in a way that it uses the ojs-extension mechanism described above.

Details about the walker implementation and AST design have been discussed in 6.1.4. The following example illustrates the usage of the jsonml-package in order to get an overview about the features as they are contributed by the package.

The first part of the example (figure 6.19) shows how node constructors are created by the factory before a parser can make use of them within semantic actions.

The second part of the example as seen in figure 6.20 illustrates a translator that is able to translate abstract syntax trees created by the above parser. To traverse through every single nodes of the AST the rule `walk` is provided by the parent grammar `JsonMLWalker`.

```
// parser.ojs
var Number = Factory("Number", { value: undefined }, function(digits) {
  this.value(parseInt(digits))
})
var ArrayExpr = Factory("ArrayExpr", {}, function(elements) {
  this.appendAll(elements);
});

ometa Parser {
  primaryExpr = number | arrayExpr,
  number      = <digit+>:ds                            -> Number(ds),
  arrayExpr   = "[" listOf("primaryExpr", ","):els "]" -> ArrayExpr(els)
}
```

Figure 6.19: *Example of a parser that utilizes the jsonml-package*

```
// translator.ojs
ometa Translator <: JsonMLWalker {
  Number    :n = empty                                 -> n.value().toString(),
  ArrayExpr :n = walk*:els                             -> ("[" + els.join(',') + "]")
}
```

Figure 6.20: *Continuation of 6.19. Translator using the jsonml-package*

We also may notice that the translator makes use of the method `value()`. This method is added automatically by the node-constructor, since it is part of the attribute-object provided to the factory (See definition of `Number` in figure 6.19).

6.2.3 The ES5 Package

Placed on top of the first level the *es5-package* provides everything that is needed to successfully parse JavaScript code, create an intermediate representation and translate it back to code. The package consists of the two grammar-modules `ES5Parser` and `ES5Translator` as well as one auxiliary module that contains all node-constructor definitions.

The package can be used in four ways. Firstly, the language which is recognized by the parser may be extended. This can be achieved by adding new syntactic constructs to the parser and reusing the translator (This is how the example language EJS is implemented). Secondly, a pretty printer could be created by re-implementing the translator. Thirdly, intermediate translations may be added in order to optimize or compress the source code. Finally, the parser may be reused to generate the abstract syntax tree of a piece of source code in order to statically analyze the code. This option may be used for instance to provide code assistance or to verify the code with tools like “linters”[12].

In order to allow grammar inheritance the interface as seen in table 6.5 offers access to both grammar objects. The same applies to the collection of node constructors, which may be extended in further steps. The naming of the node constructors is inspired mainly by Spidermonkey’s Parser-API [16] but it is not identical. Some names have been shortened and a naming convention for expressions and statements has been introduced: All expressions are postfixed with `Expr` while all statements end with `Stmt`.

[12]For example JSLint tries to discourage the use of so called anti-patterns (`http://www.jslint.com`)

Method / Property	Description
`.parse(code)`	Utilizes the `ES5Parser` to parse the given `code` and outputs an AST in JsonML format
`.translate(ast)`	Takes an AST structued as JsonML and translates it back to JavaScript
`.compile(code)`	Combines the functions `.parse()` and `.translate()` to perform a full compilation cycle on code. The result should be semantically equivalent.
`.parser`	Instance of the `ES5Parser` OMeta-module
`.translator`	Instance of the `ES5Translator` which inherits from `JsonMLWalker`
`.nodes`	Collection of all node-constructors which have been created with the help of the JsonML factory.

Table 6.5: *Interface of the es5-package*

Additionally, the interface provides three methods that can be used to control the full compilation life-cycle. The following example illustrates the usage of `parse` and `translate`:

```
var tree = es5.parse("var foo = 4"); //=> ["Program", {}, ["VarDeclStmt", ...]]
es5.translate(tree); //=> "var foo=4"
```

Testing

The developing process of the es5-package has been accompanied by extensive testing. To achieve a full coverage of the language specified by Ecma International [5] thousands of tests are necessary. Three approaches have been followed in order to avoid the time-consuming task of manually writing these tests but at the same time guarantee a high quality of code.

Firstly, an own test-suite has been implemented which covers most of the basic operations and all errors, that occurred during development. The set of this tests is small and cannot guarantee a correct compilation of all JavaScript programs.

While a full coverage of all language elements is appreciated, in practice only a subset is used by most of the programmers. In conclusion, successfully compiling the most used JavaScript libraries can paint a picture about how the compiler will work in a practical context. A set of six JavaScript has been parsed, translated and finally compared to the original source. To avoid differences in indentation, whitespaces and missing comments the source as well as the final result have been compiled with the closure-compiler[13] REST-API. To only normalize whitespaces the closure compiler has been configured to “Whitespaces only” and “Pretty print”.

[13] `http://closure-compiler.appspot.com/home`

Library	Version	Linecount	Success
Dojo	1.7.1	15.545	✓
ExtJs	4.0.7	22.381	✓
jQuery	1.7.1	9.266	✓
mootools	1.4.3	6.371	✓
prototype	1.7.0	6.082	✓
YUI	3.0.5	10.245	✓

Table 6.6: *Tested Libraries*

The results of the tests can be seen in table 6.6. All libraries have been successfully compiled by the es5-package and the results of the compilation did not differ from the source. Nevertheless, this round-trip testing does not allow conclusions about the correctness of the intermediate AST. Errors in both components parser and translator may be evened out in the result. At the same time errors in the closure-compiler implementation can have a large impact on the results of the tests further decreasing the validity of the tests. Despite the disadvantages, some conclusions about the correctness can be drawn from the fact that the parsing does not throw errors and therefore the input has been fully recognized.

Chapter	Without Compilation	Compiled	Total
07 Lexical Conventions	706	504	716
08 Types	123	103	124
09 Type Conversion and Testing	128	87	128
10 Executable Code and Execution Contexts	183	163	184
11 Expressions	1.306	909	1.310
12 Statements	519	431	525
13 Function Definitions	199	174	200
14 Program	24	20	24
15 Built-in ECMAScript Objects	7.288	6.363	7.970
Total	10.476	8.754	11.181

Table 6.7: *Results of applying the ECMAScript test suite*

To enhance the spectrum of tests a third approach utilizes the tests provided by the ECMAScript committee[14]. The tests are categorized into chapters akin to the outline

[14]A online version of the tests can be found at `http://test262.ecmascript.org/` - the source is allocated

of the specification. This comprehensive test suite can be used to test any JavaScript engine for correct implementation of the ECMAScript 262 standard. The test suite could be reused by performing a compilation step before running the tests. Hence, each JavaScript file containing tests is parsed and translated back to code before it is interpreted by the engine. Again, the validity of the tests highly depends on the JavaScript engine, the wrapper and the reliability of the testing suite. Errors in the construction of the syntax tree still cannot be found that way if the tree is correctly translated back to code. To compare the effects of the compilation-pass the tests have been performed twice. The first time without the additional compilation and the second time with the compilation. The results as seen in table 6.7 show that only about 84% of the tests that have passed the first stage also have been tested successfully after compilation. On the one hand this illustrates the need for improvement. On the other hand, considering the more practical approach of compiling real-world libraries the es5-package seems to be compatible for most applications.

6.2.4 The EJS Package

The fourth level, consisting of the *ejs-package,* is where the extension of JavaScript takes place. It only serves as a placeholder for any language-extension that builds on the es5-package. At the same time a skeleton of all necessary files is provided in order to allow an easy start. The extension than can be performed by filling the skeleton with implementation.

Method / Property	Description
`.parse(code)`	Parses the extended JavaScript code and creates an intermediate AST containing EJS-specific node-types.
`.translate(ejs-ast)`	Translates the given AST to be compatible to the translation of es5-package. Resolves all EJS-specific node-types and returns the resulting AST.
`.compile(code)`	Combines the methods `parse`, `translate` as well as `es5.translate` to compile extended JavaScript code to valid JavaScript.
`.parser`	Instance of the `EJSParser` which inherits from `ES5Parser`
`.translator`	Instance of the `EJSTranslator`
`.nodes`	Collection of all node-constructors required by EJS, also includes all es5-nodes

Table 6.8: *Interface of the ejs-package*

The final extension can be used in two ways. On the one hand the interface as seen in table 6.8 offers methods to access the basic functionality of the package. The most important one is `compile` which encapsulates the complete compilation process behind one simple method invocation. Incorporating the example extension from 6.1.5 a call to `compile` could look like

at `http://hg.ecmascript.org/tests/test262/`

```
ejs.compile("loop { do_it() }"); //=> "for(;;) { do_it() }"
```

which “desugars” the novel loop-statement to a regular for-statement.

On the other hand, the `ejs`-extension has been registered with Node.js. Every file with this extension is automatically compiled before it is evaluated as regular module. This allows to use modules written in EJS the same way as common JavaScript modules within Node.js. The following example illustrates the use of a module `server.ejs` implemented in the extended language

```
exports.start = function() {
  loop { handle_connections() }
}
```

from within another module `index.js`.

```
var server = require('server.ejs')
server.start();
```

In the remainder of this section we will take a look at how the scaffold which is included in the ejs-package looks like. Chapter 7 on page 75 provides further examples how actual extensions can be implemented.

The first component addresses the fact that the extended language has to be recognized by the parser. The grammar-module `EJSParser` inherits from the parser originated in the es5-package for this very purpose. The contents of the scaffolded grammar-module can be seen in figure 6.21.

```
var ES5Parser = require('es5').parser;
ometa EJSParser <: ES5Parser {}
module.exports = EJSParser;
```

Figure 6.21: *Scaffold of the `EJSParser` module*

First it is assured that all required dependencies are loaded before the actual implementation can take place. Finally the interface of the module is defined to be the parser itself. Being confronted with the task to create the abstract syntax tree the parser makes use either of the node constructors specified in the es5-package or the ones additionally added in the EJS nodes modules. Figure 6.22 shows the minimal contents of the nodes-modules.

```
var nodes = module.exports = Object.create(require('es5').nodes),
    Factory = require('jsonml').factory;
```

Figure 6.22: *Scaffold of the EJS-nodes module*

As already seen, an extension can be performed without the need to introduce new node-types to the AST by directly resolving the new syntactic elements during the process of parsing. In that case the use of the nodes-module as well as the translator is not obligatory. Nevertheless, when dealing with more complex constructs the implementation may be less of a burden when the resolution can be delayed into a separate translation pass. If not filled with implementation the translator seen in figure 6.23 performs a null-transformation, not modifying any of the nodes it visits.

```
var JsonMLWalker = require('jsonml').walker;
ometa EJSTranslator <: JsonMLWalker {}
EJSTranslator.force_rules = false;
module.exports = EJSTranslator;
```

Figure 6.23: *Scaffold of the EJSTranslator module*

The configuration flag `force_rules` is set to `false` in order to allow transparently walking all nodes for which no translation-rule is specified. Visiting such a node all children are recursively walked, but no transformation is performed on the node itself.

Helper Functions

Finally, the ejs-package provides a few auxiliary functions with the purpose to make the task of translation easier to achieve. The three most important helper functions are `join`, `expr` and `stmt`.

The first function `join` expects an arbitrary count of elements and performs a join operation on them. It is equivalent to `[el_1, el_2, ..., el_n].join(”)` but more convenient to read and write.

```
join("Hello ", "World") //=> "Hello World"
```

The functions `expr` and `stmt` are almost equivalent to each other. Again, both of them accept an arbitrary number of arguments. They concatenate the arguments just like `join`, but afterwards start a parsing process to create an abstract syntax tree from the given input. If a provided argument is a node-object, it is automatically translated to a string before `join` is applied. The difference between the two functions is the rule specifying the start of the parsing process. Hence, `expr` expects the input arguments to form an expression, while `stmt` expects it to be a statement. Likewise, the output of the two functions is either an AST representing an expression or a statement. The following example illustrates the use of `stmt` in the context of a parser implementation.

```
loop    = "loop" stmt:s           -> stmt("for(;;)", s)
```

The result of this rule is equivalent to the one presented in figure 6.15 but causes computational overhead, since the statement `s` is translated back and forth before it can be added as child-node to the newly created for-statement. Despite this performance issues, this auxiliary functions show their strength in an environment where complex AST structures need to be created from medium-sized templates.

6.3 The Grammars

Every package which is part of the framework includes at least one OMeta/JS grammar. Hence, the dependencies between the different packages naturally appear most plausible when inspecting the inheritance chain of the contained grammars.

A majority of the grammars inside the ometa-package (four grammar-modules containing a total of 11 grammars) is only needed at compile-time. This set of grammars forms the OMeta-compiler that is always used to compile OMeta/JS code to JavaScript. Most grammars of the compiler could be taken from the original OMeta/JS-project with only a few adoptions according to section 6.1.1.

Summarizing the insights from section 6.1.5 it becomes clear that the inheritance of the different grammars is highly influenced by the way of extension that is chosen:

$Code_{EJS} \rightarrow Code_{ES5}$ Requires the `ES5Parser` to generate Code instead of AST. No translation is needed - whether from es5-package nor from ejs-package.

$Code_{EJS} \rightarrow AST_{ES5} \rightarrow Code_{ES5}$ `EJSParser` inherits from `ES5Parser` and directly "desugars" new syntax to valid ES5 subtrees. The existing ES5 backend can be reused. No `EJSTranslator` is needed.

$Code_{EJS} \rightarrow AST_{EJS} \rightarrow Code_{ES5}$ `EJSParser` inherits from `ES5Parser` but emits EJS-nodes. New translator is required to resolve all non-ES5 nodes and output valid JavaScript code. `EJSTranslator` has to inherit from `ES5Translator`. Only one pass of translation is performed. ES5 backend cannot be reused.

$Code_{EJS} \rightarrow AST_{EJS} \rightarrow AST_{ES5} \rightarrow Code_{ES5}$ Parser is the same as the previous one. Also requires new translator, but this time the translator outputs valid es5-tree, like the parser from approach two does. Two steps of translation. Every translator inherits from `JsonMLWalker`.

In any case `EJSParser` has to inherit from the original `ES5Parser` in order to extend the recognition capabilities and at the same time maintain a full backwards compatibility with JavaScript. The first case differs from the rest in a way that it requires the `ES5Parser` to be implemented different. It needs to emit strings of JavaScript code instead of an AST. In consequence, this strategy is not supported by framework at hand.

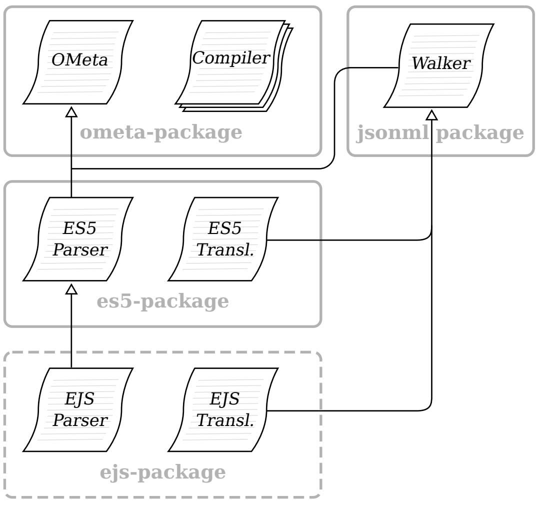


Figure 6.24: *Packages as seen from the grammar point-of-view*

Approach two and four both allow to reuse the existing backend facilities that are independent from EJS, like tree-to-source translation, code analysis, visualization and optimization. Due to this reason these two strategies are natively supported by architecture as seen in figure 6.24). Nevertheless, with some minor modifications (changing the inheritance of `EJSTranslator` from `JsonMLWalker` to `ES5Translator`) it is possible to implement an extension with the tools provided by following approach three.

6.4 Summary

In this chapter a five step process has been presented which allows to easily add syntax extension to JavaScript. After preparing the environment and adapting OMeta/JS a complete compilation life-cycle has been introduced. JavaScript is firstly parsed, a JsonML AST is created to be finally translated back to JavaScript. The subsequent step builds upon this life-cycle in order to create a language extension. The different ways of using parsers and translators have been presented from which two approaches have been chosen. Following the first chosen approach, the parser directly emits an ES5-compatible AST, which in turn can be translated to JavaScript code by the `ES5Translator`. The second alternative introduces a new tree format AST_{EJS} which is created by the `EJSParser`. In an additional step of translation it is than translated to AST_{ES5} . Finally a new loop-statement has been introduced to demonstrate how an extension can take place.The five step process resulted in an architecture which has been presented in the second section of this chapter. All involved packages have been analyzed. The ometa-package can be used standalone to allow the inclusion of OMeta grammars in any project. In addition the jsonml-package may be used if node-constructors and a prepared walker-grammar are required. The es5-packages provides a full implementation of a JavaScript parser and translator. The resulting syntax tree may be used to be analyzed, optimized or manipulated in any other way. Finally, the ejs-package uses all of the previously introduced packages in order to provide a scaffold for an easy extension of JavaScript. The third section illustrated the dependencies of all grammars that are included in the packages to support a better understanding of the architecture at hand.

Chapter 7

Usecase: Example Extension

Contents

In the previous chapter we have seen how an extension can be created in general. Except for a newly introduced loop-statement our fictional language EJS has not much to offer so far. In this chapter we will discuss some more example extensions in order to illustrate how easy it is to extend JavaScript using the framework at hand. In general the possible extensions can be grouped into two categories:

1. Extensions that can be translated directly to JavaScript at *compile time*. One example for this category is the `loop`-statement presented in section 6.1.5, which can be mapped to `for(;;)`.
2. Extensions that require an additional *runtime* library to work. In section 7.4 a syntax for classical object orientation is introduced which depends on an implementation of the class-system at runtime.

It should be obvious that the extensions presented in this chapter neither form a complete new language, nor do they claim to “fix” JavaScript as a language. Their purpose is to *a*) enrich the individual programming experience by providing notational convenience and *b*) to illustrate the extension mechanisms provided by the framework.

For each example a common pattern or problem specific to JavaScript is identified before a solution in JavaScript itself is presented. In some cases a small introduction into some JavaScript internals is provided to gain a better understanding for the problem at hand (For a detailed explanation of the ECMAScript internals I recommend Dmitry Soshnikov's excellent writings [24, 25]). Afterwards this solution is mapped to a new syntax, mostly to avoid the verbose constructs that arose from the solution. The examples are presented in order of ascending complexity.

All extensions are implemented following the three step approach $Code_{EJS} \rightarrow AST_{EJS} \rightarrow AST_{ES5} \rightarrow Code_{ES5}$ and therefore require custom node-types. In the implementation, often the variable `nodes` is used to store all constructors for node-types. As it can be seen in section 6.2.4 the collection of EJS-nodes inherits from the nodes contained in the es5-package.

7.1 Scope Forcing with `!{}`

JavaScript uses *static lexical scoping* together with the concept of environments to implement variable resolution. Variables are declared using the `var` keyword. The term `var x = 42` introduces a new variable `x` to the current *environment* and sets it's value to `42`, we call this procedure a *binding*. In contrast to other programming languages such as C or Java that use block scoping, JavaScript only creates environments when lexically entering a new function[1]. Therefore it is said to be "function scoping". Environments are nested and the search always starts with the currently active environment. Every environment holds a link to the surrounding environment, called outer-environment. All entries that can be found directly in an environment are stored in the environment-record as key-value pairs. Due to the chaining characteristic of nested environments it is sometimes referred to as *scope chain*.

This commonly leads to the pattern of immediately calling anonymous function expressions in order to create a new scope and prevent pollution of the outer scope, as seen in figure 7.5.

```
var foo = 5;
if(true) { var foo = 6; }
console.log(foo); //-> "6"

// this creates a new scope
(function() {
  var foo = 7;
  console.log(foo); //-> "7"
})();
console.log(foo); //-> "6"
```

Figure 7.1: *Forcing scope in JavaScript by immediately calling an anonymous function*

That way the creation of an environment record is forced and the declaration of variable `foo` can shadow the equal named variable defined in the outer environment without overriding it. Additionally, the shorter version seen in figure 7.2 looks like a reminiscence of how `let` can be implemented in Lisp using lambdas.

[1] There are some corner cases like the `catch`-statement. Also code evaluated in the global environment or within `with`-statements leads to the creation of so-called "object environments".

```
(function(foo) {
  console.log(foo); //-> "7"
})(7);
console.log(foo); //-> "6"
```

Figure 7.2: *Using arguments to introduce new variables*

Closures

Generally speaking, a closure is a function which preserves the lexical context it is defined within. This condition is automatically met in JavaScript since each function stores it's context in the internal property `[[Scope]]`. The purpose of a closure is to assure access to all non-function-local variables referenced inside the function, even if the outer context already terminated.

```
function Outer(conserved) {
  return function() {
    console.log(conserved);
  }
}
var closure1 = Outer(1),
    closure2 = Outer(2);
closure1(); //=> "1"
closure2(); //=> "2"
```

Figure 7.3: *Closures preserve the lexical context of definition*

Figure 7.3 shows the function `Outer` which awaits one argument `conserved` and returns a dynamically created anonymous function. This anonymous function itself is a closure because it references the variable `conserved` defined in the scope of `Outer`. Every time we call the function `Outer` the execution enters the very same function code, creates a new environment and binds `conserved` to the argument passed during the call. This environment is preserved as property `[[Scope]]` of the anonymous function which after all is returned by `Outer`.

Furthermore, when calling the returned function stored in `closure1` as well as in `closure2` we suddenly gain indirect access to variables inside of the environment which only lives on through it's conservation within a closure.

Revealing Module Pattern

The example in figure 7.3 illustrates how the variable `conserved` can be created and afterwards be accessed only by calling the returned function. In this setup there is no other way to either change or read the variable's contents. This very important fact initially led to the *module pattern*, refined by Christian Heilmann under the name *Revealing Module Pattern*. The pattern is used everyday by many JavaScript developers, as it is an effective way to create a category of variables which can be considered to be private or hidden.

Figure 7.4 on the next page shows a typical example-usage of the revealing module pattern. By calling `Person` with `“Peter”` as argument a new environment with three

```
var Person = function(name) {
  // this is private
  var age = 0;

  // this will be public
  function say_hello() {
    console.log("Hello, my name is " + name + " and i’m " + age + ".");
  }

  // specification of the interface
  return {
    greet: say_hello,
    age: function(new_age) { age = new_age; return this; }
  }
}
var peter = Person("Peter").age(32);
peter.greet(); //=> "Hello, my name is Peter and i’m 32";
```

Figure 7.4: *Usage of revealing module pattern*

```
// No new scope is created
{ var foo = 4; }

// foo is local to the new scope
!{ var foo = 4; }

// output of compilation
(function(){ var foo = 4; })()
```

Figure 7.5: *Forcing the creation of a new scope*

entries is created. Firstly the formal parameter `name`, secondly the variable binding `age` and finally the function declaration `say_hello`. Initially we can think of all three variables as private. However, by returning a new object with the two properties `greet` and `age`, some private information is *revealed*. This object can in some way be considered as “public interface” to the module in order to access some of the hidden information. The implementation of `age` shows that we may even change the contents of the variables conserved within the closure. This once again is remindful of how object orientation can be implemented from scratch in Lisp or other functional programming languages supporting closures and first-class functions.

7.1.1 Introducing a new Syntax

Due to the widespread use of the pattern`(function() {})()` a shorter syntactical description may be introduced which expresses the true character of this phrase. Since the goal of applying this pattern is to *force a new scope* we may call the novel expression “scope expression”. The key idea is to force the creation of a new scope on a block by prepending an exclamation mark. This gets visible in figure 7.5. The first block-statement does not create a new scope. As a result, `foo` is declared in the outer environment. The second block is prefixed with an exclamation mark to use the novel syntax. The result of compiling this expression can be found at the bottom-line.

The exclamation mark has been chosen, since it perfectly represents the “forcing”-character of the new operator. Due to it’s close visual relationship to block-statements it appears naturally to also use it in combination with for-statements as in figure 7.6.

```
for(var i = 0; i < arr.length; i++) !{
  var el = arr[i];
  ...
}
```

Figure 7.6: *Example usage of the scope-forcing operator*

In contrast to block-statements, scope expressions may return results. This characteristic also makes them suitable for revealing module declarations. An example for this usage can be seen in figure 7.7.

```
APP.module = !{
  ...
  return {
    // interface of the module
  }
}
```

Figure 7.7: *Using the scope-forcing operator to implement the revealing module pattern*

7.1.2 Implementing the Syntax

The implementation of the scope expression can be accomplished straight forward. The first thing to do is to extend `EJSParser` in order to be able to recognize the new syntax. In this special case this can be achieved without complications. The usage of the exclamation mark only collides at one possible position with the common ECMAScript grammar - when it is followed by an expression to form a unary-expression. Luckily, a block-statement is not a valid JavaScript expression[2] which leaves us with the grammar as seen in figure 7.8.

```
scopeExpr = "!" block:b   -> nodes.ScopeExpr(b),

// Register scopeExpr as new expression
expr      = scopeExpr | ^expr
```

Figure 7.8: *Extending the parser to allow scope forcing*

The second line of code is needed to tell the parser to expect the new scope expression at any place a common expression may occur. In this implementation the major pattern for extension gets visible. At first a small grammar module is implemented to recognize the novel syntax. Afterwards this module is “registered” at the places it is expected to occur, here as expression.

The node which is instantiated by the parser and added to the AST is of the type `ScopeExpr`. Since this is not a valid ES5 node it needs to be created first. This can be seen in figure 7.9.

[2]There is one small corner case with an empty block statement, since it cannot be distinguished from an object-literal. Since forcing a new scope and than leaving the block empty is pointless, this corner case renders itself insignificant.

```
Factory('ScopeExpr');
```

Figure 7.9: *Adding a new node-type to represent scope-expressions*

The factory function is called with the node-type as the only argument, since no attribute-object or special callback is required. Now that the AST_{EJS} contains nodes that are alien to the translator of the es5-package we need to replace it with a subtree representing the immediate call of a anonymous function. Therefore, the `EJSTranslator` is extended with the code from figure 7.10.

```
ScopeExpr :n = walk:block   -> stmt('(function()', block, ')()')
```

Figure 7.10: *Extending the translator to convert scope expressions into wrapping function calls*

The only child, a block statement, is recursively translated in order to be reused in the semantic action on the right hand side. The helper function `stmt` is called with a mixture of strings and the AST node representing the block statement. It constructs a new AST node representing the function call with the block filling the function's implementation. The subtree returned by the helper is a valid AST_{ES5} and can be translated by the `ES5Translator` to the appropriate JavaScript source code.

7.2 The Substitution Operator “#{}“

The second extension deals with the simple fact that JavaScript does not support string-substitution. In JavaScript strings and other primitive values are commonly concatenated using the `+` operator. Assembling a simple greeting message like “`Hey Jim, how are you?`”, with the name being a variable, requires a destruction of the string into three parts:

```
var greeting = "Hey " + name + ", how are you?"
```

This manual concatenation is not only inconvenient to read and write, but also error prone. The optional in-between whitespaces and plus-operators make it difficult to assure that whitespaces and punctuation are placed correctly. Additionally, inserting linebreaks before a plus-operator terminates the expression, since a semicolon is inserted automatically. This can be avoided by using the alternative expression:

```
var greeting = ["Hey ", name, ", how are you?"].join('')
```

Compared to the original version, the convenience to read and write this expression is not much increased. Nevertheless, it comes with three advantages:

1. Linebreaks may be inserted at any arbitrary position without breaking the expression, since it is terminated with the closing bracket
2. Calculations like `age + 10` may be inserted without parenthesis. The correct precedence of the operators is assured by using commas as separator.

3. A higher performance can be expected according to [31]. The `join` operation has to allocate memory only once in order to concatenate the different parts instead of reallocating it for each single concatenation[3].

7.2.1 Introducing a new Syntax

Other languages like PHP and Ruby offer an elegant solution to the problem of inserting variables and expressions into strings. It is called string substitution. Even the C standard library handles this usecase by supporting a similar functionality with `sprintf`. All three solutions can be seen in figure 7.11.

```
// PHP
$greeting = "Hey $name, how are you?";

// Ruby
greeting = "Hey #{name}, how are you?";

// C
char greeting[256];
sprintf("Hey %s, how are you", greeting, name);
```

Figure 7.11: *Three different implementations of string-substitution*

In the following we will see how to incorporate the Ruby syntax for string substitutions into EJS. Obviously, the PHP syntax could be implemented like-wise. The example in figure 7.12 shows how the resolution of the syntax to JavaScript can take place. It gets visible that the string has to be split at any occurrence of a substitution in order to be converted into the array-notation.

```
// input: EJS
var greeting = "Hey #{name}, how are you?";

// output: JavaScript
var greeting = ["Hey ", name, ", how are you?"].join(’’)
```

Figure 7.12: *Resolving the string-substitution of EJS to JavaScript*

7.2.2 Implementing the Syntax

According to the implementation pattern we have seen in the previous example, the first thing to do is to extend the parser in order to recognize the new syntax. This can be seen in figure 7.13 on the following page. For reasons of simplicity only double-quoted strings are enabled to support the extended syntax while single quoted strings may remain unchanged.

Right after the opening quote an arbitrary number of strings and string-substitutions is expected. The string-substitution recognizes the opening sequence `#{`, then matches an expression before it consumes the closing curly brace `}`. The rule `strPart` in turn simply

[3]This phenomena could not be reproduced by a micro-benchmark using Node.js. Without having performed further investigation, it is most likely that the JIT-compiler of the underlying V8 engine does a lot of optimization in this very case.

```
stringExpr = spaces ’"’ (strSubst | strPart)+:cs ’"’   -> nodes.StringExpr(cs),
strSubst   = ‘‘#{’’ spaces expr:e spaces ’}’          -> e,
strPart    = <( ~(’"’| ‘‘#{’’) char)+>:cs             -> nodes.String(cs),

// add string expression as new primary expression
primExpr   = stringExpr | ^primExpr
```

Figure 7.13: *Extending the parser to allow string-expressions*

consumes all characters that neither indicate the end of the string nor the begin of a string-substitution. The node-type `StringExpr` finally is used to create the appropriate AST node. The creation of the required node-constructor using the factory can be seen in figure 7.14.

```
Factory(’StringExpr’, {}, function(parts) {
  this.appendAll(parts);
})
```

Figure 7.14: *Adding a new node-type to represent string-expressions*

The provided callback-function is used to append all recognized parts as individual child-nodes. Otherwise, the array `parts` containing all strings and expressions that are part of the expression would end up as the only child of the node.

Equivalent to the previous extension, the last step of implementation is to adapt the translator in order to convert the new node type to a valid AST_{ES5}. This can be seen in figure 7.15.

```
StringExpr :n = walk*:els    -> expr(nodes.ArrayExpr(els), ’.join("")’)
```

Figure 7.15: *Extending the translator to "desugar" string expressions to arrays*

To translate the string expression node `n` into a subtree, which in turn can be compiled to the expected JavaScript code, several tasks have to be performed. At first, all children are translated recursively and stored into the variable `els`. Then a new array expressions is created using the node-constructor `ArrayExpr` with the child-nodes as parameter. Finally, the magical helper function `expr` is used to avoid creating the member-expression (`join`), the call-expression and the argument (`""`) by hand.

Granted, the extension of the parser might be a little more sophisticated than in the first example, but in total only ten lines of code are necessary to add a second valuable feature to JavaScript.

7.3 Lambda Expressions {||}

One of the most important features of JavaScript is that functions are first-class objects. Hence, there are not only function declarations (which are in fact statements) but also function expressions, also called "inline functions" or "anonymous functions". This allows functions to be used the same way as common objects. For example they can be passed as arguments to higher order functions which in turn may return yet another dynamically constructed function.

We have already seen that in JavaScript functions are not only used in the traditional way, in order to allow code reuse, but also to control the scope of variable declarations and ensure encapsulation. They are also part of every constructor definition, which we will see in the following section 7.4.

Since functions are so important the keyword is dispersed everywhere. While it may be reasonable for function declarations or constructor functions in order to create a visual footprint in the source code, it is too verbose for function expressions, especially when used as callback functions. The following example makes use of jQuery to illustrate the usage of passing functions as arguments:

```
$('#navigation a').filter(function() {
  return $(this).html() === $(this).attr('title')
}).map(function(i, el) {
  return $(el).parent();
}).click(function(evt) {
  console.log("The parent of a link has been clicked");
  return false;
});
```

Despite the sparse sense of this example, it gets clearly visible that a great part of the code consists of redundant character sequences. As already stated, calling a function by passing another function appears quite often in JavaScript. This always results in the pattern:

```
called(function(a,b,c){ return a+b+c; })
```

Brendan Eich, the creator of JavaScript, apologized for picking such a long keyword[4] and made clear that it is originated in AWK[5].

The remainder of this section deals with removing much of this visual noise which is not possible in JavaScript itself.

7.3.1 Introducing a new Syntax

In order to pass behavior to a function-call the programming language Smalltalk provides a mechanism called “code blocks”. The blocks can be compared to functions in JavaScript or lambdas in Lisp, since they all are first-class objects and save their lexical scope as a closure. Ruby adapted the concept of code blocks and introduced the following syntax[6]:

```
called {|a,b,c| a+b+c }
```

It is worth noting that this syntax is in discussion to be added to the latest release of the ECMAScript Standard [7] where the syntax is representing “block-lambdas”. In consequence, maybe in the near future, the extension developed in this section is not

[4] http://brendaneich.com/2011/01/harmony-of-my-dreams/
[5] http://brendaneich.com/2010/07/a-brief-history-of-javascript/
[6] In fact Ruby provides several ways to create blocks. This one is chosen here, since it is the one most compatible to JavaScript.

needed anymore since it will be natively implemented in JavaScript. Nevertheless, the semantics of block-lambdas are slightly different than the results achieved by directly translating `{||}` into `function(){}`. To list just some of the differences:

1. Block-lambdas do not create a new `arguments`-object
2. Block-lambdas do not change the binding of `this`
3. Block-lambdas behave the same way as common block-statements (i.e. `{}`) when calling `break`, `continue` or `return`

Despite the fact, that the implementation presented here cannot hold up to any of these requirements we will denote it as *lambda expressions* in order to differentiate between normal function-expressions and the extension discussed in the remainder of this section.

The extension itself is three-pronged. The first and most obvious prong is the replacement of the function keyword by using the unique pattern `{||}` as it is known from Ruby. The second prong is the introduction of a new syntax for calling a function. If the only argument is a lambda-expression the surrounding braces may be omitted since the lambda is already bracketed by curly braces as delimiters. Finally, the last prong is the *implicit return*, which reduces the need of explicitly calling `return`. If the last statement inside of the function's body is an expression it will be returned automatically. Applying this novel syntax to the example above results in a much cleaner version:

```
$('#navigation a').filter {|i| $(this).html() === $(this).attr('title') }
                  .map {|i, el| $(el).parent() }
                  .click {|evt|
                     console.log("The parent of a link has been clicked");
                     false
                  }
```

About 24% of the used characters[7] could be removed leaving us with code that is much easier to read without being distracted by redundant braces and keywords.

It is still possible to add other parameters than a function callback by simply passing them the usual way:

```
called(x,y) {|a,b,c| a+b+c }
```

Actually, this expression is semantically equivalent to placing the callback within the calling braces:

```
called(x,y, {|a,b,c| a+b+c })
```

Yet another example illustrates the combined use of calling a function, both with and without additional parameters:

```
$.getJSON('/get_posts.php') {|data| console.log(data); }
.error {|evt| console.error("An error ocurred") };

// which is compiled to
$.getJSON('/get_posts.php', function(data) { console.log(data)})
.error(function(evt) { console.error("An error ocurred") })
```

[7]Original character-count: 207, optimized version: 157. Whitespaces are not included.

Again, the jQuery API is used for this example. This time to send an asynchronous request to the server and register two callbacks. One function in case of success and one to handle errors.

7.3.2 Implementing the Syntax

The three pronged nature also applies to the implementation of the new syntax. As usual, we first point our attention to the implementation of the parser as it can be seen in figure 7.16.

```
// 1. Allow lambda-expression as alternative function expression
lambdaExpr = "{" ("|" spaces listOf(#formal, ',')):a "|"  -> a
                 | "||"                                  -> []
                 ):args spaces srcElem*:ss
             "}"  -> nodes.LambdaExpr(nodes.FunctionArgs(args),
                                      nodes.BlockStmt(ss)),

funcExpr   = lambdaExpr | ^funcExpr,

// 2. Introduce a new syntax for calling functions
lambdaCallExpr :p = ("(" listOf(#assignExpr, ','):a ")" -> a
                    | empty                            -> []
                    ):args lambdaExpr:f -> nodes.CallExpr(p, args.concat(f)),

accessExpr        = accessExpr:p lambdaCallExpr(p)
                  | ^accessExpr
```

Figure 7.16: *Extending the parser to allow the new function expressions*

Firstly, the matching function for the lambda-expression is implemented. There are two cases to handle, whether there are given arguments or not. Afterwards, an arbitrary amount of source elements is matched in order to form the function's body. The arguments are wrapped inside of a `FunctionArgs` node, whereas the body is represented by a block-statement containing all matched statements.

Secondly, the new syntax for calling functions with a trailing block-lambda is introduced. Again, it has to be handled if additional arguments are supplied or not. The new call-expression takes one parameter `p`, which is the left associative access-expression the call will be send to. This strategy has been discussed in section 6.1.2. In this section we learned how to make use of `accessExpr` and parametrized rules in order to allow different types of access-expressions (for instance member-expressions or call-expressions) to be chained in arbitrary order.

The rule `lambdaExpr` creates an instance of the almost equally named node-type whose simple implementation can be seen in figure 7.17.

```
Factory('LambdaExpr');
```

Figure 7.17: *Adding a new node-type to represent lambda-expressions*

Once again, the last step is to extend the `EJSTranslator` in order to translate the `LambdaExpr` nodes to valid es5-nodes (see figure 7.18).

```
LambdaExpr :n = walk:args implicit_ret:body -> nodes.Function(args, body).expr(true),
implicit_ret  = walk:body !this.helpers.implicit_return(body)
```

Figure 7.18: *Extending the translator to convert lambda-expressions into functions*

Both children, arguments and body, are recursively translated before a common function-node is created. In addition, the function-node is flagged as expression. A more interesting part is the implementation of "implicit returns". Instead of traversing the function body by applying the rule `walk` a new rule `implicit_return` is introduced. The new rule in turn applies `walk` before a helper function is used to manipulate the results. The implementation of the helper can be found in appendix A.4. In brief, the function analyses the child-nodes included in `body`. If the last child-node is an expression, which may be returned a new `ReturnStmt` is created around the child-node and the modified function body is returned.

An alternative to implement this behavior is to write a dedicated OMeta/JS translator and call it as a foreign rule (Also see section 5.1.9). Using this recursive approach it is more easy to also detect "indirect" expressions which may occur if the last element is an if-statement.

7.4 Classes and Object Orientation

The last extension presented here deviates from the previous three. Not only the implementation of the translator is a little more challenging but the extension, as we will see, also requires a *runtime* in order to be used. It also appears to be the most interesting one, since we take the chance to not only introduce some "syntactic sugar", but also try to close the gap between classical and prototypal inheritance (at least when it comes to class-definitions).

There is a vast amount of libraries simulating a class-based inheritance as known from languages like Java, C++ or Ruby to make inheritance and object orientation (abbr. OO) in JavaScript less of a burden. Some examples are Joose[8], Klass[9], JsClass[10], myClass[11], species[12] and proto-js[13], just to name a few standalone libraries. Larger base-libraries like Ext.JS[14], Dojo[15] and MooTools[16] also implement their own class-based inheritance amongst many other features. For those readers yet unfamiliar with prototypal inheritance the following section provides a small overview into this very topic.

[8] `http://joose.it/`
[9] `http://dustindiaz.com/klass`
[10] `http://jsclass.jcoglan.com/`
[11] `http://myjs.fr/my-class/`
[12] `https://github.com/k33g/species`
[13] `https://github.com/rauschma/proto-js`
[14] `http://sencha.com/`
[15] `http://dojotoolkit.org`
[16] `http://mootools.net`

Prototypal Inheritance

In order to understand how inheritance is achieved in JavaScript this section introduces the core-concepts of prototype-based inheritance and shows the differences to the class-based alternative.

In class-based object-orientation many objects are grouped within one class, which describes the layout and shared behavior of all of it's instances. Behavior further can be specialized by creating a subclass of the superclass, or in other words *extending* the superclass. Instances of the subclass embody the behavior specified in the subclass as well as the one specified in the superclass.

This relationship between an object and it's class often is referred to as *instance-of*-relationship [26]. Class based object-orientation can be compared to building a car from a blueprint. It hence can be split into two separated temporal phases. At modeling-time the engineer layouts how the car should look like and behave once it is built. At runtime a factory can create instances of the abstract blueprint and the finished shiny cars are ready to be used.

The concept of prototype-based inheritance is quite different, but it is said to be more easy to understand, because there are less basic-concepts one have to understand to use the language [26]. Instead of instantiating objects from an abstract plan objects are created directly by *cloning* an existing object. Additional to cloning the language must provide a functionality to *create a parent-link* from one object to it's prototype. The object automatically inherits all state and behavior of it's prototype once such a link is realized. Of cause further behavior can be added to the object. Only if a desired functionality cannot be found in the object itself, it's prototype is consulted. If the prototype does not know the answer either - again it's prototype is asked. The search continues in the so called *prototypal-chain*. It is important to note that the cloning-process preserves the link to the prototype of the cloning-source. If we clone an object multiple times, all newly created objects, as well as the original share a single prototype. They seem to form a class of objects and share behavior. Hence changes to the prototype automatically influence all derived *child-objects*.

The analogy to the subclassing-process as seen in class-based inheritance now can be broken down into a two-step process. At first create a new object together with it's prototype-link and afterwards fill it with the new desired behavior and state.

Implementation in JavaScript

Every object in JavaScript has a number of different *internal properties*. Among these, there is `[[Prototype]]` which describes the link to the objects prototype. As we are dealing with an internal property there is no direct way to access or manipulate this property like we would expect. So how can we set the prototype of one object to point to another object?

In figure 7.19 on the next page the difficulty of this operation gets visible. Let's disassemble the internal process of calling `new Person()`:

1. A new object, let's say `obj`, is created with it's internal property `[[Prototype]]` set to `person`.

2. The constructor function is called in context of `obj` Using the arguments provided in the call to `new`.
3. Finally `obj` is returned.

```
var Person = function(){};
Person.prototype = person; // instance containing shared behavior
var peter = new Person();
```

Figure 7.19: *A prototype-link in JavaScript can be created by an intermediate constructor-function*

This clearly is one of the most irritating miss-conception in JavaScript's language design. According to Douglas Crockford [4] the purpose of this constructor-pattern was "to make the language more familiar to classically trained programmers". He also proposed an alternative implementation, which avoids the unnecessary constructor-noise and reveals "JavaScript's true prototypal nature". This pattern has found it's way in the 5th edition of the ECMAScript standard [5] as the built-in function `Object.create(obj, prop)`[17]. It equips us with a single-step-tool to easily create a new object and link it's prototype to a specified object.

```
var peter = Object.create(person);
Object.getPrototypeOf(peter) === person; //=> true
```

Figure 7.20: *`Object.create` is a shorter solution to create a prototypal-link between two objects*

With this functionality we might re-implement the new-operator as seen in listing 7.21. This simplified implementation illustrates the basic three internal steps described above.

```
Function.prototype.new = function() {
  var obj = Object.create(this.prototype);
  this.apply(obj, arguments);
  return obj;
}
// now ‘new‘ could be used like
var peter = Person.new();
```

Figure 7.21: *Simplified implementation of the new-operator using `Object.create`*

Working with JavaScript's prototype-system it is important not to confuse the internal `[[Prototype]]` property and the `prototype` property of a constructor-function. The first is used to resolve member-access in the prototypal chain. The second one in contrast simply stores a reference to the object, which will be the prototype of all objects created with this constructor.

7.4.1 Introducing a new Syntax

As already illustrated in the introduction of this section there are quite a few different implementations for object orientation in JavaScript. In the following we will take a

[17] `Object.create` expects two arguments. The first one will be used as prototype when creating the new object. The second optional argument describes the object's properties[15]

brief look at two examples: Ext.JS and MooTools[18]. Yet, both are fully compatible to JavaScript and do not require extra syntax. They mastered the challenge of creating a domain specific language within JavaScript.

Since we are trying to develop an example extension, the following examples only cover the basic features of object orientation, such as class-definition and inheritance. Of course both implementations offer a lot more functionality.

```
// Ext.JS
Ext.define("Person", {
  constructor: function(name, age) {
    this.name = name; this.age = age;
  },
  birthday: function() { return this.age++ },
  greet: function() {
    return "Hey, my name is " + this.name +
           " and I’m " + this.age " years old";
  }
});

// MooTools
var Person = new Class({
  initialize: function(name, age) {
    this.name = name; this.age = age;
  },
  birthday: function() { return this.age++ },
  greet: function() {
    return "Hey, my name is " + this.name +
           " and I’m " + this.age " years old";
  }
});
```

Figure 7.22: *The definition of classes in Ext.JS and MooTools*

Figure 7.22 illustrates how “classes” are defined in each implementation. It gets visible that there are some differences between the two implementations. While classes are defined in Ext.JS calling the method `Ext.define` and supplying the new class-name as first argument, MooTools creates classes by instantiating a new `Class`. Additionally, the naming of the constructor-functions differs.

In figure 7.23 on the next page we can inspect how the classes of the previous example can be extended to add specialized behavior. A new method `work` is implemented to allow a person to do it’s every day job. Sadly, this does not change anything about the fact, that the age of each instance is constantly incremented every year.

Again, differences can be discovered in the way the parent class is specified. The first difference is the naming of the property (`extend` vs. `Extends`). A more significant difference is the fact that Ext.JS requires the name of the parent class as string, whereas MooTools expects the parent-object itself.

In the following step we will take a look at a unified and library neutral syntax. The goal of this novel syntax (as it can be seen in figure 7.24 on the following page) is to *a*) close most of the gap between the two different implementations and *b*) to be able to easily switch the underlying base-library.

In this example new syntax elements, such as the `class` keyword, are introduced to create a intermediate level of abstraction. The constructor is specified by an anonymous

[18]Detailed information about the individual implementation can be found at `http://docs.sencha.com/ext-js/4-0/#!/api/Ext.Class` for Ext.JS and `http://mootools.net/docs/core/Class/Class` for MooTools.

```
// Ext.JS
Ext.define("Employee", {
  extend: "Person",
  constructor: function(name, age, job){
    this.callParent(arguments);
    this.job = job;
  },
  work: function() {
    return "Doing my job as " + this.job;
  }
});

// MooTools
var Employee = new Class({
  Extends: Person,
  initialize: function(name, age, job) {
    this.parent(name, age);
    this.job = job;
  },
  work: function() {
    return "Doing my job as " + this.job;
  }
})
```

Figure 7.23: *Class inheritance in Ext.JS and MooTools*

```
class Person {
  function(name, age) {
    this.name = name; this.age = age;
  }
  birthday: {|| this.age++ }
  greet:    {|| "Hey my name is #{this.name} and I’m #{this.age} years old." }
}

class Employee < Person {
  function(name, age, job) {
    super(name, age);
    this.job = job;
  }
  work: {|| return "Doing my job as #{this.job}" }
}
```

Figure 7.24: *Unified syntax for class definition*

function instead of explicitly naming it. Inheritance can be achieved by using the `Child < Parent` operator in the header of the class definition. Again, this solves the problem of finding the correct property name to save the parent-class in. Properties describing the instances of the class can be added as key-value pairs. Separating commas are optional.

Even if in this example only behavior is added to describe the instances, also atomic values or objects could be used. Depending on the actual OO-implementation the properties may either be added to the prototype of the instance or to the instance itself. The example also utilizes syntax extensions as they are presented earlier in this chapter, such as lambda-expressions and string substitution.

7.4.2 Implementing the Syntax

We start off by making changes to `EJSParser` just like we did with all previous extensions. The grammar needs to recognize the class-definition statements. The required changes can be seen in figure 7.25.

```
klass      = "class" spaces <name ('.' name)*>:n ("<" spaces <name ('.' name)*>:e)?
               "{"
                  funcExpr?:c
                  klassProps?:p
               "}"
               -> nodes.ClassStmt(n, e, c, p),

klassProps = objBinding:f ((sc | ",") objBinding)*:r -> nodes.ObjectExpr([f].concat(r)),
stmt       = klass | ^stmt
```

Figure 7.25: *Extending the parser to allow class-statements*

The format of the declaration header is stated in the first line of code. The name of the class is followed by an optional name specifying the parent-class. The body of the class consists of one optional function-expression, representing the constructor, followed by an also optional definition of class-properties. Each individual property is recognized by the rule `objBinding`, which is part of the `ES5Parser` grammar and matches bindings such as `foo: 42`. The properties can be separated using commas, linebreaks or semicolons. The node representing the statement is constructed by passing four parameters being the name, parent-class, constructor and the instance-description. Figure 7.26 illustrates the implementation of the node-constructor.

```
Factory('ClassStmt', {
  name: undefined,
  parent: undefined
}, function(name, parent, constr, spec) {
  this.name(name)
      .parent(parent)
      .append(constr, spec);
});
```

Figure 7.26: *Creation of the node-constructor for class-statements*

In contrast to the node-constructors we have seen so far, the `ClassStmt` makes use of the attributes-object to store the class name and the name of the parent class. The callback function is used to set the values for `name` and `parent`. Afterwards the nodes representing the implementation of the class body are appended as children.

The most interesting part of the class-statement is the implementation of the translator. When designing the translator we have to choose between two approaches *a*) map the class definitions directly to a special OO-implementation like Ext.JS or *b*) create a neutral class-description object.

The latter option can, at runtime, be mapped to a specific implementation and hence appears to be more flexible. Using this neutral approach, the definition of the example class `Employee` might look as seen in figure 7.27 on the next page.

It gets visible that an actual implementation of `Object.define_class` has several options in order to assemble the class. The first argument simply represents the name of the class which shall be created. The second argument is used to further specify the internals, much like a blueprint does. Let's step through this specification object. The first two properties `parent_name` and `get_parent` both allow to access the parent of the class.

```
Object.define_class("Employee", {
  parent_name: "Person",
  get_parent: function() { return Person },
  set_class: function(__class__) { Employee = __class__ },
  constructor: function(name, age, job) { ... },
  spec: {
    work: function() { ... }
  }
});
```

Figure 7.27: *Compilation result for class `Employee`*

The latter of the two is a function which stores a reference to the parent object within a closure. This defers the resolution of the reference to the invocation of `get_parent`. The third property `set_class` makes use of a similar mechanism. It allows to bind the created class to a variable in the lexical scope of definition. The constructor function is, independently from the base library, always stored in the property named `constructor`. Finally, all other properties describing the instance are stored within the object `spec`.

Before discussing the actual implementation of the translator it is worth taking a look at how a mapping to both libraries Ext.JS and MooTools might be achieved.

```
Object.define_class = function(name, desc) {
  var klass         = desc.spec;
  klass.extend      = desc.parent_name;
  klass.constructor = desc.constructor;
  return Ext.define(name, klass);
}
```

Figure 7.28: *Mapping the class-definition to Ext.JS*

The implementation for Ext.JS can be seen in figure 7.28. Here it gets visible how the string, specifying the parent-class, is mapped to the property `extend`, while the constructor is saved as equally named property. Finally, `Ext.define` is called to actually start the class-definition process of Ext.JS.

```
Object.define_class = function(name, desc) {
  var klass        = desc.spec;
  klass.Extends    = desc.get_parent();
  klass.initialize = desc.constructor;
  desc.set_class(new Class(klass))
}
```

Figure 7.29: *Mapping the class-definition to MooTools*

In contrast to the Ext.JS implementation, figure 7.29 shows how a mapping to MooTools can take place. This time the parent object, stored in the closure of `get_parent`, is mapped to the property `Extends` and the constructor is saved as property `initialize`.

Finally, the function `set_class` is used which assigns the newly created class to a variable in the lexical scope of definition.

Figure 7.30 displays the additions to the translator that are necessary to output the neutral class-specification discussed above.

```
ClassStmt :n = walk:constr walk:spec
             -> stmt('Object.define_class("', n.name(), '", {',
                      'parent_name: "', n.parent(), '",',
                      'get_parent: function() { return ', n.parent(), '},',
                      'set_class: function(__class__) { ', n.name(), '=__class__},',
                      'constructor:', constr, ',',
                      'spec:', spec,
                    '})');
```

Figure 7.30: *Implementation of the translator for class-statements*

Following the implementation pattern a last time all children are translated recursively in a first step. Afterwards the `stmt`-helper is used in order to create a subtree representing the given textual input. The usage of the helper is comparable to it's first application at the translation of the `ScopeExpr`. The increased complexity adheres to nothing but the pure number of arguments.

The observant reader may have noticed that we did not discuss the realization of the call to `super` in figure 7.24. We silently jumped over this feature to keep the implementation of the example at a reasonable level. Of course the keyword `super` might be compiled to an invocation of `this.__super` which in turn could be mapped to the appropriate method call, such as `.callParent` for Ext.JS.

In summary, the extension elaborated in this section allows to use a neutral syntax to define classes. This syntax, in combination with a runtime mapping to the desired base-library, offers a lot of possibilities for future configuration and extension.

7.5 Summary

The four extensions presented in this chapter only allow a small peek at the wide range of possibilities offered by the framework. The first extension, forcing the creation of scope, illustrated how a simple extension can be implemented. The second one, string substitution, required a more challenging parser but followed the same model of implementation. The level of complexity has been increased by introducing the third extension, lambda expressions. Both implementations, parser as well as translator have been more sophisticated. In order to allow implicit returns the latter had to call a function which analyzes the contents of the expression body. The last and most advanced syntax-extension equips the programmer with a new way to declare classes and express inheritance. The neutral implementation allows a mapping of the class-declaration to a library of choice (like Ext.JS or MooTools), which is performed at runtime.

Chapter 8

Conclusions

Contents

This thesis showed how language extensions for JavaScript can be created and proposed a framework implementation which may be used as foundation for this very purpose. Such a language extension may be applicable in many contexts like experimentation with future JavaScript features, as a domain specific language within a small company or by individual web developers to increase their productivity.

Nevertheless, the professional usage highly depends on future work and support. "One language for one company" sounds like a great idea at the first glance. All developers of a company may contribute to a great language which fits the usual business perfectly. They may add more and more syntax for common design patterns. But all this may turn into a problem if the one who built the compiler leaves the company. As usual, the question of support and maintenance casts a dark shadow on small projects like this framework.

8.1 Related Work

Of course the language extension framework presented in this thesis is influenced by a lot of related implementations out there. The original implementation of OMeta/JS did have the largest influence of all since I started with modifying the JavaScript parser and translator delivered as part of OMeta/JS.

Furthermore, my framework is not the only JavaScript-Parser implementation written in OMeta/JS. Tom Van Cutsem created a parser which is much closer to the ES5 specification than the parser presented in this thesis. His implementation[1] has been the inspiration for using JsonML as data structure for the AST.

Under the many existing JavaScript extensions CoffeeScript (created by Jeremy Ashkenas) appears to be the most successful one. CoffeeScript adds many features to

[1]The parser can be found at `http://code.google.com/p/es-lab`

JavaScript in order to make working with it less of a burden. Similar to the `EJSParser`, CoffeeScript's parser is automatically generated from a grammar (Instead of using OMeta/JS it uses jison[2] as parser generator). There is one major difference between CoffeeScript and the framework presented here. CoffeeScript represents a language derivate of JavaScript, not a superset. It is not built to *extend* JavaScript but to modify it at large portions. My framework in contrast provides not a complete language but a generic toolkit to easily create an individual JavaScript extension. Nevertheless, CoffeeScript has been a huge inspiration since it demonstrated that a language which compiles to JavaScript can be used in production. Additionally, the idea of registering extensions to the module-system of Node.js in chapter 6 derives from CoffeeScript.

Most comparable implementations can be grouped in two categories. Either they form a completely (or partially) new language like CoffeeScript does or they try to implement the features of future ECMAScript versions in order to experiment with them (One prominent candidate for this category is Google's traceur compiler[3]). My approach in turn is to be more generic in order to be able to experiment with new syntax elements for JavaScript. The presented work is neither a full language nor an implementation of the future JavaScript. Nevertheless, it can be used for both purposes.

8.2 Future Work

Since this work is mainly built on OMeta any improvement on OMeta would also have a direct impact on the language extension framework.

With the rewrite of OMeta/JS as it is presented in chapter 6.1.2 on page 51 some improvements could be achieved. One large drawback that came with the rewrite is an increased time for compilation. In previous discussing the performance of OMeta grammars has been no concern for the presented platform. The compilation from e.g. EJS to JavaScript is only performed once during deployment. Nevertheless, there is much space for improvement at this point. The same applies to error reporting provided by OMeta/JS. The rewrite could improve the error reporting a little. However, PEG.js showed that packrat parsers can support solid error-reporting.

Some design decisions (for instance choosing JsonML over the Object Notation) have been in induced by the fact that OMeta/JS does not yet support the pattern matching of generic objects. A proposal to solve this problem can be found in appendix B.3. After having added the object pattern matching capabilities to OMeta/JS it would be possible to implement a new AST format fully compatible to the Spidermonkey Parser API. This would be not just a cosmetic change but highly increase the chance for a reuse of the AST.

On the other hand, the parsers and translators may be modified to preserve position and whitespace[4] information. The code generator (`ES5Translator`) can be improved to create line-to-line equivalence in order to support a better debugging on generated code.

The implementation of JsonML nodes also may be enhanced by adding more features to perform queries on the children of each node. The inspiration for this querying

[2]`http://zaach.github.com/jison/docs/`
[3]`https://code.google.com/p/traceur-compiler`
[4]Comments are currently also treated as whitespace and hence discarded.

capabilities come from jQuery. Equipped with this functionality one might write rules like:

```
rule :n = ?n.find('> VarBinding').empty() walk+:children -> // No direct declarations
```

As a result, this could combine the elegance and expressive power of both worlds, grammars and query languages.

Finally, another approach of creating a code generator has been presented by Kaehler and Warth [12]. They describe how an OMeta parser may be applied in reverse to generate code from a syntax tree. The information which is not stored within the AST has to be extracted from the parsing rules. This strategy sounds very promising and could be used to remove the need to write a translator by hand in order to further simplify the usage of the framework.

Appendix A

Code Samples

Contents

A.1 PEG-Grammars

```
grammar Lisp

  Program    <- Atom

  Atom       <- Pair
             / nil
             / number
             / string
             / identifier

  Pair       <- ( "(" _ car:Atom _ "." _ cdr:Atom _ ")" )

  nil        <- ( "nil" )

  number     <- ( [0-9]+ )

  string     <- ( "\"" [^"]* "\"" )

  identifier <- ( [a-zA-Z_$]+ )

  _          <- [ \t\n]*
```

Figure A.1: *PEG-Grammar for Canopy*

```
Program    = Atom EOS

Atom       = Pair
           / nil
           / number
           / string
           / identifier

Pair       = '(' _ Atom _ '.' _ Atom _ ')'

nil        = 'nil'

number     = [0-9]+

string     = '"' [^"]* '"'

identifier = [a-zA-Z_$]+

_          = [ \t\n]*

EOS        = !.
```

Figure A.2: *PEG-Grammar for Language.js*

```
ometa Lisp {

  Program    = Atom,

  Atom       = Pair
             | nil
             | number
             | string
             | identifier,

  Pair       = '(' _ Atom:car _ '.' _ Atom:cdr _ ')'
             -> [car, cdr],

  nil        = 'nil'
             -> { type: 'nil' },

  number     = digit+:digits
             -> { type: 'num', val: parseInt(digits.join(''))},

  string     = '"' (~'"' anything)*:chars '"'
             -> { type: 'string', val: chars.join('')},

  identifier = (letter | '_' | '$'):chars
             -> { type: 'id', val: chars.join('')},

  _          = space*

}
```

Figure A.3: *PEG-Grammar for OMeta/JS*

```
Program    = Atom EOS

Atom       = Pair
           / nil
           / number
           / string
           / identifier

Pair       = '(' _ car:Atom _ '.' _ cdr:Atom _ ')'
           {return [car, cdr]}

nil        = 'nil'
           {return { type: 'nil' }}

number     = digits:[0-9]+
           {return { type: 'num', val: parseInt(digits.join(''))}}

string     = '"' chars:[^"]* '"'
           {return { type: 'string', val: chars.join('')}}

identifier = chars:[a-zA-Z_$]+
           {return { type: 'id', val: chars.join('')}}

_          = [ \t\n]*

EOS        = !.
```

Figure A.4: *PEG-Grammar for PEG.js*

A.2 Visitor Based Implementation

```
var Translator = new JsonMLWalker({
  Number: function(n) {
    return n.value();
  },
  VarDecl: function(n) {
    var bindings = this.walkAll(n.children());
    return ['var ', bindings.join(', ')].join('');
  },
  VarBinding: function(n) {
    if(n.size() == 1) {
      var init = walk(n.first())
      return [n.name(), '=', init].join('');
    }
    return n.name();
  }
});
```

Figure A.5: *Manual implementation of a translator for variable declarations*

A.3 Recursive Descent Lisp Parser

```
function Program() {
  var result, next = peek().type;
  if(next === '(') {
    result = List();
  } else if(isAtom(next)) {
    result = Atom();
  } else {
    throw "Expected list or atom, got " + next + "at position" + pos();
  }
  consume('eos');
  return result;
}

function List() {
  var first, last;
  consume('(');
  first = ListItem();
  consume('.');
  last = ListItem();
  consume(')');
  return [first, last];
}

function ListItem() {
  var result, next = peek().type;
  if(next === '(') {
    result = List();
  } else if(isAtom(next)) {
    result = Atom();
  } else {
    throw "Expected list or atom, got " + next + "at position" + pos();
  }
  return result;
}

function isAtom(type) {
  return type === 'id' || type === 'number' || type === 'nil';
}

function Atom() {
  var result, next = peek();
  if(next.type === 'id') {
    consume('id')
    return next.value;
  } else if(next.type === 'number') {
    consume('number');
    return parseFloat(next.value);
  } else {
    throw "Expected id or atom, got " + next + "at position" + pos();
  }
}
```

Figure A.6: *Implementation of a recursive descent lisp parser*

A.4 Implementation of Implicit Returns

```
EJSTranslator.helpers = {
  implicit_return: function(body) {
    var allowed    = ['String', 'Id', 'Number', 'Function'],
        statements = body.children();
    if(statements.length == 0)
      return body;

    var last = statements.pop(),
        type = last.type();
    // only implicitly return allowed values (only expressions)
    // naming convention helps: Everything, that ends with Expr
    if(allowed.indexOf(type) !== -1 || type.match(/Expr$/))
      last = nodes.ReturnStmt(last);
    statements.push(last);
    return nodes.BlockStmt(statements);
  }
}
```

Figure A.7: *Helper-method of `EJSTranslator` to allow implicit returns*

Appendix B

OMeta/JS

Contents

B.1 OMeta/JS - Required Files for Compilation

Filename	Description
`lib.js`	Basic library functions like string buffering, stream objects and string escaping
`ometa-base.js`	The heart of OMeta/JS, the implementation of the `OMeta` base object all grammars inherit from
`bs-js-compilers.js`	A parser and translator for a subset of JavaScript, written in OMeta and compiled to JavaScript
`bs-ometa-compiler.js`	A parser and translator for OMeta-language, also written in OMeta and compiled to JavaScript
`bs-ometa-optimizer.js`	Different optimizing translators, designed to work on the parser output of `BSOMetaParser`
`bs-ometa-js-compiler.js`	Merges the parsers and translators for JavaScript and OMeta (`BSOMetaJSParser` and `BSOMetaJSTranslator`)

Table B.1: *All files required for compilation of OMeta/JS grammars*

B.2 OMeta Base Grammar

```
ometa Base {

  anything                  = !(this.bt.comsume())

  pos                       = !(this.bt.pos()),

  apply                     = :rule !(this._apply(rule)),

  // derived rules
  exactly           :wanted = :got ?(wanted === got),

  end                       = ~anything,

  empty                     = !(true),

  true                      = :obj ?(obj === true),

  false                     = :obj ?(obj === false),

  undefined                 = :obj ?(obj === undefined),

  number                    = :obj ?(typeof obj === 'number'),

  string                    = :obj ?(typeof obj === 'string'),

  char                      = :obj ?(typeof obj === 'string' && obj.length === 1),

  range           :from :to = char:x ?(from <= x && x <= to)   -> x,

  digit                     = range('0', '9'),

  lower                     = range('a', 'z'),

  upper                     = range('A', 'Z'),

  letter                    = lower | upper,

  letterOrDigit             = letter | digit

  space                     = char:value ?(value.charCodeAt(0) <= 32),

  spaces                    = space*,

  token                  :t = spaces seq(t),

  firstAndRest :first :rest = apply(first):f (apply(rest))*:r  -> [f].concat(r),

  listOf       :rule :delim = apply(rule):f
                              ( token(delim) apply(rule) )*:r  -> [f].concat(r)
                            | empty -> [],

  fromTo          :from :to = <seq(from) ( ~seq(to) char )* seq(to)>,

  notLast             :rule = apply(rule):r &(apply(rule)) -> r

}
```

Figure B.1: *Grammar implementing methods from the OMeta-base*

Please note that the rule range is not included in OMeta by default but has been added to allow a more convenient implementation.

B.3 Object Pattern Matching in OMeta/JS

As seen in section 5.1 (Pattern Matching) it can be difficult to match an object in OMeta/JS, though it is possible. For example some minor workarounds are necessary in order to match person-objects like:

```
var person = {
  name: "Alice",
  age: 32
}
```

The name is just a combination of letters while the age should be any positive numerical value. To allow matching these objects a grammar might be written as in figure B.2. Despite the fact that this implementation is simplified and therefore does not work[1] actually, it still looks more difficult than it is.

```
ometa Person {
  identifier = <letter+>,
  number     = ^number:n ?(n > 0) -> n,
  person     = anything:p identifier(p.name) number(p.age) -> p
}
```

Figure B.2: *Grammar to match person-objects*

The call to `identifier` with one argument results in `p.name` to be pushed into the input stream before the rule is applied. Since arguments are handled the exactly same way as normal input (Actually they just *are* normal input) the rules `number` and `identifier` do not need to expect special parameters. Consequently, the passed arguments are simply used as upcoming input.

For special object-types like lists, characters, numbers and strings there are special notations provided by OMeta/JS. On the other hand, JavaScript objects consisting of properties cannot be matched that easy.

In addition there are three semantically equal ways to express a semantic action:

1. The arrow notation`-> ...`
2. The side-effect notation `!(...)`
3. The curly-brace notation `{...}`

While this might cause some problems with existing code it might be reasonable to drop the third alternative in order to favor generic object matching expressions. JavaScript objects mostly are created by using the object-literal syntax as seen in the introductory example. This notation is close to the current curly-brace notation for semantic actions. After removing support for the latter, a pattern matching for objects and their properties could be incorporated as seen in figure B.4.

In this example the need for all semantic predicates is replaced by the alternative curly brace notation. Because JavaScript programmers are already familiar with this syntax

[1] The reason for this is the fact that `p.name` is not converted into an input-stream, so the characters cannot be matched individually

```
ometa Person {
  identifier = <letter+>,
  number     = ^number:n ?(n > 0) -> n,
  person     = { name: identifier, age: number }
}
```

Figure B.3: *Improved grammar to match person-objects, using generic object-pattern matching*

the idea to use object-literals for pattern matching of objects seems obvious. Please note that `identifier` and `number` are both applied in context of the OMeta-language, not JavaScript. An OMeta/JS grammar to parse those expressions is trivial and can be seen in figure B.4.

```
objectMatch   = "{"  listOf(#propertyMatch, ',') "}"
propertyMatch = identifier:n ":" ometaExpr:x
```

Figure B.4: *Grammar to parse object-pattern matching expressions*

Advanced Usage

To reveal the power of the novel object-pattern notation we might take a look at a slightly more complex example grammar (fig. Enhanced grammar to match person-objects) which matches only persons older then 21. Those persons optionally may contain a second person as property `partner`.

```
ometa Person {
  identifier      = <letter+>,
  older number:n  = number:i ?(i > n) -> i,
  person          = { name: identifier, age: older(21), partner: person? }
}
```

Figure B.5: *Enhanced grammar to match person-objects*

The rule `person` in the above example combines the usage of rules, parametrized rules and optional rules within a single object-pattern matching expression. In addition, it invokes itself recursively inside of property-match `partner`. This recursive match has to be optional - otherwise only endless partner-chains would be a valid match.

In personal communication with Alessandro Warth, the creator of OMeta, he fully agreed with the idea of adding support for object pattern matching but was concerned about breaking backwards compatibility.

Bibliography

[1] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321486811

[2] Ashkenas, Jeremy: *List of languages that compile to JS*. (Online; as of December 20th, 2011). – `https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS`

[3] Crockford, Douglas: *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. – ISBN 0596517742

[4] Crockford, Douglas: *Prototypal Inheritance in JavaScript*. (Online; as of November 29th, 2011). 04 2008. – `http://javascript.crockford.com/prototypal.html`

[5] Ecma International: *ECMAScript Language Specification, Standard ECMA-262 5th Edition*. 12 2009

[6] Eich, Brendan: *Popularity*. (Online; as of November 28th, 2011). 04 2008. – `http://brendaneich.com/2008/04/popularity`

[7] Eich, Brendan: *strawman:block_lambda_revival*. (Online; as of January 17th, 2012). 01 2012. – `http://wiki.ecmascript.org/doku.php?id=strawman:block_lambda_revival`

[8] Ford, Bryan: Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. In: *SIGPLAN Not.* 37 (2002), September, S. 36–47. – ISSN 0362-1340

[9] Ford, Bryan: Parsing expression grammars: a recognition-based syntactic foundation. In: *SIGPLAN Not.* 39 (2004), January, S. 111–122. – ISSN 0362-1340

[10] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1994. – ISBN 0201633612

[11] Griffiths, David: *altJS compile-to-JavaScript language list*. (Online; as of December 20th, 2011). – `http://altjs.org/`

[12] Kaehler, Ted ; Warth, Alessandro: Running OMeta Parsers Backwards for Source to Source Translation. (2008)

[13] Maffeis, Sergio ; Mitchell, John C. ; Taly, Ankur: Object Capabilities and Isolation of Untrusted Web Applications. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010 (SP '10), S. 125–140. – ISBN 978-0-7695-4035-1

[14] Maffeis, Sergio ; Taly, Ankur: Language-Based Isolation of Untrusted JavaScript. In: *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, 2009, S. 77–91. – ISBN 978-0-7695-3712-2

[15] Mozilla Foundation: *JavaScript Reference — Mozilla Developer Network*. (Online; as of November 29th, 2011). – `https://developer.mozilla.org/en/JavaScript/Reference`

[16] Mozilla Foundation: *Parser API - MDN*. (Online; as of January 5th, 2012). – `https://developer.mozilla.org/en/SpiderMonkey/Parser_API`

[17] Parr, T. J. ; Quong, R. W. ; Dietz, H. G.: *The Use of Predicates in LL(k) And LR(k) Parser Generators*. 1993. – School of Electrical Engineering, Purdue University, West Lafayette

[18] Parr, Terence: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. – ISBN 193435645X, 9781934356456

[19] Pepper, Peter: *LR Parsing = Grammar Transformation + LL Parsing - Making LR Parsing More Understandable And More Efficient*. 1999

[20] Redziejowski, Roman R.: Some Aspects of Parsing Expression Grammar. In: *Fundam. Inf.* 85 (2008), January, S. 441–451. – ISSN 0169-2968

[21] Reinke, Claus: *Javascript development tools*. (Online; as of December 20th, 2011]. – `http://clausreinke.github.com/js-tools/resources.html`

[22] Richards, Gregor ; Lebresne, Sylvain ; Burg, Brian ; Vitek, Jan: An analysis of the dynamic behavior of JavaScript programs. In: *SIGPLAN Not.* 45 (2010), June, S. 1–12. – ISSN 0362-1340

[23] Soller, Stephan: *Concepts of the Lagrange Programming Language*. Bachelorthesis, Stuttgart Media University (HdM). 02 2012

[24] Soshnikov, Dmitry: *ECMA-262-3 in detail. Chapter 1. Execution Contexts.* (Online; as of February 14th, 2012). – `http://dmitrysoshnikov.com/ecmascript/chapter-1-execution-contexts`

[25] Soshnikov, Dmitry: *ECMA-262-5 in detail. Chapter 0. Introduction.* (Online; as of February 14th, 2012). – `http://dmitrysoshnikov.com/ecmascript/es5-chapter-0-introduction`

[26] Ungar, David ; Smith, Randall B.: SELF: The power of simplicity. In: *LISP and Symbolic Computation* 4 (1991), S. 187–205. – 10.1007/BF01806105. – ISSN 0892-4635

[27] Warth, Alessandro: *Experimenting with Programming Languages*, University of California, Dissertation, 2009

[28] Warth, Alessandro ; Douglass, James R. ; Millstein, Todd: Packrat parsers can support left recursion. In: *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2008 (PEPM ’08), S. 103–110. – ISBN 978-1-59593-977-7

[29] Wikipedia: *Comparison of parser generators — Wikipedia, The Free Encyclopedia*. (Online; as of December 21th, 2011). 2012. – `http://en.wikipedia.org/w/index.php?title=Comparison_of_parser_generators&oldid=477079386`

[30] Zakai, Alon: Emscripten: an LLVM-to-JavaScript compiler. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011 (SPLASH '11), S. 301–312. – ISBN 978-1-4503-0942-4

[31] Zakas, Nicholas C.: *High Performance JavaScript*. 1st. USA : Yahoo! Press, 2010. – ISBN 059680279X, 9780596802790

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CFG	Context Free Grammar
DOM	Document Object Model
EBNF	Extended Backus-Naur Form
EJS	Extended JS or Example JS
ES5	ECMAScript edition 5
JIT	Just in Time
JS	JavaScript
JSON	JavaScript Object Notation
JsonML	JSON Markup Language
LLVM	Low Level Virtual Machine
OO	Object Orientation
PEG	Parsing Expression Grammar
RE	Regular Expressions
SQL	Structured Query Language