STAGED JAVASCRIPT

Yannis Apostolidis

Heraklion, November 2015

The work reported in this thesis has been conducted at the Human Computer Interaction (HCI) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship.

University Of Crete
Computer Science Department

STAGED JAVASCRIPT

Thesis submitted by
Yannis Apostolidis
In partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

Author	____
	Yannis Apostolidis, Department Of Computer Science

Committee approvals:

Thesis Supervisor	____
	Anthony Savidis, Professor
Member	____
	Yannis Tzitzikas, Assistant Professor
Member	____
	Irini Fundulaki, Principal Researcher of ICS-FORTH
Department approval	____
	Angelos Bilas, Professor, Director of Graduate Studies

Heraklion, November 2015

Abstract

Metaprogramming is the ability to treat the source code as first-class data and emit it to the program. In Multi-stage metaprogramming the above procedure can be repeated in infinite nesting level. Currently, metaprogramming languages separate the normal program from the metaprogram, distinguishing each environment with a different programming language.

In our work, we integrate the metaprogram with the normal program denoting that metaprograms are essentially normal programs. We extend JavaScript with multi-staged metaprogramming and implement it in SpiderMonkey JavaScript engine. We indicate the minimal modification occurred in SpiderMonkey by minor extending the lexer, parser, AST, reflection (parse) and the addition of the unparsing procedure, some library functions, the staged logic and a debugging system. Our approach handles each stage as an isolated entity by collecting all the meta-annotations for the specific depth, respecting the order, and creating a coherent program.

Furthermore, we pass the boundaries of smaller-scale macro-related examples appearing in the literature, by implementing a large scale application scenario, called Litemail, an elementary email client. In this context, we discuss the MDE maintenance issues and offering a solution that exploits the metaprogramming principles. Additionally, we are studying JavaScript source code segments from the real world and propose metaprogramming modifications to achieve reusability and runtime performance. We identify that design patterns are not currently a strictly definition in the modern programming languages, hence we methodologically implement reusable design patterns.

Finally, we identify that modern web browsers have developed sophisticated and powerful JavaScript execution systems. A part of these systems are the JavaScript debugging tools. We exploit the possibilities of those utilities by exporting the metaprogramming multi-staged execution system in the browser environment. We have implement a debugger client as a web page application that communicates via

Ajax requests with the extended SpiderMonkey backend debugger. The client provides features that are valuable to the metaprogramming debugging process, as proved in our case study development lifecycle. It offers an AST displaying mechanism that inspects variables currying source code and useful staged information.

Περίληψη

Το Abstract στα Ελληνικά…

Acknowledgements

First of all, I would like to thank my supervisor, professor of the University of Crete, Anthony Savidis, initially for trusting me and then for his continuous support and his valuable advice. I am also grateful to the professors … and … for participating in the supervisory committee. I would also like to thank the Computer Science Department of Greece for offering a high level of academic education and the HCI Laboratory of ICS-FORTH for providing a high-level research environment.

I would also like to thank my friends for supporting me all through this period. Finally, most of all, I would like to thank my parents Eleni and Panayiotis, and my brother Pantelis. I am grateful for all their love and support. Without them, I would not be the person I am today.

TODO: review Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An algorithm is defined as a finite raw of steps, strictly defined and executable, in finite time that is targeting to solve a specific problem. Essentially, algorithm is the foundation notion under the process of programming. Distinguishing a specific problem to finite steps is a process that taking place only once and can reused forever. Thus, we are aiming to solve once a common appeared problem. In the middle of the twenty century, exploiting the definition of algorithm, we came up with the procedure of computer programming. Programming is the process that leads to executable computer program using an algorithm formulation. Many steps are interposed in the programming process. Briefly, some elementary ingredients are the analysis, developing, understanding, generating algorithms, validate the correctness of algorithms, implementation, testing and evaluation.

The infant steps in our attempts to generate more readable and reusable programs was the passage from machine code to Assembly. Programmers had the requirement to read the source code in a more legible form. Thus, they moved from a pure machine representation of the source program to a more readable form. Assembly is using symbolic tokens that are mapped to machine code using a utility program referred as assembler. At that point, programmers had the ability to directly read the core programming notions in a human readable form, in contrary with the machine representation that consists from a batch of zero and one sequences. Programmers observe the repetition patterns in Assembly and was trying to reuse the code snippets, inverting function call and jump design patterns in combination with meaningful comments. As the programs grows, Assembly was beginning to be non-manageable. The unlimited lines of Assembly code was an enormous obstacle in the program developing. The integration of more expressive tools for the development process was imperative. Hence, the first generation of programming languages occurred. Many programming languages appeared aiming to fill the developing gap. A well-known example is the general-purpose programming language C. In C we meet up notions such as structure programming, lexical variable scoping and recursion. Programmers rapidly raise the source code length, writing larger amount of source code lines in compared with Assembly. There, we observe the first thoughtful usages of metaprogramming. The C preprocessor is a text-based preprocessor that is independent from the C compiler. Programmers are using the preprocessor to include files, compile selected source code segments, concatenations and insertion of code snippets. C Preprocessor phase can be characterized as metaprogram, thus you can generate a source code that creates a source code. Henceforth, as the source code amount was raising, programmers was investigating techniques for better code organization and reusability. A writing motive practice observed, as programmers was trying to associate particular data with functions. They was creating data structures that contain data and code in the form of procedure, as a solitary entity, using advanced programming techniques. The programming languages category that generated to fill this gap called **Object-oriented (OOP).** Widely used programming languages include Python, C++, Java and PHP belongs to OOP family. Almost the entire variety of the current applications are written in OOP languages. Hereafter, while programmers was developing algorithms and data types in OOP form, a reusable issue appeared. There was not a straightforward solution in case of the requirement to apply an equivalent algorithm to a different data type. Programmers was either replicating the algorithm or trying to abstract the content using abstruse and tricky techniques. **Generic Programming** provides the solution to the problem of algorithm and data type abstraction. It abstracts the fundamentals requirements on types, across concrete examples, instances of algorithms and data structures, using generic mechanisms. Taking effort of all the above technologies and stepping forward to a higher abstract level, **Design Patterns** are used by programmers as reusable solutions to commonly occurred problems within a given context. Specifically, a design pattern is not a specific scheme that can directly transformed into source code. It is a recipe that describe the methodology to resolve a problem, in a manner to be possible applied in several different circumstances. As we have been taught by the software developing history, when advanced programmers massively begin to design and write source code using a specific technique, programming languages are trying to incarnate it. For example, when programmers was trying to embody data and code in one entity, the OOP introduced. Design Patterns are the next innovation, thus programmers are developing source code using specific techniques. In this context, there is a problem that had not occurred in the pipeline of programming history. Programming languages have rapidly grow up and the quantity of Design patterns is huge. Additionally, new software engineering requirements came up for code reusability. We are shifting to the capability of manipulating the source code as pure first-class language variable, hence we can generate, concatenate and manipulate every segment of the final source code. We are aiming to raise up the software abstraction technique. Instead of waiting to note if the programming languages will assimilate the current needs, we are exploiting the advantages of **multi-staged metaprogramming**.

1.1. Multi-staged Metaprogramming

Multi-stage programming languages [14] incarnate meta-programs that is actually normal programs and evaluate them in a process before the actual interpretation or compilation of the final program. In many cases, the execution of a meta-program evaluates to an Abstract Syntax Tree (ASTs) that substitutes in the original program. Since the above procedure is a characteristic of macros systems, the term *macro* is also used to denote a generative meta-program. The main staging process is illustrated in Figure 1. Essentially, it is the ability to treat the source code of programs as first-class values, in a manner to read, generate, compose, and transform the source code of other programs. Briefly, to *create a program that creates a program* in infinite depth.

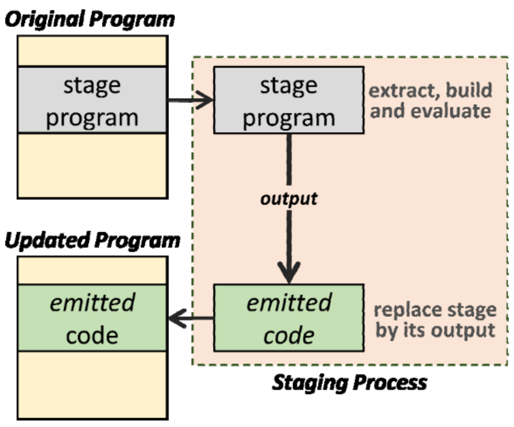


Figure 1 - Stage-evaluation process in multi-stage languages

Furthermore, we represent a variety of different application fields of metaprogramming:

- Code analysis
- Aspect-oriented programming
- Exception handling
- Source code validation
- Model-Driven engineering
- Custom source code manipulation

1.2. JavaScript

Multi-staged metaprogramming can adopted in any languages. We choose to implement the metaprogramming on a current widely used language. JavaScript is standardized in the ECMAScript language specification. It is a dynamic prototype-based object-oriented programming language. Initially used as the scripting programming language of the web, commonly used as an integral part of web browsers. By the years, as the web users increased, novel and faster JavaScript implementations developed. Targeting to accelerate the execution time, many mechanisms incarnated in the language implementation stack, included Just-In-Time compilation (JIT), bytecode generation, dead code eliminations, loop-invariant code motion and fold constants. JavaScript is currently used in many applications fields. Usually, the applications that embed JavaScript, keeps the language functionality invariant and inject a custom object, providing access to the internal structures. We briefly mention the most known implementations on each application field at Table 1.

Application field	**Implementation**	**Details**
OS for desktop	https://node-os.com	An operating system build entirely in JavaScript and managed by npm
OS for mobile/TV	https://www.tizen.org	An operating system based on the Linux kernel, providing application development tools based on the JavaScript
Robotics	http://sbstjn.github.io/noduino	A simple and flexible JavaScript and Node.js Framework for accessing basic Arduino controls from Web Applications using HTML5, Socket.IO and Node.js
Windows/Linux/ Web/Phone apps all in one	http://electron.atom.io	A framework lets you write cross-platform applications using JavaScript, HTML and CSS. It is based on io.js and Chromium
Graphics	http://voxeljs.com	A collection of projects to create 3D voxel games like Minecraft all in the browser running WebG
Graphic card drivers	http://www.slideshare.net/jarrednicholls/javascript-on-the-gpu	Running JavaScript on GPU
Game engines	http://docs.gameclosure.com	HTML5 JavaScript game development kit compatible on Browsers, Android,

		IOS
Server	https://nodejs.org	A platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications

Table 1 - JavaScript application fields

1.3. SpiderMonkey

JavaScript engine has many implementations in different platforms and programming languages. We cannot admit that a specific implementation is more powerful than other. Each of the current JavaScript engines carries different advantages and disadvantages. We choose to extend the SpiderMonkey JavaScript engine. SpiderMonkey initially developed at Netscape Communications. Now it is an open source project maintained by the Mozilla Foundation. Spidermonkey embedded in various well-known applications like Mozilla Firefox, Thunderbird, Adobe Acrobat, Dreamweaver and Yahoo widgets and GNOME 3. Internally, it is written in C/C++ (cross-platform) and contains an interpreter, a JIT compiler (IonMonkey), a garbage collector and mechanisms like fold constants. We thoroughly choose to implement metaprogramming in Spidermonkey because it is open source, widely used and clearly documented.

1.4. Implementation

During the design and the implementation of the multi-staged model, we was aiming to convenience the user of the language, but also simplify the implementation process for the language author. To address those challenges, we focus on the maximum reuse of the current language tools, avoiding to reinvent the wheel. Putting all the requirements on a straight line, we implement a multi-stage metaprogramming extension on top of Spidermonkey JavaScript engine. As we outline in Figure 1, we thoroughly implement some minor extensions. As a result, we yielded a full-scale multi-stage metaprogramming system for JavaScript. Furthermore, we enrich the staging model with extra features, introduced in [15]. Our work is as expressive as macros, improved with an extra staging tag targeting to support larger-scale metaprograms.

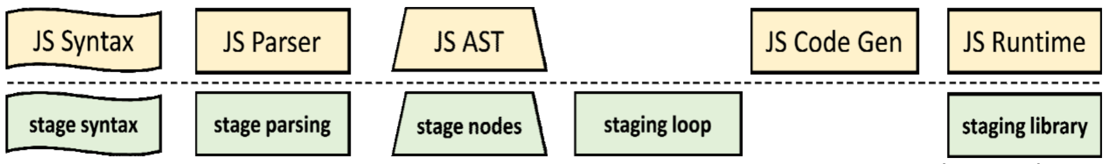


Figure 2 - Outline of the original JavaScript components in the SpiderMonkey implementation (top layer) and the respective extensions we have introduced to support staging (bottom layer).

We enabled the use of browser debugging tools, for debugging stages and the final JavaScript program, through a JavaScript wrapper that is playing the role of the debugger front-end. This client support: (i) inspection and visualization of ASTs, with multiple views and code unparsing; and (ii) seamless transition to the next stage with the normal step debugging command issued on the last statement of the current stage.

Chapter 2

Related Work

Our approach operates as a multi staged macro system. In bibliography, we meet a huge variety of macro systems, several with staged techniques. Considering the acne of JavaScript in the latter years, tones of ink has spilled to solve erroneously practices, generate optimizations, organize the source and extend the language. We focus on multi staged macro approach, considering that we minimally extend the SpiderMonkey engine. Our work, wisely reuse all the engine's features, for instance, bytecode and runtime system remain unaffected.

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output according to a defined procedure. The mapping process that instantiates a macro, used inside a specific sequence, is known as macro expansion. A typical macro system evaluation depicts in Figure 3. There are two kind of macros, (i) Text-Based; and (ii) Syntax-Based.

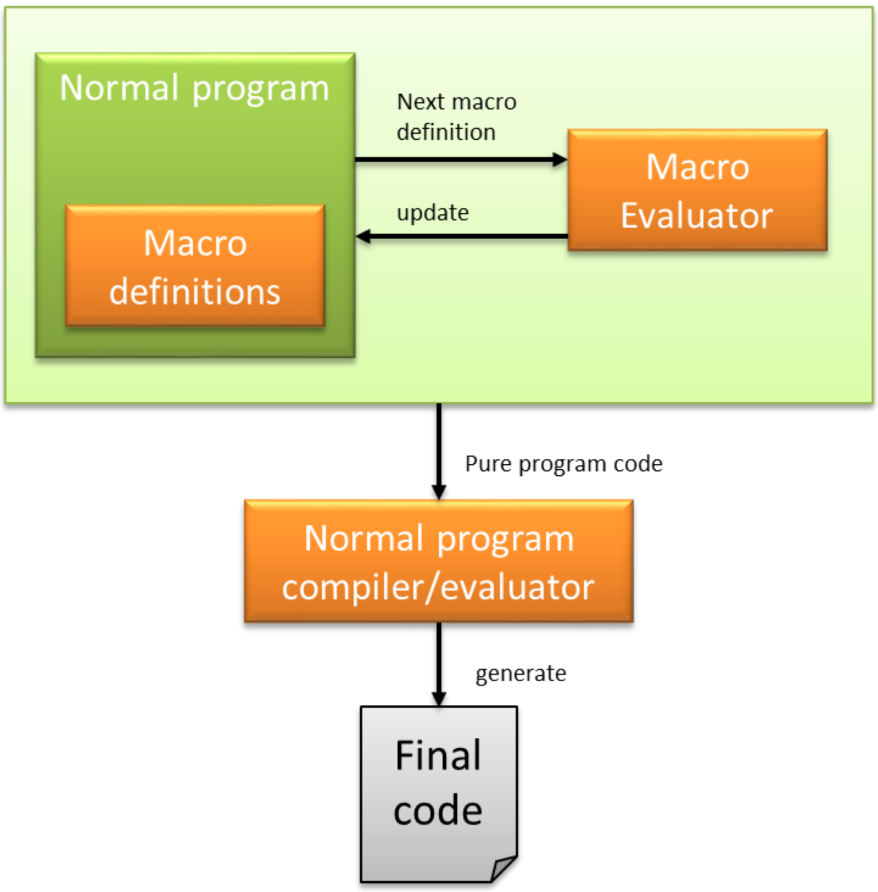
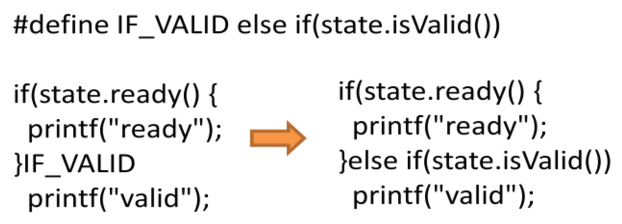


Figure 3 – Typical macro system evaluation process

2.1. Text-Based Macros

Text-Based Macros are external preprocessors that are usually independent from the programming language. Especially, they lack of knowledge about the host language. Text-Based macros work by simple textual search-and-replace at the token, rather than the character level. They process the target source as a pure text file and performs text substitutions, generating the final source file. The language execution environment is unaware about the appearance of the preprocessor. Observably, the possibility of side effects and erroneous results in extremely probable. Text-Based preprocessors does not provide any safety about the substitution environment. There is no guaranty about the source code stability and the result of bindings in the enclosed environment.

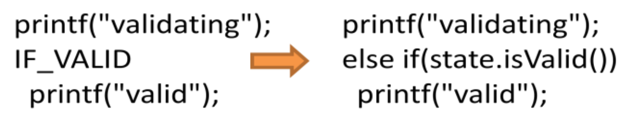
The most popular and representative Text-based preprocessor is the CPP preprocessor. The C preprocessor or CPP is the default macro preprocessor for the C and C++ computer programming languages. The preprocessor performs preliminary operations on C and C++ files before they are passed to the compiler. You can use the preprocessor to conditionally compile code, insert files, specify compile-time error messages, and apply machine-specific rules to sections of code. In practice, it is a separate program invoked by the compiler as the first part of translation. A typical preprocess example is illustrated bellow:



```
#define IF_VALID else if(state.isValid())

if(state.ready() {               if(state.ready() {
  printf("ready");                 printf("ready");
}IF_VALID                        }else if(state.isValid())
  printf("valid");                 printf("valid");
```

The preprocessor will correctly produce an if-else-if statement. In contrary, a misuse case may generate an invalid source code. The preprocessor, as a pure Text-Based tool, will not identify the inaccuracy of the result. Subsequently, the error will be appeared at the later compiler steps.



```
printf("validating");            printf("validating");
IF_VALID                         else if(state.isValid())
  printf("valid");                 printf("valid");
```

2.2. Syntax-Based Macros

Syntax-Based macros encountering the substitutive source code as an Abstract Syntax Tree (AST) segment. Usually, they transform the macro expressions to ASTs, in the parsing phase of the programming language. The parser identify the AST, evaluate it and trying to substitute it to the enclosing AST. A further step in the macro procedure that Text-Based does not offer, is the insurance that the macro generated AST will only substituted, if the enclosed AST is matching syntactically. For example, a macro that is enclosed by a while block statement is expecting a statement, in case that the macro generated AST is different, an error will occurred. This procedure offer a proper safety and guaranties for the macro substitution well-formless.

An interesting Syntax-Based macro approach is **Sweetenjs** [1]. Sweetenjs proposed a novel solution for solving the bidirectional dependency issue between lexical and parsing phase in the JavaScript implementations. They argue that the specific disambiguate methodology can applied to any other language. After the separation of lexical and parsing phase, they implement a hygienic macro system for JavaScript. Especially, they insert an extra component between the lexical and the parsing phase, called *reader*. Reader takes as input tokens from the lexer and constructing a token tree. Autonomously decides when a token tree is ready. Finally, the token tree is intended to be the input of the parsing phase. This procedure cut the repeatedly inform dependency circle between lexer and parser. Considering previous works, they introduce the term enforestration [3], which is a methodology to convert a flat stream of tokens into an S-expression form. This S-expression form consist the token tree. The first appearance of enforestration technique is in **Honu** programming language [4]. Additionally, they implement a hygienic macro system for JavaScript. The macro system programming language is different from JavaScript. It is a custom language that provides the ability to extend the JavaScript syntax. The extension syntax is the following,

```
macro <name > {
  rule { <pattern > } => { <template > }
}
```

Exploiting the advantages of this syntax, we have the ability to develop many interesting expressions. For instance, we can generate a function declaration expression like:

```
macro def {
  rule {
    $name ( $params ( ,) ...) { $body ... }
  } => {
    function $name ( $params ...) {
     $body ...
  }
}
```

After the specific declaration, we can use this function definition form, as a macro, to the source code:

```
def id (x) { return x; }
//expands to:
function id (x) { return x; }
```

ExJS [13] is an extension of JavaScript that offers syntactic extensibility of the core language through hygienic syntactic macro facility. This macros are defined by a simple pattern matching technique and are similar to syntax-rules pattern that appears in Scheme. ExJS uses a multi-staged parsing technique and generate context-dependent syntax rules using the parsing expression grammar technology. The parsing pipeline process can outlined by the following steps:

a) Analyze the syntax of macro forms.

b) Generate PEG grammars.

c) Combine JavaScript and macro to a macro-aware form.

d) Parse the macro-enabled program.

e) Convert the AST to S-expression.

f) Macro-expand the AST to S-expression.

g) Convert the S-expression back to macro-free JavaScript.

Take advantage of ExJS, programmer can generate new expressions and statements in JavaScript using a special syntax. For example,

```
statement unless {
  expression : C;
  statement : S;
  { unless (C) S => if (!C) S }
}
```

is a macro directive that denotes a custom *unless* statement in JavaScript. In the illustrated example you can detect the *unless* syntax parts that consists from an expression C, a statement S and the corresponding macro expansion that is indicated by the left and right fragments of symbol =>. Henceforth the above declaration, programmer can produce statements like:

```
unless(key=="A"){
  console.log("A not pressed");
  if(key=="B")
    console.log("B pressed");
  else
    console.log("A and B not pressed");
}
```

After the ExJS preprocess, the specific expression is expanded to:

```
if(!key=="A"){
  console.log("A not pressed");
  if(key=="B")
    console.log("B pressed");
  else
    console.log("A and B not pressed");
}
```

2.3. Metaprogramming

Haskell [5] is a standardized, strongly-typed, general-purpose purely-functional programming language. It support compile-time meta-programming through some semantics similar to Staged-JavaScript. Haskell macro system is inlining source code to the main program. The macro substitution procedure is taking place at compile time. Haskell generates macros by creating the AST from the corresponding source code, manipulating and substituting in the context. Metaprogramming facilities are implemented by reusing all the pure language features. The ASTs are embodying in the language environment as Haskell data types. **Template Haskell** adopts two basic semantics. *Quazi-quote* brackets, illustrated by [| ... |], are used to get the corresponding AST from the enclosing expression. *Splice* brackets, $(...), used to convert the AST to source code.

Converge [6] is the dynamically typed object orientated language which allows compile-time meta-programming in the spirit of Template Haskell. Quasi-quote, [| ...

|], produces AST. In Converge, ASTs called ITrees. ITrees respect the source code enclosed inside the expression and the language scoping rules. Similar with Template Haskell, Converge contains the splice annotation $<<...>>. Splice evaluates its enclosed expression at compile-time and substituting itself with the ITree yielded from the evaluation. Exploiting Quazi-quotes and splicing functionality, many interesting preprocesses can take place. Noteworthy is the fact that compile-time metaprogramming is not effecting the bytecode and the later Virtual Machine instruction generation, since it acts at parsing time. We depict a typical implementation of the function *make_power* in Converge:

```
func expand_power(n, x):
  if n == 0:
    return [| 1 |]
  else:
    return [| $c{x} * $c{expand_power(n - 1, x)} |]

func make_power(n):
  return [|
    func (&x):
      return $c{expand_power(n, [| &x |])}
  |]

power3 := $<make_power(3)>
```

Metalua [7] is an extension of Lua. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine. Essentially, it exposes the generic principles of meta-programming assets like meta-levels in sources, AST representation of code and meta-operators. It provides compile-time metaprogramming and offers the ability to extend the syntax of Lua. Metalua have special individual metaprogramming semantics. Quazi-quotes, +{ ... }, produce the AST of the corresponding enclosed expression and increase the staged level by one. Programmer is unrestricted to produce unlimited staged levels. The splicing functionality produced by the contrary -{ ... } semantic. Splicing expression reduces the staged level by one. The AST will be substituted when -1 staged level occurred. For example,

```
-{stat:
  function ternary (cond, b1, b2)
    return +{ (function()
                    if -{cond} then
                      return -{b1}
                      else
                        return -{b2}
                      end
                  end)() }
  end }

lang = "en"
hi = -{ ternary (+{lang=="fr"}, +{"Bonjour"}, +{"Hello"}) }
print (hi)
// prints "Hello"
```

Additionally, Metalua provides some syntactic sugar extensions,

```
function(arg1, arg2, argn) return someExpr end
```

can be written:

```
|arg1,arg2,argn| someExp
```

To achieve the metaprogramming functionality, Metalua contains a wide variety of tools. We briefly the most essential subset. MLP is the dynamically extensible Metalua parser. GG is the grammar generator. MATCH is an extension supporting structural pattern matching. WALK is a code walker generator. HYGIENE offers hygienic macros. Runtime libraries for staging and AST manipulation (metalua.runtime, metalua.compiler). Despite all these powerful functionalities, Metalua distinguished an embarrassing logic. Although the original pure Lua language components are fully available in stages, its implementation features consists from a separated scanner, parser, and a custom reimplementation of its virtual machine for stage evaluation.

Groovy [8] is an object-oriented dynamic programming language for the Java platform. It is usually used as the scripting language for Java environment. Groovy supports runtime and compile-time metaprogramming. We will focus on compile-time metaprogramming, since runtime metaprogramming is a runtime reflection-related mechanism that JavaScript already encompasses. Compile-time metaprogramming in Groovy allows code generation. It is modifying the AST during the compilation process. The metaprogramming procedure is operating by using semantic annotations at the target source segment. Groovy distinguishes two metaprogramming alternatives. Firstly, providing a huge mixture of AST transformations. Those transformations are covering a large portion of the programmer requirements. Some of these are operating on reducing the boilerplate source code, implementing design patterns, logging, declarative concurrency, cloning, safer scripting, tweaking the compilation, implementing Swing patterns, testing and eventually managing dependencies. For example, we mention a metaprogramming example in the context of *ToString* transformation facilities:

```
import groovy.transform.ToString

@ToString
class Person {
  String firstName
  String lastName
}

def p = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p.toString() == 'Person(Jack, Nicholson)'
```

Secondly, programmer can manually define a custom AST transformation by implementing the *ASTTransformation* interface. This interface provides an entry point to access and manipulating a specific AST. The transformation selection is effected by annotating the source code with the regarding markers.

Groovy metaprogramming implementation reuse the language compiler and the runtime system. Furthermore, it provide the proper debugging mechanisms to inspect the AST transformations directly from the IDE [9]. In contrary, programmer has not the ability to see the exported AST. Moreover, Groove does not support multi-staging, hence, sophisticated staged evaluations, such metagenerators are impossible to be implemented. Additionally, transformation components are isolated, thus, integrated metaprograms that systematically transform the main program are unachievable.

Lisp is a fully parenthesized or s-expressions prefix notation programming language, introducing many primary compute science notions like artificial intelligence research, tree data structures, dynamic typing, self-hosting compiler. A premium data structure is the linked list. The entire Lisp source code consists by lists. Mostly, Lisp implementations share the same interpreter for both language and macros. Its main derivations are **Common Lisp** [11] and **Scheme** [10]. Common Lisp manipulate the source code as data structure using Macros. ASTs can be generated using list processing functions or quasi-quotes directive. Programmer has the ability to traverse and manipulate the generated AST.

```
(defmacro until (test &body body)
(let ((start-tag (gensym "START"))
      (end-tag   (gensym "END")))
   `(tagbody ,start-tag
             (when ,test (go ,end-tag))
             (progn ,@body)
             (go ,start-tag)
             ,end-tag)))
```

```
(until (= (random 10) 0)
  (write-line "Hello"))
```

Lisp yields the surrounding source code to a transformed source form. This procedure repeated until the elimination of macros. The final source code is executed at runtime. Common Lisp provides name capturing and a hygienic macro mechanism that guarantee the uniqueness of variables using the directive *gensym*. Scheme macros are transformation procedures accompanied by a simple pattern matching sublanguage. It provides a hygienic macro system using syntax-case clauses. These clauses offering the ability to selectively apply variable capturing. Language normal source execution traced via the normal debugger. In contrary, programmer must use a special macro stepper to inspect and trace the AST transformations. Generally, studding Lisp macro system in the aspect of metaprogramming, it does not provide you a coherence metaprogram. Programmer treating each macro as an individual expression and cannot combine the macros in the same context. This observation entails that Lisp macro system does not collecting all the macros to a staged coherence metaprogram, decreasing the expressive power.

MetaML [12] is a general-purpose functional programming language that extends the ML programming language. It is reusing the pure Language basic features and tools, like the compiler, the runtime system. MetaML belongs to the multi-staged runtime metaprogramming family that is based on annotations. More specific, it is uses three annotations. Brackets <>, to delay the reduction of the enclosed expression. Escape ~, to splice multiple delayed expressions together and generate a larger expression. Inline *run*, to force the reduction of a delayed expression.

For example the power generator function can be written as,

```
fun power n = fn x => if n=0
  then <1>
  else < ~x * ~(power (n-1) x) >)
```

We can use this function to the following expression,

```
map (run <fn z => ~(power 2 <z>)>) [1,2,3,4,5]
```

after the evaluation of the annotated expression, we yield to,

```
map (fn z => z * z * 1) [1,2,3,4,5]
```

Even the exploitation of the basic language, MetaML is not using the debugging system for the metaprogramming proposes.

Chapter 3

Staged JavaScript

3.1. Semantics

Spidermonkey JavaScript engine has extended to support multi-staged metaprogramming. Staged semantics must be evident and easy to use. The general purpose of this thesis is to achieve hygienic well-formed ease-of-use invocations for the language user. Reflecting these principals, we define staged syntax tags and semantics.

3.1.1. AST Manipulation Tags

Staged programs are injecting AST definitions in the parent program, this procedure is repeating until the appearance of the final program. Essentially, all the staged process is elaborating AST creations and concatenations. To achieve that, we introduce the following AST manipulation Tags. Such tags are provided to ease the composition of the AST and are not relevant with the staged process. Practically, they reify the language parser, the internal AST structures and the AST creation functions.

Quazi-quotes (.< >.) are holding JavaScript code definitions in order to convey them to AST form. It is essentially a syntactic sugar tag that easily convert a source text to AST, assisting the programmer to generate hygienic ASTs, without the boilerplate procedure of generating custom definitions of the corresponding JavaScript AST object. Variables within quasi-quotes are resolved with their names in the context where the respective AST is finally inserted, essentially are lexically scoped at the insertion point.

```
var xDeclAst = .< var x = 1 + 4; >.;
```

```
var xDeclAst = { loc: {
    start: { line: 1, column: 0 },
    end: { line: 1, column: 5 },
    source: null },
  type: "Program ", body: [
  { loc:..., type: "VariableDeclaration", kind: "var", declarations: [
    {  loc: ..., type: "VariableDeclarator",
      id: { loc: ..., type: "Identifier ", name:"x"},
      init:{loc:..., type: "BinaryExpression", operator: "+",
      left: {
        loc: ...,
        type: "Literal",
        value: 1
      },
      right: {
        loc: ...,
        type: "Literal ", value:4}
      }
    }
]} ] }
```

Escape (.~) is defined only inside a Quazi-quote definitions and its argument can only contain JavaScript AST object. It prevents an expression to convert its context to AST by evaluating its current value. Usually, it used to incorporate a previously generated AST to the current Quazi-quote context. For example, assume the statements *.< var x = rand(); .~assertAst; y = x; >.* , and a variable that has already defined *assertAST = .< assert(x); >.* . Our purpose, after the evaluation of that code snippet, is to prevent the variable *assertAst* of converting as it is, but replacing the *.~assertAst* with the carrying *assertAST* value.

Escape-value (.@) is used when we need to prevent the conversion of an expression to AST by matching the current evaluated value to an AST form. So it acts as Escape, in the difference that Escape-value argument is only JavaScript first class value (number, string, Ec.).

3.1.2. Staged Tags

Staged tags generate the metaprogramming logic in the program. Essentially, they define the code that will produce the stage evaluation, the nesting depth and indicates the segments where the staged code will be injected to source.

Inline (.!) injects the declared location with the AST of the evaluated value, eliminating itself. After the evaluation, the expression must be a JavaScript AST object. Inline is actually the core of staging programming because it generates preprocessor sense and modifying the main program. When an Inline expression is defined inside a Quazi-quote, it is behaving as the entire JavaScript syntax. By converting the inline expression to AST, we can produce some interesting and significant functionalities. A very common pattern that appears in metaprogramming is the metagenerators. A metagenerator is a directive that can produce infinite nesting stage by generating other metagenerators statically or dynamically. For example *.!(.<.!.(<print('stage 1').>).>)* will inject to the source the *.!.<print('stage 1').>*, as a result, this expression will generate an additional staging depth that will inject the *print('stage 1')* to the final program.

Execute (.&) defines a statement that will be executed in the staged process. You can swift the execution in any staging depth by appending more Execute symbols. For example, *.&.&.&statement* will be executed in the stage depth 3.

In our approach, the environment of each staging depth execution is isolated. That means that each staged depth behaves as a unique program. The only communication between each stage is the potential injection of the enclosed inline. The execution order of the Execute and Inline statements in each staged depth is corresponding to the source definition order.

For example of metaprogramming usage, we demonstrate the ordinary power generator function. Our purpose is to accelerate the power functionality by calculating at preprocess time the multiplication of the base variable.

```
.& {
  var gen_power = function( baseAst, exponent ) {
    var resAst = .< .~baseAst; >.;
    for(var i=0; i<exponent; ++i) {
      resAst = .<
        .~resAst * .~baseAst;
      >.;
    }
    return resAst;
  };
};

x = Math.random();
y = Math.random();
var rand = .!gen_power( .< x + y; >., 10 );
console.log( rand );
```

After the evaluation of the staged process, the final source will be exported as:

```
x = Math.random();
y = Math.random();
var rand = (x + y) * (x + y) * (x + y) * (x + y)
           * (x + y) * (x + y) * (x + y) * (x + y)
           * (x + y) * (x + y) * (x + y);
console.log(rand);
```

3.2. Staging Process

Stagedmonkey operates as a preprocessor tool. It takes as input pure JavaScript, enriched with meta-JavaScript and exports pure JavaScript. In this pipeline process, many steps are formed to achieve the result. Functionalities like parse and evaluation of JavaScript source code are certainly required in the entire process. As we mentioned, we are extending the Spidermonkey JavaScript engine in our preprocessor core. For our purposes, we modify and extend many part of the engine. More specific, we can separate the preprocess procedure in some logical parts, as numbering below and illustrating in Figure 4.

1. Receive as input the meta-JavaScript source file.

2. Call Reflect.parse() JavaScript library function setting the source text as argument and yielding a JavaScript AST object. This object essentially represents the AST of the corresponding source file.

3. Iterate the AST object, identifying the innermost stage nesting level.

4. Assemble the integrated metaprogram for the specific nesting level, respecting the appearing order in the source file. Convert Quazi-quotes definitions to JavaScript AST object incorporating the escape values. Internally mark the inline nodes in the AST.

5. Execute the integrated program. Due to the execution, inline functions will substitute the marked inline nodes with the result of their argument, and erase themselves.

6. Repeat the process, starting from step 3, until no staging level occur.

7. After the previous steps, AST contains only pure JavaScript nodes. The finally step, unparses the AST to produce the final source file.

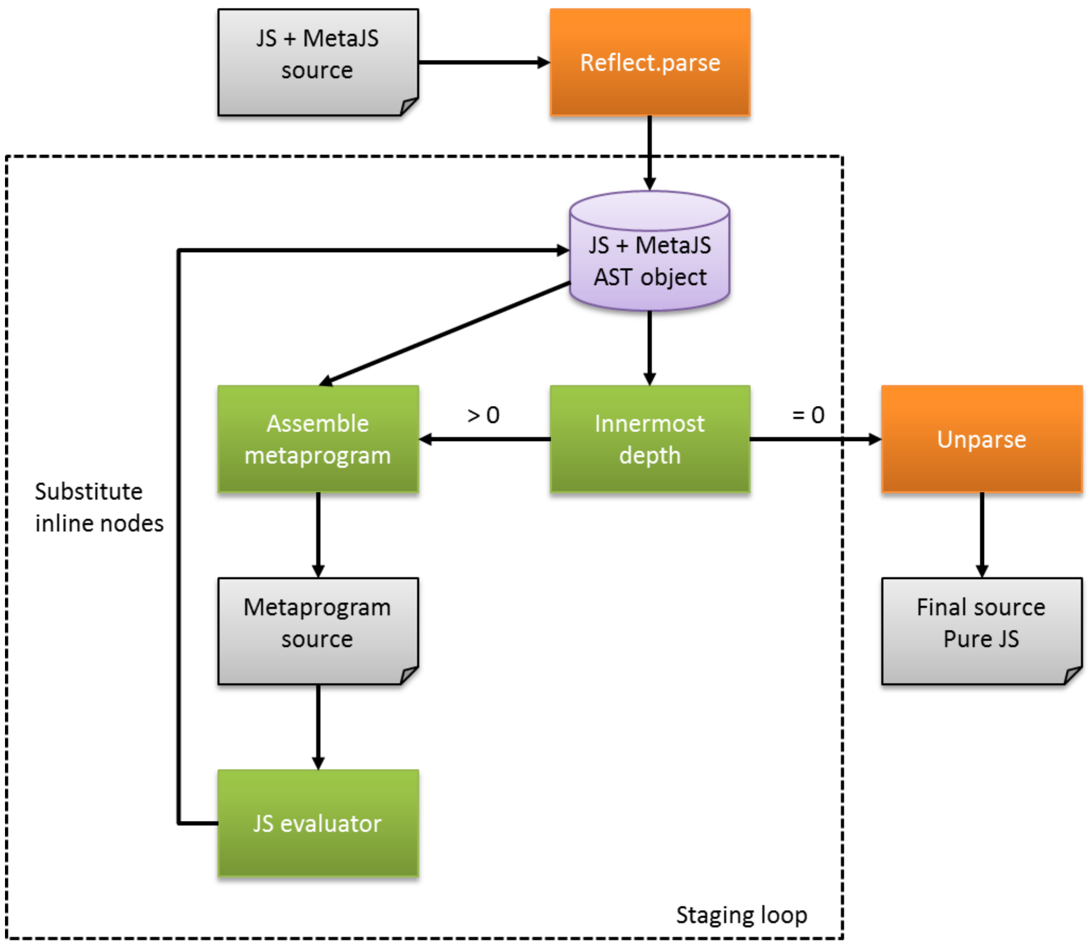


Figure 4 - Diagram of Multi-Staged Metaprogramming evaluation we implement in SpiderMonkey

This loop essentially instantiates the overall staging process. It is invoked exactly after parsing, taking as input the program AST, which includes the new tags. After the staging evaluation it results a modified AST, with no staging-related tags. In the overall translation pipeline, it lies between parsing and code generation, although internally involving code generation and execution rounds of the main JavaScript engine (extracted stages are always pure JavaScript programs). The extracted stage is a coherence JavaScript program, which is unparsed and evaluated with a new JavaScript engine instance. The unparsed source is saved to disc for convenience, enabling programmers view the entire stage outside the enclosing program.

Initially, we aimed to adopt the internal AST structures of SpiderMonkey (ParseTreeNode in C++) and use it throughout the staging process as the input/output data type for ASTs. Then, we observed that using the original AST structures entailed a few issues, including many missing features for tree editing and composition, since ParseTreeNode was not designed to be mutable during the translation process. Then, we noticed that SpiderMonkey defines the format of JavaScript objects that can be considered as well-formed AST values, offering methods of its reflection library to convert between ParseTreeNode* and JSObject, the latter being SpiderMonkey internal representation of JavaScript objects. Thus, we decided to use this facility, and convert the initial AST to JSObject*, hence work exclusively on JSObject* values as the staging AST representation type.

Based on this, the detailed logic of the staging loop, that we implemented for SpiderMonkey is provided under Figure 4. As shown, when extracting the nodes comprising the current stage, we also keep a list of node references, corresponding to the inline directives of the stage with their order in the source. Such references are the actual positions of inline tags in the main AST, not the stage one. This list helps in evaluating inline directives, in particular for replacing their presence in the main AST by their argument, as is discussed later in detail.

3.3. Spidermonkey: Modifications and Additions

3.2.1. Staged Syntax

3.2.1.1. Lexical

In lexical phase, SpiderMonkey is processing sequentially the source characters and transforming them to tokens. SpiderMonkey adopts the entire functionality of ECMAScript5 extended with some custom syntax that is solely supported and represents the metaprogramming semantics. Staging tags must distinguish from the rest JavaScript tokens, mainly for two reasons. Apparently, staging tags must have no conflict with the current reserved JavaScript tags, and potentially future reserved tags. Secondly, staging phase will be evaluated preceding the main JavaScript program, so the syntax must be distinguish to the eye. All the staging tokens spectrum are prefixed or postfixed by the dot "." punctuation mark. Consequently, the staging tokens are the following:

- Quazi-quotes" .< ", " >. "
- Escape " .~ "
- Escape-value " .@ "
- Execute " .& "
- Inline " .~ "

3.2.1.2. Grammar

The implementation of SpiderMonkey parser is custom. Hence, tools for automatic grammar generation is missing. The grammar extensions inserted hardcoded, following the internal SpiderMonkey implementation pattern practices from similar grammatical rules. In JavaScript the entire rules are derivates from either a statement or expression. Thus, our staging extensions to the language behaves respectively.

Extension Rules

We enrich the grammatical rules for supporting the staged grammar.

unaryExpression -> .~ expression
unaryExpression -> .! expression
primaryExpression -> .& statement
unaryExpression -> .@ expression
primaryExpression -> .< statements >.

We will define the grammar of staging tags.

- Primary-Expression -> Quazi-quotes (.< statements >.) any kind of JavaScript valid code can interleaved with them. Quazi-quotes can be posed at anyplace that a primary-expression can be written.

- Unary-Expression -> Escape (.~expression), Escape-value (.@expression), Inline (.!expression) for grammar simplicity those rules are chosen to be placed as unary-expression. This decision diminishes some expressiveness regarding to the places that they can be posed, but in contrary, it simplifies the grammar complexity.

- Primary-Expression -> Execute(.&statement)

AST Additions

As an incidence of the grammar modification, new AST node types inserted to the current AST structure. Thus, each of the precede rule had a respectively addition to the SpiderMonkey AST definition. The original AST structures have been extended to accommodate the new required node types. In SpiderMonkey, this just meant the insertion of extra values for the *ParseNodeKind* enumerated type (*PNK* also used as a synonym). The original node creation methods were sufficient since all new tags are unary operators with a single AST operand, something also apparent in the following creation expressions (now part of the parser code).

```
new_<UnaryNode>(PNK_METAQUASI,JSOP_NOP,pos,defs)
unaryOpExpr(PNK_METAESC,JSOP_NOP,expr)
new_<UnaryNode>(PNK_METAEXEC,JSOP_NOP,pos,stmt)
unaryOpExpr(PNK_METAINLINE,JSOP_NOP,expr)
```

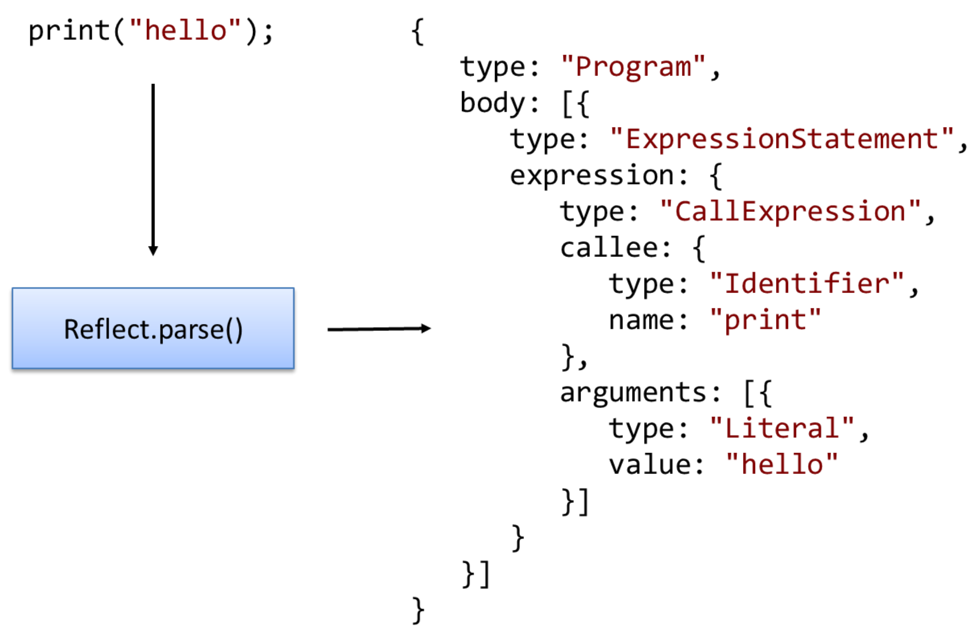
The only modification we had to accomplish in ParseNode, concerned the addition of an extra union field when using escaped or inlined function names, as shown below.

```
class ParseNode { …
  union { …
    struct { …
      ParseNode* escOrInlineFuncName;
    } name;
  } pn_u;
};
```

Finally, a few AST structural assertions had to be loosened by accommodating the presence of staging tags. For example, we support escape and inline expressions to be part of a function name or argument. This type of syntax is not supported by the hosting language.

3.2.2. Parse Source Text

In programming languages, parsing source text is defined as the procedure of transforming a sequence of characters in a target structure. Our purpose is to transform JavaScript source text to a JavaScript AST object. AST object is a valid JavaScript object that essentially encompasses the definition of a JavaScript source code in an AST form. For example the corresponding AST of a *print('hello')* is illustrated in Figure 5.



```
print("hello");
```

```
{
   type: "Program",
   body: [{
      type: "ExpressionStatement",
      expression: {
         type: "CallExpression",
         callee: {
            type: "Identifier",
            name: "print"
         },
         arguments: [{
            type: "Literal",
            value: "hello"
         }]
      }
   }]
}
```

Figure 5 - Reflect.parse() functionality

SpiderMonkey has already incarnate the parse logic, by reflecting the internal parse functionality, using the Reflect.parse [16] extension. A rich ecosystem of tools has formed around this particular structured representation of JavaScript programs, a notably example is the popular Esprima parser. The reusability and composability of these tools has define this format as the unofficial standard for all modern projects that manipulate the structure of JavaScript source code. Such applications regarding with source code editors, IDEs, and convertors.

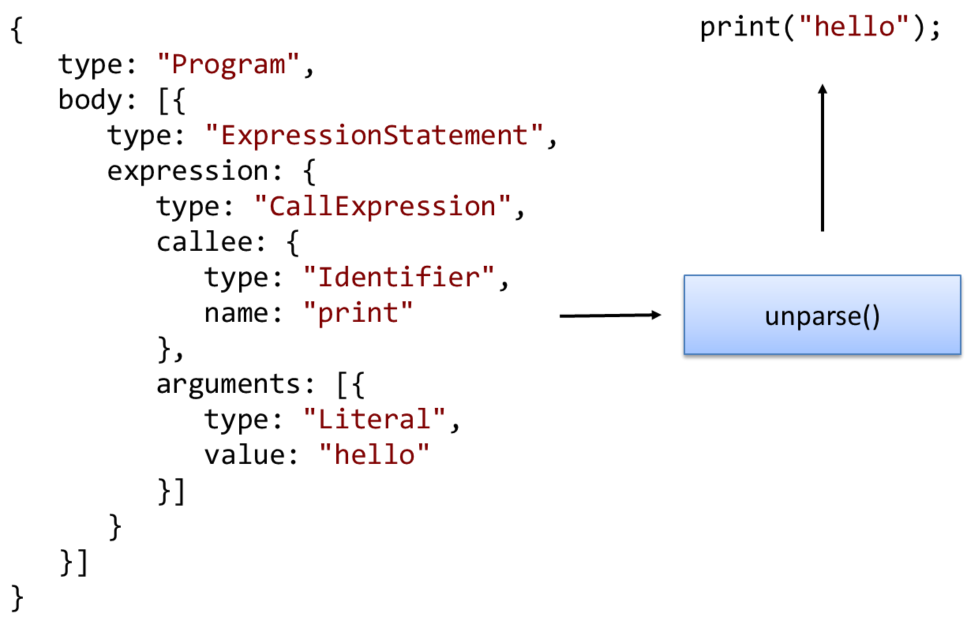
Our interference in the Reflection.parse was the definition of the staging tags. Firstly, in the representation of AST object, and secondly in the core parsing logic.

3.2.3. Unparse AST

We call “*unparse AST*” the procedure of transforming an AST form to a precise source text representation, as depicted in Figure 6. Unparse AST functionality is completely missing from the SpiderMonkey assets, hence we implement it from scratch. In general, we argue the equality of the following formula:

Unparse(Parse(Source-Text)) == Source-Text.

This is not precisely equal in JavaScript. Intuitively, this peculiarity seems to generate enormous inaccuracies. In Contrary, this disparity has no effect in the prior equality formula because a JavaScript source expression can be written in several ways, essentially comprehending equivalent. For instance, a string definition in JavaScript can be defined using either (‘’) or (“”). During the parsing procedure, when a string is recognized, an AST node is generated containing the string value. Henceforth, there is no way to separate the string punctuation definition origin. Thus, in the unparsing procedure, each string AST node will be interpreted using (“”) punctuation delimiters. Therefore, this bidirectionality will lead to the same coherent program. Conclusively, we argue that even if the equality in not strictly the same, it essentially results to the same execution.



```
{
   type: "Program",
   body: [{
      type: "ExpressionStatement",
      expression: {
         type: "CallExpression",
         callee: {
            type: "Identifier",
            name: "print"
         },
         arguments: [{
            type: "Literal",
            value: "hello"
         }]
      }
   }]
}
```

```
print("hello");
```

Figure 6 - unparse functionality

Due to the unparsed procedure, some AST nodes had special treatment. Specifically, as we previously mentioned, the AST manipulation tags are syntactic sugar AST facilitators. Considering this, we unparsed those nodes conceptually.

Quazi-quotes is essentially the syntactic sugar for convening a source text representation of a JavaScript definition to its AST form. For our purposes, we are using the SpiderMonkey AST definition. Thus, a JavaScript definition around Quazi-quotes is strictly equivalent with a JavaScript AST object that describes the same definition.

Consequently, a JavaScript AST object is yielded form a source text definition in the unparsed procedure of a Quazi-quote node.

Escape, as we mentioned, is preventing an expression inside a Quazi-quote definition to be parsed as simple AST. We handle that by replacing the escape node with a function that takes the Escape definition as argument and concatenating the corresponding AST with the Quazi-quote parent AST, as shown in Figure 7.

Escape-value is acting precisely as Escape, with the difference that Escape-value argument is pure JavaScript first class values (boolean, number, string, et.). For example, assume the expression *.<print(.@x, 4)>.*, where *x* is assumed previously defined and contains the string "hello". This expression evaluates *to print("hello", 4)*. Therefore, during unparsed procedure, the Escape-value node substituted with the corresponding runtime JavaScript function *escape_value*.

A comprehensive example indicating the translation of Quazi-quote and escape depicts in Figure 7 .

```
.& {
  function Ctor (name, args, stmts) {
    return .<
      function .~(name)(.~args) {
        .~stmts;
      }
    >.;
  }
}
a = .< x, y >.;
s = .< this.x = x; this.y = y; >.;
point2dCtor = Ctor (.< Point2d >., a, s);
point3dCtor = Ctor (
  .< Point3d >.,
  .< .~a, z >.,
  .< .~s; this.z = z; >.
);
```

```
function Ctor (name, args, stmts) {
  return {
    type: "Program",
    body: [{
      type: "FunctionDeclaration",
      id: meta_escape(
          "SINGLE_ELEM",
          name,
          "IS_EXPR"
        ),
      params: meta_escape(
          "LIST_ELEM",
          [],
          [{index: 0, expr: args}],
          "IS_EXPR"
        ),
    body : [{
      type: "BlockStatement",
      body: meta_escape(
          "LIST_ELEM",
          [],
          [{index:0, expr: stmts}],
          "IS_STMT"
        )
      }]
    }]
  };
}
```

Figure 7 - JavaScript unparse of Ctor function

3.2.4. Runtime JavaScript Library Functions

As mentioned earlier, stages become coherence JavaScript programs where the staged tags is converted to the invocation of a respective library function, installed upon start-up on the extended SpiderMonkey engine. For the requirements of metaprogramming, some JavaScript Global functions inserted.

Unparse function takes as argument a JavaScript AST object and return the source text respectively. Essentially, unparse function is the reflection of the internal Unparse AST procedure. Unparse will execute any occurrence of staged code, until pure JavaScript code remains. Consequence, unparse is the gemstone of the JavaScript metaprogramming edifice.

Inline function takes as argument a JavaScript AST object and substituting itself with the enclosing program. A Runtime Exception will be thrown in cases that Inline argument is different than JavaScript AST object. Obviously, the inline node position in the program AST is internally marked during the staging phase. Inline is not appropriated for custom usage.

Escape function takes as argument a JavaScript AST object and concatenating it to the enclosed JavaScript AST object. To archive this functionality arguments are parameterized depending on each of the following enumerated case. More specific, the escaped AST with the enclosed AST has four possible concatenation cases. Each of the concatenation appearances modifies the Escape function arguments. The escape signature is *meta_escape(true, {normal ast nodes, and postition}, {escape ast nodes, and postition}, isFromStmt)* Figure 8 or *meta_escape(false, {escape node}, isFromStmt)* Figure 9.

- Multiple statements,
  source code example: *if(1){ print(1); .!x; t=2; },*
  formation: *meta_escape(true, [{node:print(1), index:0}, {node: t=2;, index:2}], {node:x, index:1}, true)*

- Multiple Expressions,
  source code example: *print(1, .!x, "world"),*

formation: *meta_escape(true, [{node:print(1), index:0}, {node: t=2;, index:2}], {node:x, index:1}, false)*

- Single Statement,

  source code example: *if(1).!x*,

  formation: *meta_escape(false, x, true)*

- Single expression,

  source code example: *return .!x*,

  formation: *meta_escape(false, x, false)*

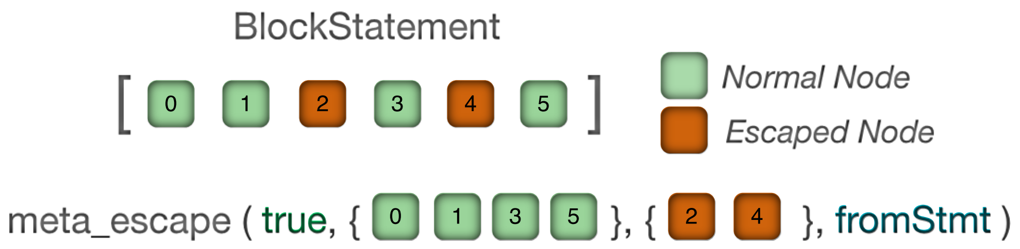


Figure 8 - Addressing escape treating as list of expressions or statements

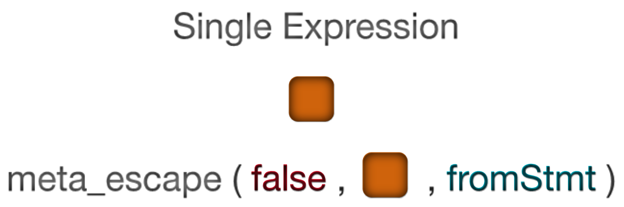
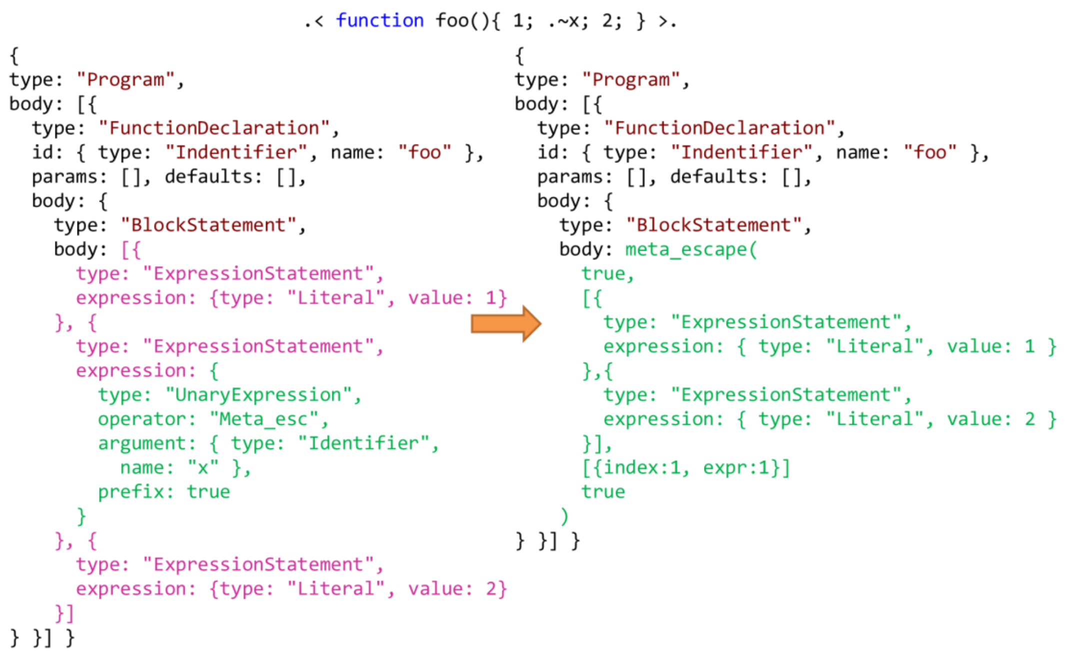


Figure 9 - addressing escape treating as a single expression or statement

We depict an *escape* excerpt from an AST segment in Figure 10 that clarifies the transformation.



```
.< function foo(){ 1; .~x; 2; } >.
```

```
{
type: "Program",
body: [{
  type: "FunctionDeclaration",
  id: { type: "Indentifier", name: "foo" },
  params: [], defaults: [],
  body: {
    type: "BlockStatement",
    body: [{
      type: "ExpressionStatement",
      expression: {type: "Literal", value: 1}
    }, {
      type: "ExpressionStatement",
      expression: {
        type: "UnaryExpression",
        operator: "Meta_esc",
        argument: { type: "Identifier",
          name: "x" },
        prefix: true
      }
    }, {
      type: "ExpressionStatement",
      expression: {type: "Literal", value: 2}
    }]
} }] }
```

```
{
type: "Program",
body: [{
  type: "FunctionDeclaration",
  id: { type: "Indentifier", name: "foo" },
  params: [], defaults: [],
  body: {
    type: "BlockStatement",
    body: meta_escape(
      true,
      [{
        type: "ExpressionStatement",
        expression: { type: "Literal", value: 1 }
      },{
        type: "ExpressionStatement",
        expression: { type: "Literal", value: 2 }
      }],
      [{index:1, expr:1}]
      true
    )
} }] }
```

Figure 10 - unparse to AST translation of a Quazi-quote snippet with an enclosed escape expression

escape_value function takes as argument a pure JavaScript first class value, creates and return an AST node, containing the specific value. A Runtime Exception will be thrown in cases that Escape-value argument is different than JavaScript first class value. This is very useful in cases that we need to append simple values to an AST. For example, assume the following code

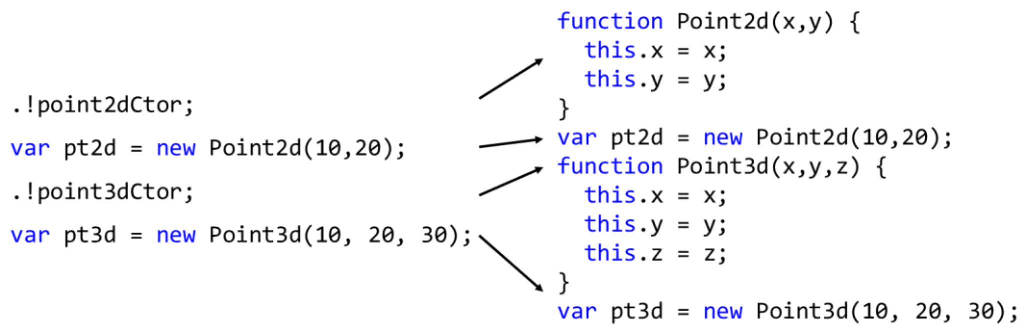
```
function genLargeElement( size ) {
  return .<
    div = document.createElement('div');
    div.style.width( .@size + 'px' );
  >.;
}
```

We set the size of the generated div element according to the *size* variable, where *size* is a pure JavaScript number.

Extending the previous stage example, where we composed two ASTs carrying the *point2dCtor* and point3dCtor variables, we add the following extra four lines:

```
.!point2dCtor; (*staged code, 1st inline in I*)
var pt2d = new Point2d(10,20);
.!point3dCtor; (*staged code, 2nd inline in I*)
var pt3d = new Point3d(10, 20, 30);
```

In this case, the two inline tags are concatenated exactly after the code block of the execute tag (.&) of our earlier example, making the first stage to evaluate, being also the only stage in this case. The stage assembly logic has been outlined earlier, in the staging loop under Figure 4. As discussed, it collects all statements at the same stage nesting, in the order they are met, and inserts inline tags belonging to the stage. Thus, for example, the inline list contains the node references for the expressions *.!point2dCtor* and *.!point3dCtor* from the AST of the main program. The evaluation of the two inline tags causes a rewriting of the main AST by replacing the inline directives with the content of *point2dCtor* and *point3dCtor*. It results the following final code of the main program, after staging is performed, Figure 11.



```
.!point2dCtor;
var pt2d = new Point2d(10,20);
.!point3dCtor;
var pt3d = new Point3d(10, 20, 30);
```

```
function Point2d(x,y) {
  this.x = x;
  this.y = y;
}
var pt2d = new Point2d(10,20);
function Point3d(x,y,z) {
  this.x = x;
  this.y = y;
  this.z = z;
}
var pt3d = new Point3d(10, 20, 30);
```

Figure 11 - Inline transformation

Chapter 4

Staged Errors and Debugging

The proper tools for meaningful error reporting and right debugging facilities are foundation concepts of an integrated development environment. A development environment that lack of essential inspection tools for its programming functionalities, is a flavorless medium that wastes the user’s time. In the context of metaprogramming, adapted to the staged needs, we integrate a vital error reporting functionality and an essential debugging mechanism.

4.1. Error Report

Through the staged process, during the staged depths, the AST creation and composition, several possible errors may occur. Ideally, we would like to have an accurate report of the error, embodying the full trajectory with a precise concatenation trace. To accomplish this prescription we reuse the source location information of the AST node, produced by the *Reflect.parse()* routine. Those source information encompass the starting and the ending position of the AST node. During the concatenation process, produced by escape, the concatenated string remains unharmed. When an error occur regarding to the AST manipulation, programmer can inspect the origin of the corresponding AST composition parts. The following source code example indicates the location information attached in the AST form of `var AstOfNumberTwo = .< 2; >.;`.

```
var AstOfNumberTwo = {
  loc: { start: { line: 1, column: 20 },
         end: { line: 1, column: 22 }, source: null
  }, type: "Program", body: [{
    loc:{start:{line:1, column:20},
          end:{line:1, column:22}, source:null
    }, type: "ExpressionStatement",
    expression: {
      loc: { start: { line: 1, column: 20 },
             end: { line: 1, column: 21 }, source: null
      }, type: "Literal",
      value: 2
    }
  }]
};
```

Additionally, we wisely insert to the staged JavaScript library function, many runtime sanitize checks. Those checks are trying to clarify the soundness of a wide variety of exploits and errors. Inline, Escape and Escape-value are checking the validity of their arguments. After each AST composition, we check the hygienic of the outcoming AST. In case of the occurrence of a specified error, a runtime exception will be thrown.

4.2. Debugging

Each program that assembled and evaluated during the staged process is an isolated standalone coherence JavaScript program. Consequently, metaprograms treated as pure JavaScript programs. Hence, we export the internal staging procedure, aiming to evaluate each stage to any external JavaScript engine. This kind of reflection cause minor modifications to the current staged integration. This notion provides us the luxury to use the current JavaScript debugging infrastructure. Many sophisticated JavaScript debuggers are involving due to the raise of applications that using JavaScript, some current examples are the Chrome developer tools 82[17] and

Firefox inspection tools [18]. Thus, programmer is free to choose his preferred JavaScript debugger.

4.2.1. Debugging Model

Staged debugging is separated in two isolated architectural modules that communicate each other using http requests. Backend is the component that initialize and manage the debugging flow. Specifically,

- Read the file that programmer want to preprocess.
- Assemble the next staging program and mark the inline points.
- Initiate an http server to serve the corresponding staging file.

The frontend component contains two fundamental processes. The first process is the debugging evaluator. It is a web page that communicate with the backend via Ajax Json requests. Precisely,

- Receive the next staged source code from the backend.
- During the execution, when an inline directive reached, it notify the server to execute the corresponding inline, assigning the AST object value that evaluator currently contains.
- Provides, to the client programmer, the ability to inspect and visualize any AST object. To achieve this, the *STG_inspectAst(astObject)* function injected as a global library function. This function essentially send an AST inspection request to the backend.

The second frontend process is the debugging visualizer. It essentially depict to programmer useful information about the debugging process. The synchronization with the current debugging state is achieved by periodical Ajax communication with the Backend. Precisely,

- Shows the current staging depth.

- Show the remaining and the executed quantity of Inlines in the current stage.
- Each inspection that programmer trigger by the *STG_inspectAst*(*astObject*), is rendered as an enriched interactive tree view.

We summarize the debugging ecosystem in Figure 12.

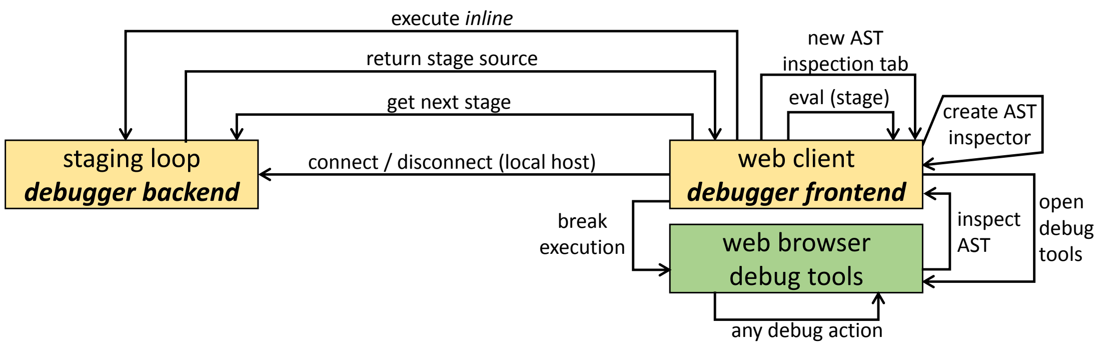


Figure 12 - Stage debugging architecture by splitting responsibilities between staging loop for stage extraction, the web browser tools for typical debugging activities, and a custom web-client for stage evaluation and AST inspection.

4.2.2. Backend

To support debugging for stages we have implemented a backend service loop as part of the staging loop, which basically responds in two requests (besides the apparent connect/disconnect ones): (i) extract and return the next stage; and (ii) apply an inline directive. The escape directive affects locally the ASTs composed in a stage and was implemented as a JavaScript function merged with the produced stage source. With this we avoided an extra message, something though not possible with inlines that affect the main AST that is kept in the running SpiderMonkey instance.

The backend process flow is the equivalent with the normal staged flow until the execution of the corresponding stage. At that point, debugger backend initialize an http server that intended to remote debug the staged execution. SpiderMonkey does not contain any coherence http library. Thus, we include a simple comprehensive web server for handling http requests (LibMongoose [19]). LibMongoose is an open source lightweight crossplatform C/C++ http and Websocket library. We create an abstract layer over the LibMongoose library for more flexible usage, adapted to our requirements. After that addition, the installation of http routes and handlers is very simple and straightforward. For example,

```
httpHandler.installRoute(
  httpRequestHandlerInfo(
    "/execinline", "application/json", "POST", &execInline
  )
);
```

can be readen as: "When we receive a post request that its content type is *application/json*, trigger the server at */execinline* route, call the *execInline* function as the request handler". Seemingly, this http abstraction can be exploited in a variety of applications, irrespective of our specific requirements. Using the specific pattern, for the needs of the debugging process, we install several routes to the backend. All the routes accepts only POST methods, with application/json content type requests. The response of the routes are always in Json format and wrapped to a response template message:

```
{ "msg": responseContent }
```

Due to the flexibility of the http protocol, any custom debugger client can be integrated, in any programing language and environment. Essentially, the only implement obligation that the client debugging application requires, is the communication with the backend, implementing the following methods.

Method: `/nextstage`
Parameters: Ø
Response:
```
{ "depth": currentStagedDepth, "srcCode": srcCode, "inlines": totalInlines }
```
Or
```
{ "depth": 0 }
```

This is the route that client debugger call in order to get the source code of the current staged execution. If no stage remaining, 0 will be given.

Method: `/execinline`
Parameters: the AST JavaScript object, to be assigned as the argument of the inline execution, serialized to Json, in the body of request.
Response: `{ "inlines": remainingInlines }`

Inform the server to execute the next inline of the current stage, assigning the corresponding argument.

Method: `/inspectast`
Parameters: the AST JavaScript object, to be inspected serialized to Json, in the body of request.
Response: `true`

Inform the backend that the specific AST object must be inspected.

Method: `/closesession`
Parameters: Ø
Response: `closed`

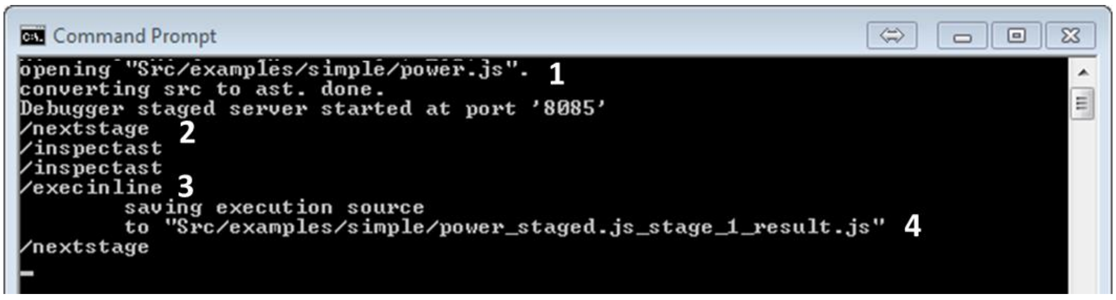
Notify the backend to finish the current debugging session.

Method: `/syncdbg`
Parameters: Ø
Response:

```
{
  "inlines": remainingInlines,
  "stage": {
    "depth": currentStagedDepth,
    "srcCode": stagedSrcCode
  },
  "inspectAst": ASTJavascriptObject
}
```

This method is the main informer about the current staged debugging process. Containing useful notions related with the remaining Inlines, the current stage depth, the stage source code and if there is any AST object to be inspected.

We depict, in Figure 13, an example of the runtime terminal created by the backend debugger program. Firstly, (1) it is opening the file that intended to be debugged, initializing the http server and notifies that it is waiting for a client debugger attaching in a specific user-defined port. During the debugging process (2), backend debugger verbose programmer about the routes that debugging client has reached. In our instance, client has initially request the next JavaScript stage, apparently during the stage debugging, client has inspected two ASTs. After the inspections an inline expression (3) has executed. Finally, the staging evaluation at the specific depth has finished, subsequently, the backend debugger saves (4) the regarding source code in a JavaScript file. Debugger client requests the next stage, therefore, the debugging loop is repeated.



```
opening "Src/examples/simple/power.js".  1
converting src to ast. done.
Debugger staged server started at port '8085'
/nextstage  2
/inspectast
/inspectast
/execinline 3
        saving execution source
        to "Src/examples/simple/power_staged.js_stage_1_result.js"  4
/nextstage
_
```

Figure 13 – backend debugger terminal on action

4.2.3. Frontend

Frontend debugger is an independent architectural component, implemented as a web page. The debugger is separated in 2 views and communicate with the server using Ajax requests. The first view (Evaluator) is following the main debugging flow, exploiting the browser's debugging tools. The second view (Visualizer) is displaying all the necessary debugging information regarding with the process.

4.2.3.1. Evaluator

Due to the prosperity of JavaScript in a variety of applications, as we previously mentioned, many sophisticated debuggers are integrating. We grab this advantage by exporting the staged execution process to the frontend. This practice had many challenges. Before we came up with the challenges, we will briefly descript the frontend evaluation flow. Firstly, we prompt the user to open the browser's inspect element tools, as shown in Figure 14. Hereafter, we get the current staged source code from the server and evaluate in the browser context using the *eval* JavaScript function. While the eval is executing, user is able to inspect the stages source, trace the execution flow, add breakpoints and watchers, and obtain the benefits from the JavaScript debugging tools that the corresponding browser is offering.

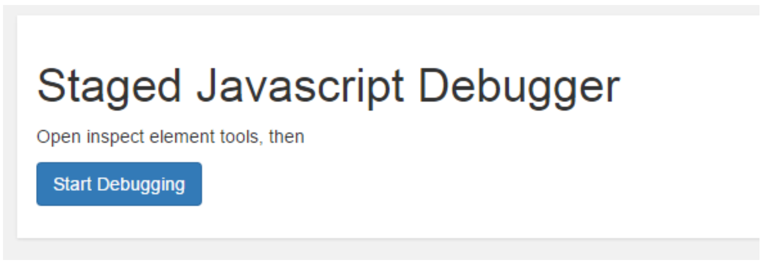


Figure 14 - user prompt to open the inspect element tools

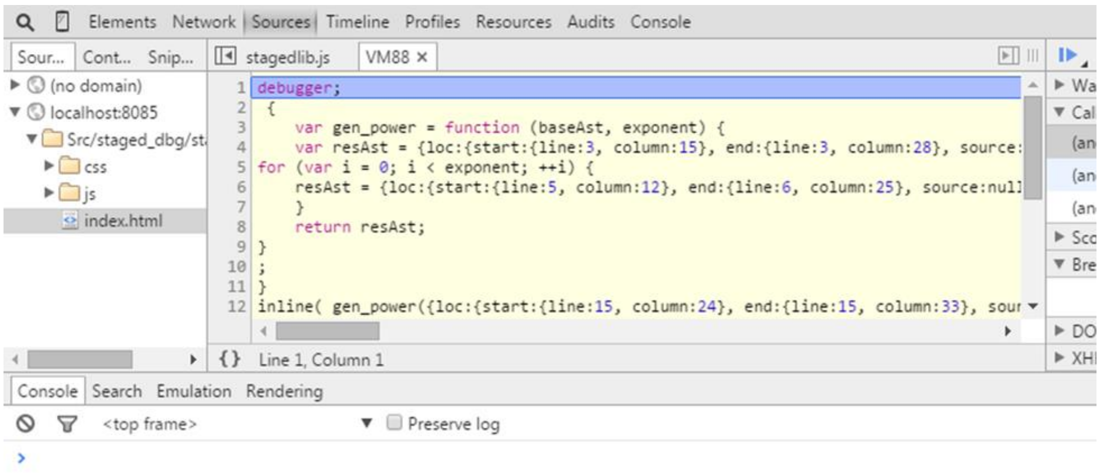
```
function startDebugging() {

  var onNextStageStart = function(msg) {
    if(msg.stage===0) // was the last stage
      openPage('Finished stage debugging');
    else {
      var src = unescape(msg.src);
      openPage('Debugging stage', msg.stage);
      setOnLoadedPage(function(){
        eval(src); // execute current stage
        startDebugging(); // debug next one
      });
    }
  };

  stageDebugSend({ //backend communication
    route:'NextStage',
    success: onNextStageStart,
    fail: onNextStageError
  });

  openPage('Start stage debugging', startDebugging);
}
```

As shown, the client loop is not actually implemented as an explicit loop, but repeats stage debug sessions as follows: (i) a message is sent to the staging loop in the backend requesting to extract the next stage, also it is setting a local response handler function *onNextStageStart*; (ii) on receipt of the response, the handler function checks if there are no more stages *(msg.stage===0)*, else it removes character escape sequences from the stage source and directly evaluates it by restarting a new debug round. In order to support stage tracing with the browser debugging tools, the staging loop inserts a debugger; statement directly at the first line of every stage source posted to the frontend client, as illustrated in Figure 15. The latter causes an instant breakpoint when *eval* is invoked, stopping execution with the debugger tools at the first line, before the stage is essentially executed. At this point, additional breakpoints may be interactively inserted by the programmer as desired, using the debug tools. An instance of the debugging execution is illustrating in Figure 16.



```
debugger;
 {
    var gen_power = function (baseAst, exponent) {
    var resAst = {loc:{start:{line:3, column:15}, end:{line:3, column:28}, source:
for (var i = 0; i < exponent; ++i) {
    resAst = {loc:{start:{line:5, column:12}, end:{line:6, column:25}, source:null
    }
    return resAst;
}
;
}
inline( gen_power({loc:{start:{line:15, column:24}, end:{line:15, column:33}, sour
```

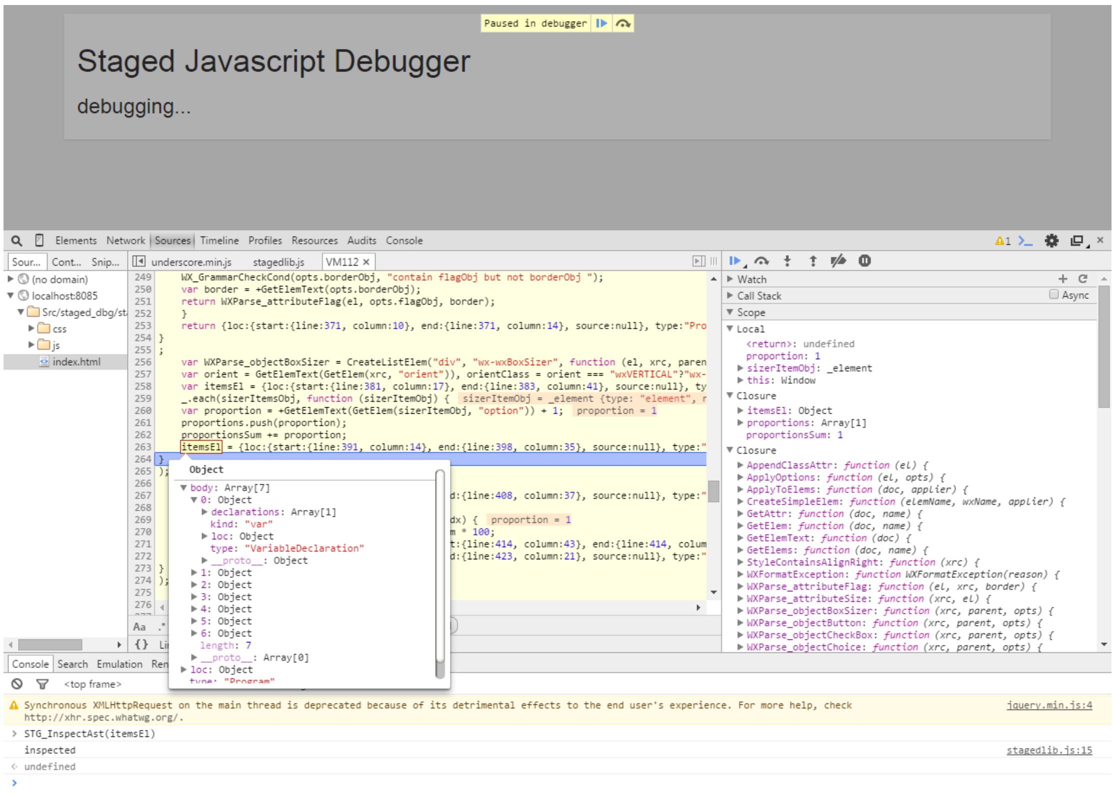
Figure 15 - Debugger evaluator, a 'debugger' statement inserted at the first line to trigger a breakpoint

An issue that came up during the integration of this procedure is related with the way that we could stop the execution before the staged source evaluation. The problem occurs when we *eval* the staged source code. The execution of the source is instantly, so there is no inspection room for the user. We resolve this issue by hardcoded inserting a debugger statement *debugger;* in the first line of the stage source code. When the execution flow reached the first line, and programmer has open the inspect element tools, debugger statement behaves as a breakpoint. Hence, the execution stops there and programmer is free to inspect the source.

The debugging evaluation of the staged source is taking place in the frontend. Frontend execution environment is different than Spidermonkey context. As a result, all the global functions of SpiderMonkey along with our additional functions, are totally missing from the JavaScript global object. Consequently, we had to replicate all the possible functionalities.

We change the functionality of inline in frontend by sending a synchronized Ajax request to backend. We force it to execute the corresponding inline, assigning the argument of the frontend's inline function. Escape and Escape-value are reproduce in a few lines of pure JavaScript code.

SpiderMonkey global functions offer a variant of functionalities, related to open/write files, evaluate source text, dump execution information, and many other. Obviously, several of these functionalities cannot reproduced in the client execution environment. For example, the SpiderMonkey's function *dumpHeap* dump information about the current heap. Browsers cannot gain access to this type of data, mostly for security reasons. For these functions we throw a *NotSupposedException*. Admittedly, some SpiderMonkey functionalities like open/write files are very useful and commonly used in staged process, as concluded from our study cases. We simulate this functionality by sending a synchronous Ajax request to backend requesting to open/write a file in the corresponding filepath.



```
    WX_GrammarCheckCond(opts.borderObj, "contain flagObj but not borderObj ");
    var border = +GetElemText(opts.borderObj);
    return WXParse_attributeFlag(el, opts.flagObj, border);
    }
    return {loc:{start:{line:371, column:10}, end:{line:371, column:14}, source:null}, type:"Pro
}
;

    var WXParse_objectBoxSizer = CreateListElem("div", "wx-wxBoxSizer", function (el, xrc, paren
    var orient = GetElemText(GetElem(xrc, "orient")), orientClass = orient === "wxVERTICAL"?"wx-
    var itemsEl = {loc:{start:{line:381, column:17}, end:{line:383, column:41}, source:null}, ty
    _.each(sizerItemsObj, function (sizerItemObj) {  sizerItemObj = _element {type: "element", r
    var proportion = +GetElemText(GetElem(sizerItemObj, "option")) + 1;  proportion = 1
    proportions.push(proportion);
    proportionsSum += proportion;
    itemsEl = {loc:{start:{line:391, column:14}, end:{line:398, column:35}, source:null}, type:"
}
);
```

```
Object
▼ body: Array[7]
  ▼ 0: Object
    ▶ declarations: Array[1]
      kind: "var"
    ▶ loc: Object
      type: "VariableDeclaration"
    ▶ __proto__: Object
  ▶ 1: Object
  ▶ 2: Object
  ▶ 3: Object
  ▶ 4: Object
  ▶ 5: Object
  ▶ 6: Object
    length: 7
  ▶ __proto__: Array[0]
▶ loc: Object
  type: "Program"
```

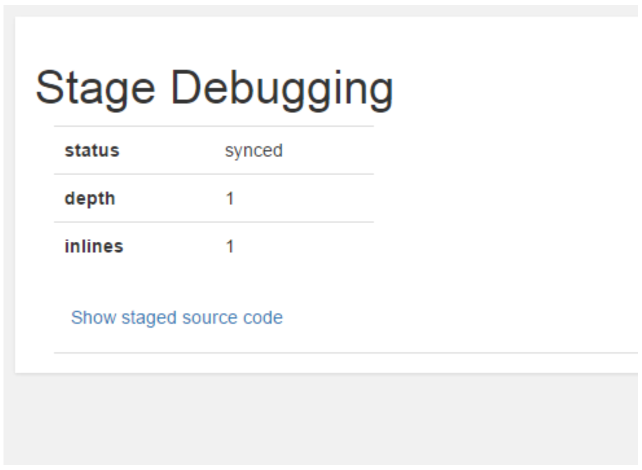
```
▼ Local
    <return>: undefined
    proportion: 1
  ▶ sizerItemObj: _element
  ▶ this: Window
▼ Closure
  ▶ itemsEl: Object
  ▶ proportions: Array[1]
    proportionsSum: 1
▼ Closure
  ▶ AppendClassAttr: function (el) {
  ▶ ApplyOptions: function (el, opts) {
  ▶ ApplyToElems: function (doc, applier) {
  ▶ CreateSimpleElem: function (elemName, wxName, applier) {
  ▶ GetAttr: function (doc, name) {
  ▶ GetElem: function (doc, name) {
  ▶ GetElemText: function (doc) {
  ▶ GetElems: function (doc, name) {
  ▶ StyleContainsAlignRight: function (xrc) {
  ▶ WXFormatException: function WXFormatException(reason) {
  ▶ WXParse_attributeFlag: function (el, xrc, border) {
  ▶ WXParse_attributeSize: function (xrc, el) {
  ▶ WXParse_objectBoxSizer: function (xrc, parent, opts) {
  ▶ WXParse_objectButton: function (xrc, parent, opts) {
  ▶ WXParse_objectCheckBox: function (xrc, parent, opts) {
  ▶ WXParse_objectChoice: function (xrc, parent, opts) {
```

```
⚠ Synchronous XMLHttpRequest on the main thread is deprecated because of its detrimental effects to the end user's experience. For more help, check    jquery.min.js:4
  http://xhr.spec.whatwg.org/.
> STG_InspectAst(itemsEl)
  inspected                                                                                                            stagedlib.js:15
<· undefined
>
```

Figure 16 - Debugger evaluator during the execution

4.2.3.2. Visualizer

Visualizer is the frontend debug component that is responsible for visualizing the debug state. Those informations are valuable to programmer for assisting to the debug process. Visualizer depict the current staged depth, the total and remaining quantity of Inlines, a read-only view of the staged source text and the most important feature, an enriched interactive tree view of the AST JavaScript object. The initial state of debugger visualizer shown in Figure 17.



Stage Debugging

status	synced
depth	1
inlines	1

Figure 17 - Debugger visualizer initial state

Interactive AST Inspection View

The entire staged preprocess is based on AST creation and composition. Thus, a common requirement during the integration of a metaprogram is a helpful AST inspection. Programmer must be able to traverse the AST nodes and inspect the enclosed values in a clear and easy inspection interface. A further step to the AST inspection process should be the ability to instantly display the source text of the corresponding AST node.

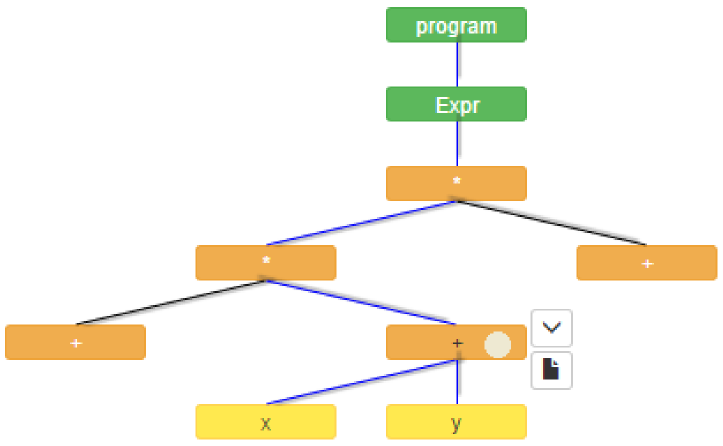
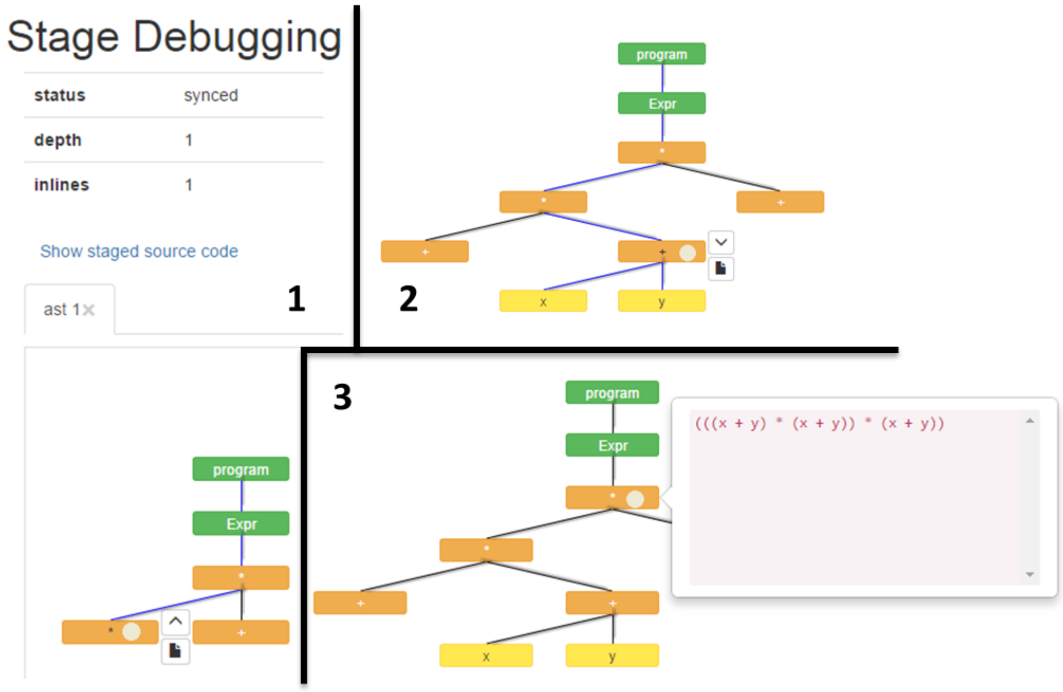


Figure 18 - AST view

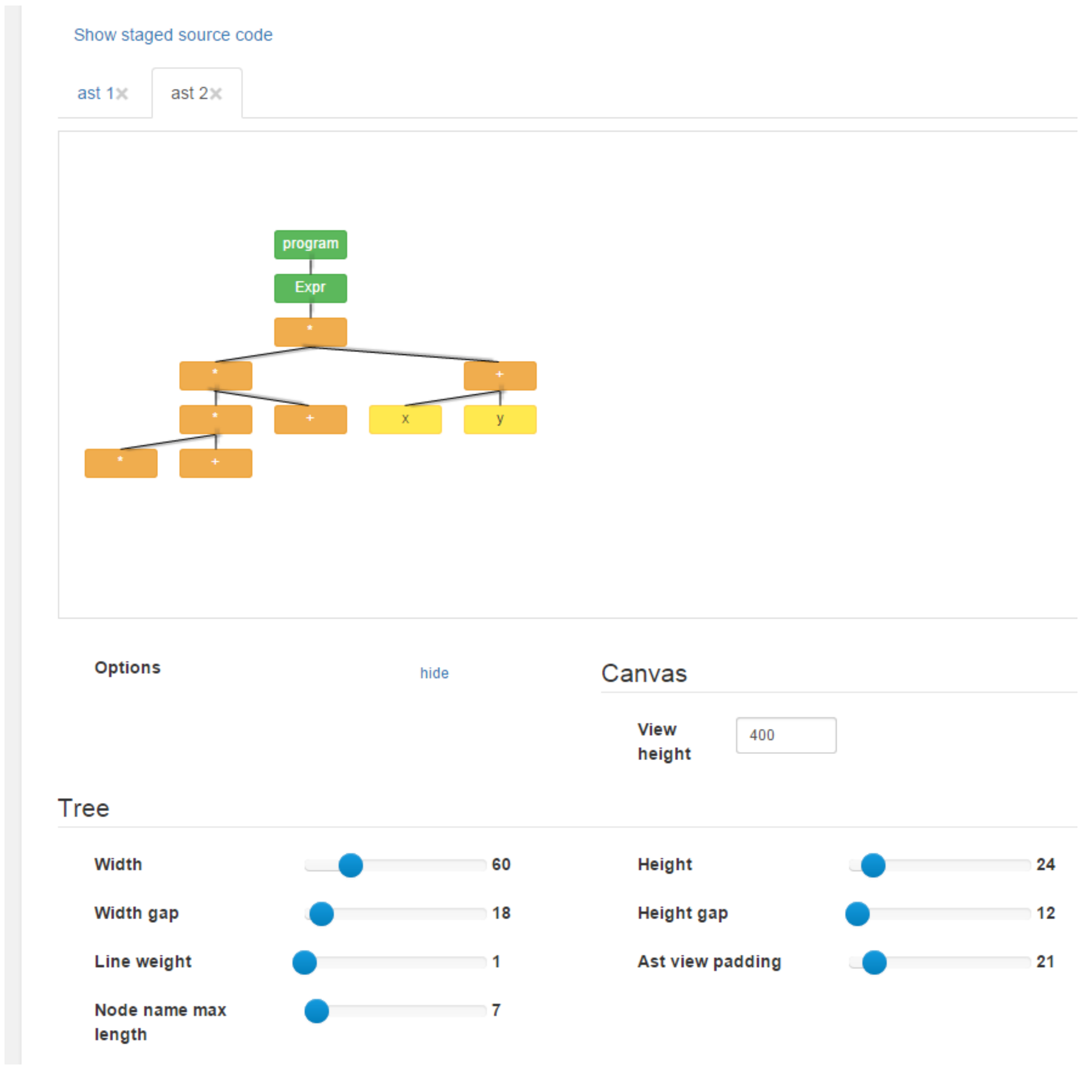
Our interactive AST inspection view, illustrated in Figure 18, is trying to assist the programmer by providing all the essential AST inspection assets. The AST is representing as a tree view. We distinguish the node types by marking them with a different color. Statements are painted green, expressions are orange, literal values are yellow, variable declarations are blue, and lists are white. By painting each node with different color, programmer can instantly separate the inner AST structure. The initial view state of the tree contains only the root node, when user hovers the cursor to the node, the option menu appears, as you can observe on hovering screenshots (1), (2) in Figure 19. There are two options. The first option is the collapse functionality. By pressing it, the parent nodes appear as children of the target node. The specific collapse button is toggled to the reserved functionality. The second button is related to the unparse AST functionality. When user trigger it, a popup window appears containing the corresponding source text, an example of this functionality depicts at (3) in Figure 19. This functionality is very useful in the inspection process, untying the programmer’s hands from trying to interpret each spectrum of AST. When a user hovers the cursor to a collapsed node, a blue line engraved the path from the root to the target node, as shown at (1) in Figure 19.



```
(((x + y) * (x + y)) * (x + y))
```

Figure 19 - Ast visualize instances

We additionally offer a variety of AST inspection view options, shown in Figure 20. User can customize the height of the view, the width and height of each node, the width and height gap of each node, the edge line weight, the maximum length of the node name (after that limit it collapsed and it is visible only by hovering the name) and the view padding. Furthermore, we support multiple AST view tabs, providing the flexibility to explore across the inspected AST views.



Options

Canvas

View height: 400

Tree

Width	60		Height	24
Width gap	18		Height gap	12
Line weight	1		Ast view padding	21
Node name max length	7			

Figure 20 - Debugger visualizer options

Chapter 5

Case Studies and Practices

The core purpose besides metaprogramming is the ability to archive code reusability in a longstanding time scale. As a side effect of this, metaprogramming can be used to bootstrap the JavaScript execution time and generate enormous performance improvements. To prove the JavaScript metaprogramming principles, we:

- Demonstrate improvements on commonly used JavaScript libraries.
- Are taking the metaprogramming advantages by indicating some design pattern generators.
- Exploit the principles of code generation by automating the model-driven process.

5.1. Code Snippets

We represent examples from the real world, refactoring some code snippets, using metaprogramming. Our purpose is to denote that JavaScript metaprogramming can be meaningful used in commonly used libraries.

Underscore.js

Underscore is a popular JavaScript library that provides a whole collection of useful functional programming helpers. While we was inspecting the source, we discover that two ordinary functions was quite similar, with a slight difference in some segments. _*.max* returns the maximum, while _*.min* function returns the minimum element of a collection. For performance reasons these two functions are separated, replicating the essential algorithm logic. We consolidated them using a min-max generator. We figure an excerpt of _*.max* functionality logic.

```
_.max = function(obj, iteratee, context) {
  var result = -Infinity, lastComputed = -Infinity, ...;
  ...
    _.each(obj, function(value, index, list) {
      if (computed > lastComputed
        || computed === -Infinity && result === -Infinity) {
         result = value;
         lastComputed = computed;
      }
    });
  }
  return result;
}
```

We observe that there are two differences between min and max functions.

Max	**Min**
-Infinity	Infinity
computed > lastComputed	computed < lastComputed

Table 2 - Differences between min and max functions

We create a single function that generate min or max function according to a boolean argument. This function preserves the algorithm logic and modifies only the segments that are responsible for the min/max calculation (Table 2). Hence, we have a single concrete function that encloses the algorithm logic. As a result, when we refract or improve this function, the modification will affect both min and max functions.

```
function genMinMax( sign ) {
  if(sign){
    objCompAst = .< computed < lastComputed; >.;
    infCompAst = .< Infinity; >.;
  }else{
    objCompAst = .< computed > lastComputed; >.;
    infCompAst = .< -Infinity; >.;
  }
  return .<
    function(obj, iteratee, context) {
      var result = .~infCompAst, lastComputed = .~infCompAst...;
      ...
        _.each(obj, function(value, index, list) {
          if (.~objCompAst
            || computed === .~infCompAst && result === .~infCompAst) {
              result = value;
              lastComputed = computed;
          }
        });
      }
      return result;
    }
  >.;
}
```

Backbone.js

Backbone.js is a frontend framework for the web, giving structure to the MVC model. Models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling are some of the API conveniences that Backbone.js provides to the programmer. The specific framework has many repetition code fragments, mostly to accelerate the runtime performance. For instance, we excerpt the following function:

```
var triggerEvents = function(events, args) {
  var ev, i = -1, l = events.length, a1 = args[0], a2 = args[1], a3 = args[2];
  switch (args.length) {
    case 0: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx); return;
    case 1: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx, a1); return;
    case 2: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx, a1, a2); return;
    case 3: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx, a1, a2, a3); return;
    default: while (++i < l)
      (ev = events[i]).callback.apply(ev.ctx, args); return;
  }
};
```

This function is an optimized internal dispatch function for triggering events, trying to keep the usual cases speedy. In several JavaScript libraries we meet many functions that verbose some segments to improve the performance. We are aiming to recognize the repetition pattern, in order to generate it automatically. This abstraction gives many advantages to programmer. It helps to devote in the significant code logic, isolating him from the repetition. Additionally, it gives the ability to replicate the pattern as many times as required.

```
function genMultiParameterOptimizer(argumentsLength){
  var optimizedAst = .<>.;
  var casesArgsAst = .< ev.ctx >.;
  for(var i=1; i<=argumentsLength; ++i){
    var caseArgsAst = .< .~casesArgsAst, args[.@i] >.;
    optimizedAst = .< .~optimizedAst;
      case .@i: event.callback.call(.~caseArgsAst); return;
    >.;
  }
  return .<
    (function(event, args) {
      var ev, i = -1;
      switch (args.length) { imparsial
        case 0: event.callback.call(ev.ctx); return;
        .~casesArgsAst;
        default: event.callback.apply(ev.ctx, args); return;
      }
    });
  >.;
}
```

We create a multi parameter generator that replicate the quantity of cases as many times as the function's argument. Hence, programmer can easily append accelerated function calls with unlimited numbers of arguments, just by changing the `genMultiParameterOptimizer` argument.

5.2. Design Pattern Generators

Design patterns are defined as a general reusable solutions to commonly occurring problems within a given context in software design. It is not a finished design that can be transformed directly into source code. It is a recipe for how to solve a specific problem that can be adapted to a variety of applications. Specifically, it is a set of strictly finite rules that helps the programmer to solve and tactically integrate common software design issues. Typically, it appeared in cases that programming language cannot offer you the appropriate tools and facilities. Design patterns accelerate the development process by providing tested and proven development paradigms. It prevents the appearance of problems that will only be visible when it is belatedly during the development process, effecting dramatically the source entropy. Most of the Design patterns are programming language independent, meaning that patterns is a guide that must be adapted in the requirements of the corresponding language.

In our work, we are aiming to create design pattern generators. We are abstracting and shifting to metaprogramming, the models of some commonly used design patterns in JavaScript. Consequently, we detach from the programmer, the tedious routine of reproducing the boilerplate code of the corresponding design pattern. Consequence, programmer will exclusively fill the meaningful content of the design pattern template.

5.2.1. Memoized

Memoized is an optimization technique used primarily to speed up the runtime performance. It achieve that by storing the results of expensive function calls (called content function) and returning the cached result when the same inputs occur again. Obviously, the content function must not affected by the external environment. Thus, the same arguments always resulting the same result.

```
function genMemoized(funDef){
  return .<
    (function() {
      var funcMemoized = function() {
        var result, cacheKey =
          JSON.stringify(Array.prototype.slice.call(arguments));

        if (!funcMemoized.cache[cacheKey]) {
          result = (.~funDef).apply(null, arguments)
          funcMemoized.cache[cacheKey] = result;
        }
        return funcMemoized.cache[cacheKey];
      };
      funcMemoized.cache = {};
      return funcMemoized;
    })();
  >.;
}
```

We create a function that takes as argument the AST of the function that intended to be memoized and yields the AST of the content function, enclosed by the memoized design pattern. Thereafter, we can reuse the memoized pattern by inlining the metaprogramming *genMemoized* function. For example, after the execution of

```
cos = .!memoized(.< function(x){ return Math.cos(x); }; >.);
```

cosine function will remember the previous results.

5.2.2. Singleton

Singleton is one of the most commonly used design patterns. Singleton design pattern guarantees the solely instantiation of a class or object. This is useful when exactly one object is needed to coordinate actions across the system. Singleton assurance that a specific class has only a single instance that is globally accessed.

```
function genSingleton(funDef){
  return .<
    (function() {
      var instance;
      myclass = function() {
        if (instance) {
          return instance;
        }
        instance = this;
        (.~funDef).apply(this, arguments);
      };
      return myclass;
    }());
  >.;
}
```

We create the *genSingleton* function that takes as argument the function AST that we need to retain merely one instance, and yields the singleton wrapped version. Consequently, an individual instance will remain, with a manner that *new* prefix will have no influence.

```
var myclass = .!genSingleton(
    function(name) {
      this.name = name;
      print(name);
    }
  );

var class1 = new myclass('class1');
var class2 = new myclass('class2');
print( class1 === class2 );
```

```
>class1
>class1
```

```
>true
```

5.2.3. Revealing Module

This pattern focuses on public and private methods. The only difference is that the revealing module pattern was engineered as a way to ensure that all methods and variables are kept private until they are explicitly exposed. A frequency technique under the implementation of revealing module, is the creation of an object that publishes the required closure variables. Revealing frustrated by the fact that we are condemned to repeat the name of the closured variable, when we need to call the corresponding method from the external environment. Bellow we see an example of the revealing pattern. We observe that we have to repeat the variables that we need to publicize (*distance, init*):

```
(function(){
  var x,y;

  var init = function(_x, _y){
    x = _x; y = _y;
  };

  var distance = function(){
    return Math.sqrt(x*x + y*y);
  };

  return {
    distance: distance,
    init: init
  };
})();
```

We create a revealing module generator that takes a list of public and private methods and orchestrate the revealing logic. Hence, programmer is only responsible for the module functionality, leaving the technical implementation of revealing to the generator.

```
function genClass(mdl) {
  function getName( attr ){
    return attr.type === 'variableDeclaration' ?
    attr.declarations[0].id : attr.id
  }
  var modules = .<>.;
  for(var i=0; i<mdl.private.length; ++i) {
    modules = .< .~modules; .~mdl.private[i]; >.;
  }
  var exposed = .<>.;
  for(var i=0; i<mdl.public.length; ++i) {
    modules = .< .~modules; .~mdl.public[i]; >.;
    var name = getName(mdl.public[i]);
    exposed = .< .~exposed; expObj[.@name] = .@name; >.;
  }
  return .<
    (function(){
      .~modules;
      return (function(){
        var exprObj = {};
        .~exposed;
        return exprObj;
      })();
    });
  >.
};


var point = .!genClass(
  {
    private: [
      .< var x; >.,
      .< var y; >.
    ],
    public: [
      .< function getSum(){ return x + y; }; >.,
      .< function init(_x,_y){ x = _x, y = _y }; >.
    ],
  }
);


point.init(2,3);
print(point.x);
print(point.getSum());


>undefined
>5
```

5.3. Staged Model-Driven Generators

What is MDE?

The concept of MDE [20] has born from the requirements of software development process. The core philosophy under MDE is that we first develop a model or view of the system, and then transform into a real entity. The most important research in that concept is the Model Driven Architecture (MDA), which is founded by the Object Management Group (OMG). MDE chases three notions:

- Maximizing compatibility between systems. Ideally, developers should have an abstract view of the project, secluded from any medium requirements.
- Simplify the process of design. The model design is an individual component inside the software development process. Thus, any member or team of the development process can participate without the knowledge of the entire system.
- Optimizing the communications between individual teams working on the system, leading to better decisions.

Interface builders

Targeting in our approach for user interfaces, we can define MDE as the involvation of all the necessary tools and utilities for resolving the problem of automated user interface engineering. The current MDE tools, like *wxFromBuilder* and *eclipse MDE plugin*s are targeting on exporting user interface source code. In our work, we are emphasizing on these tools. The main idea under these tools is to generate models that are independent from the environment. Those models are automatically transformed and exported environment dependent models and source code. Thenceforth, programmer use the exported source code in the application source code logic. This process can be achieved with two different methods. The first method is by linking the source to the application. The second is to extend the exported source code by filling it with the application logic.

MDE problem

Even the rapid evolution and the sophisticated integration of MDE tools, a fundamental issue remains. The design automation of user interfaces can never cover all the aspects and the requirements of application. In real scenarios, programmer elaborate and modify the exported source code with event handlers, interaction controls and abstract behaviors. Specially, the pipeline of user interface developing process is illustrated in Figure 22. After the modification process, the source code suffers from dependencies. Consequently, programmer insert dependencies to the exported source code, unaware of the MDE tool.

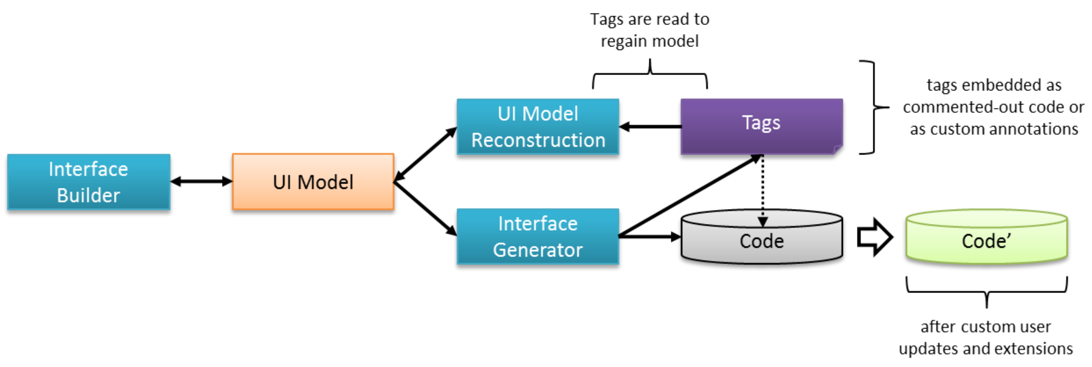

Tags are read to regain model
tags embedded as commented-out code or as custom annotations
UI Model Reconstruction
Tags
Interface Builder
UI Model
Interface Generator
Code
Code’
after custom user updates and extensions

Figure 21 – Pipeline of interactive interface builders involving: 1) interaction with the Interface builder; 2) code generation from an explicit model; 3) tags inserted in the generated source code

The problem with the above scenario is that it generate maintenance issues. More specific, in the simple scenario that user interface code remain unchanged, the exported source code is hygienic and precise. As illustrated in Figure 22, a typical develop process of a MDE based design is:

1. Generate the user interface code, as designed from the user interface builder.
2. Embed the user interface code to the application code.
3. During the integration, application code is generating dependencies to the user interface code and vice versa.

The maintenance problem appears when the generated user interface code is updated.

- Tag editing and misplacing may break model reconstruction, while any code manually inserted outside the MDE tool causes a model-implementation conflict.
- The regenerated source code overwrites the previous source code. Hence, the entire manually introduced source updates will be lost.

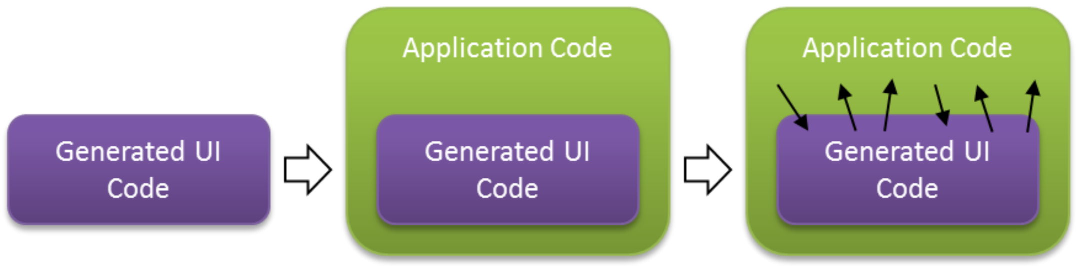


Figure 22 - Common develop process of a MDE based design

The MDE reconstruction issue in very critical for a real life application lifecycle. Even the design power and the development conveniences that MDE tools provides, it is difficult to survive in a large scale project. We solve this problem exploiting metaprogramming. We identify that using staged metaprogramming, we have the ability to preserve the exported source code of MDE tool untouched, letting the application source code grow independent. By introducing the AST in the user interface build process, we enhanced many comforts, like manipulating, processing and transforming the designed user interface. Using metaprogramming in the MDE build process, we solve the maintenance problem. Additionally, for first time in the MDE tools, we set code manipulation as a first-class concept and reveals the value of using a metaprogramming language in this context. Our work is based to a primary work that have been developed using Delta programming language [22], essentially we adapt the basic idea to JavaScript.

The MDE staged metaprogramming process swifts the user interface source code generation at build-time. Specifically, the following steps are participating at staged level to build the final application.

- MDE tools generate the concrete produced user-interfaces in a user-interface description language (UIDL). Thus, we preserve the MDE tool to remain independent from the application programming language.

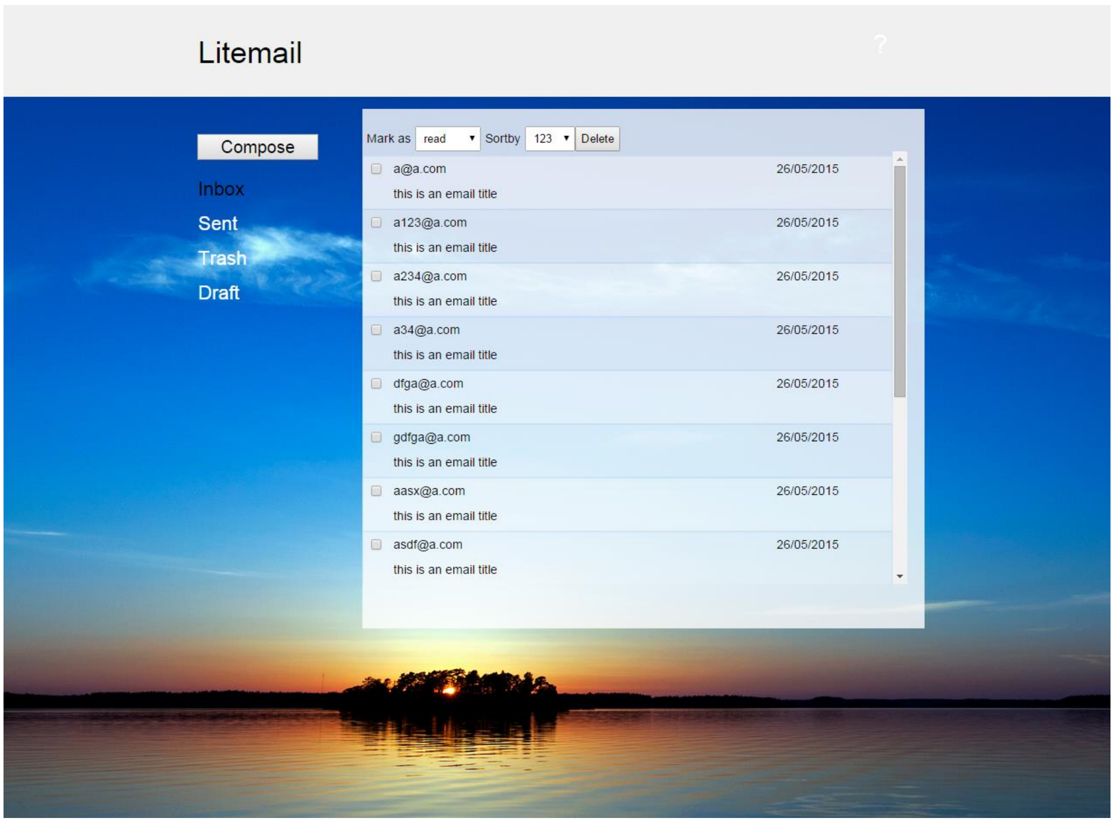
- In staged level, the exported user-interface files are taking as input, they are transformed and generate code in the form of AST. Essentially, the AST contains the regarding source code of creating user interface elements in a tree structure manner.

- The ASTs are evaluated, reading and manipulating. Programmer can elaborate the AST by cutting or merging user interface segments. He can add source code snippets, such as event handlers, and customize the exported source code at will.

- Insert the desired source fragments where needed to produce the final application.

Litemail, an email client

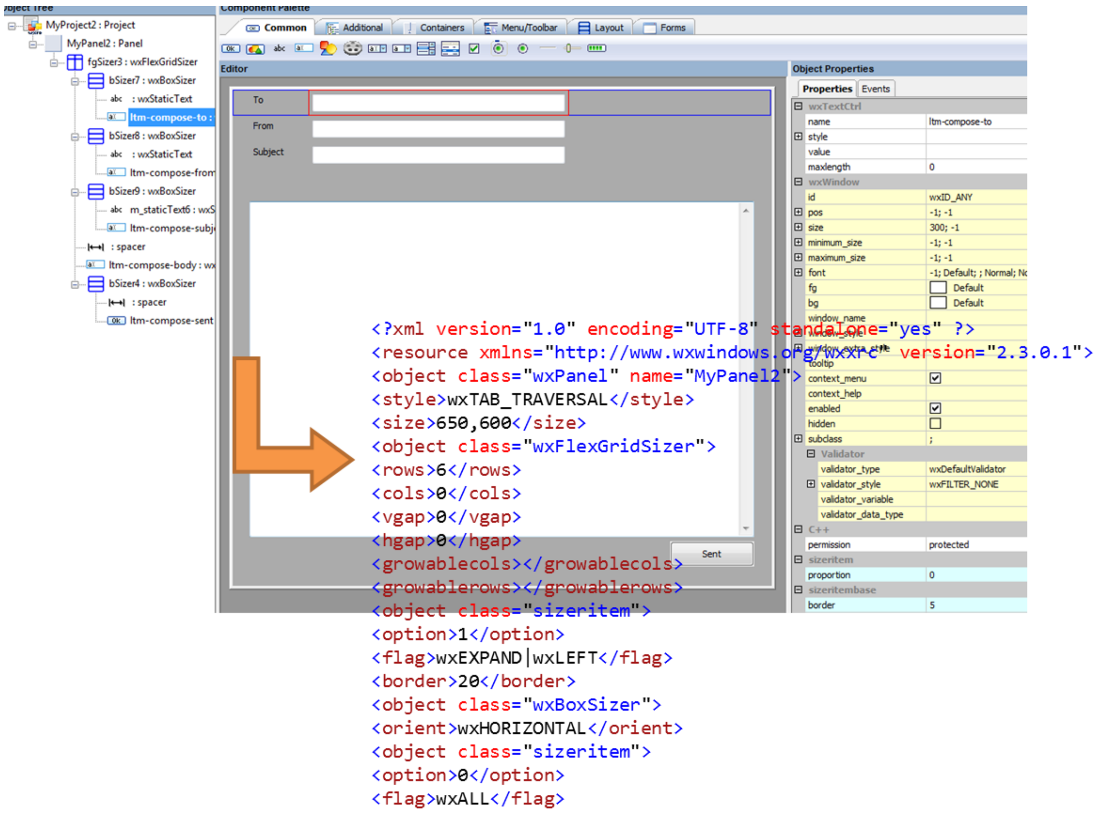
Our case study came from a real-life application, including all the challenges that occurred. We aim to bring to surface the flexibility and the powerful possibilities that endowed by metaprogramming. Our application content is an email client, called *Litemail*. The main application view is illustrated in Figure 23. Email clients have many challenging parts. Inspecting an elementary email client, we came up with many interesting components like multiple view segments, input fields, list of items and menus. We choose this example to saw that even the large demanding of the application, we easily address all the parts of the development process.



a@a.com this is an email title	26/05/2015
a123@a.com this is an email title	26/05/2015
a234@a.com this is an email title	26/05/2015
a34@a.com this is an email title	26/05/2015
dfga@a.com this is an email title	26/05/2015
gdfga@a.com this is an email title	26/05/2015
aasx@a.com this is an email title	26/05/2015
asdf@a.com this is an email title	26/05/2015

Figure 23- Litemail landing page

For the requirements of the application, we adopt the MDE authoring tool WxFormBuilder [21]. WxFormBuilder is an open source GUI designer application, written in C++, which allows creating cross-platform applications. A streamlined, easy to use interface enables faster development and easier maintenance of software. WxFormBuilder is the respectively MDE tool that we use to design our application. The user-interface description language (UIDL) that WxFromBuilder uses is the XRC, a XML-like markup language. Our initial step to the development process is to use the WxFormBuilder to design the email client user interface and export it in XRC format, as shown in Figure 24. Improving the flexibility, we separate each email client view in an individual WxFormBuilder project. The significant view elements marked by a representative unique name, simplifying the element selection in the following steps.



```
MyProject2 : Project
  MyPanel2 : Panel
    fgSizer3 : wxFlexGridSizer
      bSizer7 : wxBoxSizer
        : wxStaticText
        ltm-compose-to :
      bSizer8 : wxBoxSizer
        : wxStaticText
        ltm-compose-from
      bSizer9 : wxBoxSizer
        m_staticText6 : wxS
        ltm-compose-subj
      : spacer
      ltm-compose-body : wx
      bSizer4 : wxBoxSizer
        : spacer
        ltm-compose-sent
```

wxTextCtrl	
name	ltm-compose-to
style	
value	
maxlength	0
wxWindow	
id	wxID_ANY
pos	-1; -1
size	300; -1
minimum_size	-1; -1
maximum_size	-1; -1
font	-1; Default; ; Normal; No
fg	Default
bg	Default
window_name	
window_style	
window_extra_style	
tooltip	
context_menu	☑
context_help	
enabled	☑
hidden	☐
subclass	;
Validator	
validator_type	wxDefaultValidator
validator_style	wxFILTER_NONE
validator_variable	
validator_data_type	
C++	
permission	protected
sizeritem	
proportion	0
sizeritembase	
border	5

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<resource xmlns="http://www.wxwindows.org/wxxrc" version="2.3.0.1">
<object class="wxPanel" name="MyPanel2">
<style>wxTAB_TRAVERSAL</style>
<size>650,600</size>
<object class="wxFlexGridSizer">
<rows>6</rows>
<cols>0</cols>
<vgap>0</vgap>
<hgap>0</hgap>
<growablecols></growablecols>
<growablerows></growablerows>
<object class="sizeritem">
<option>1</option>
<flag>wxEXPAND|wxLEFT</flag>
<border>20</border>
<object class="wxBoxSizer">
<orient>wxHORIZONTAL</orient>
<object class="sizeritem">
<option>0</option>
<flag>wxALL</flag>
```

Figure 24 - WxFormBuilder adopted as user interface tool. The exported UI model defined in XRC format

The next steps are taking place during the staged process. Firstly, each XRC file is taking as input. Secondly, they parsed as an XML source text, thus, a JavaScript object produced encompassing the XML structure. Afterwards, the XRC to AST transformation process begins. Each XRC UI element transformed to the corresponding AST that represents the creation of an element using the JavaScript DOM API. Considering the application of the study case, we adopt a subset from the original XRC format. Mainly, we choose elements that are commonly and frequently used in related applications. The grammar of the subset XRC mentioned in Figure 25.

```
resource ->
object[class="wxPanel"] -> size, ?Sizer

object[class="wxFlexGridSizer"] -> rows, cols, *object[class="sizeritem"]

object[class="sizeritem"] -> flag, border, {1}sizeritem

object[class="wxBoxSizer"] -> wxVERTICAL, *object[class="sizeritem"]

object[class="wxStaticText"] -> label

object[class="wxChoice"] -> content

object[class="wxButton"] -> label, size

object[class="wxCheckBox"] -> size, label

object[class="wxListBox"] -> size

object[class="wxHyperlinkCtrl"] -> label, url

object[class="wxTextCtrl"] -> size, value, maxlength

content -> *item

Sizer -> object[class="wxFlexGridSizer"]
    | object[class="wxBoxSizer"]

sizeritem -> object[class="wxPanel"]
       | object[class="wxBoxSizer"]
       | object[class="wxStaticText"]
       | object[class="wxChoice"]
       | object[class="wxButton"]
       | object[class="wxCheckBox"]
       | object[class="wxListBox"]
       | object[class="wxHyperlinkCtrl"]
       | object[class="wxTextCtrl"]

item -> STRING

border -> NUMBER

size -> NUMBER,NUMBER

orient -> wxVERTICAL|wxHORIZONTAL

flag -> wxALIGN_BOTTOM
    |wxALIGN_CENTER
    |wxALIGN_CENTER_HORIZONTAL
    |wxALIGN_CENTER_VERTICAL
    |wxALIGN_LEFT
    |wxALIGN_RIGHT
    |wxALIGN_TOP
    |wxALL
    |wxBOTTOM
    |wxEXPAND
    |wxFIXED_MINSIZE
    |wxLEFT
    |wxRIGHT
    |wxSHAPED
    |wxTOP
```

Figure 25 - The subset of XRC format

The created AST element is appended to the parent element analogous to the XRC elements tree structure. On behalf of WxFromBuilder style mapping, we create the CSS rules respectively. Table 3 shows roughly the transformations occurred from XRC to CSS. The XRC to CSS transformation procedure is separated in 3 cases,

a) Direct mapping a single XRC to a single CSS rule (wxLEFT->padding-left)

b) Single XRC rule to multiple SCC rules (size->width, height)

c) Single XRC rule to DOM elements, when there is no corresponding CSS rule and the mapping can merely simulated by DOM elements.

XRC	CSS
wxALIGN_RIGHT	float:right
wxALL	margin: 5px
wxEXPAND	display: block
wxALIGN_CENTER_HORIZONTAL	text-align: center
Size x,y	width:x px, height: y px

Table 3 - XRC to CSS rules mapping

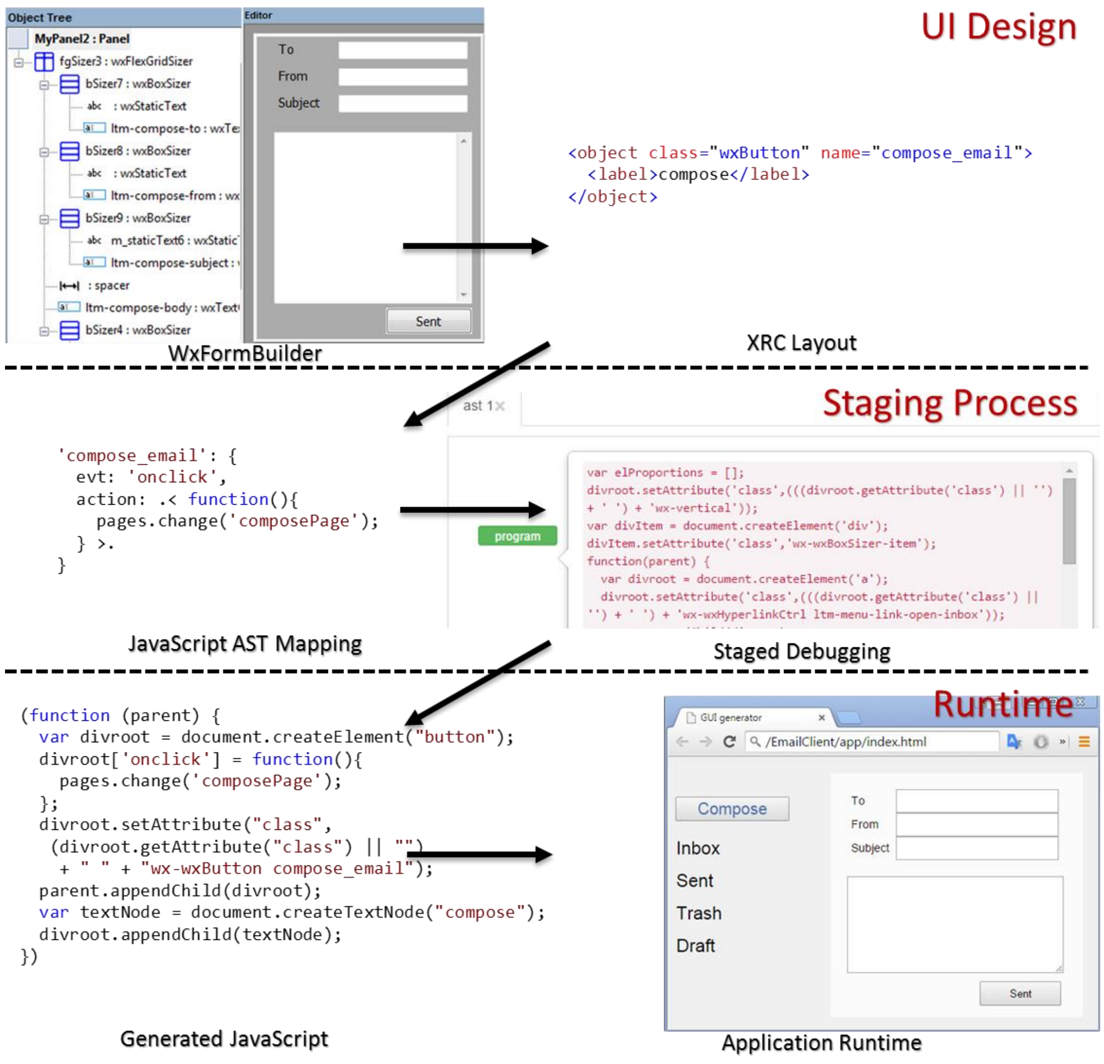
During the transformation procedure, we are inlining the event handlers at the elements that matching with the tagging name.

Afterwards, when the AST has prepared and all the necessary handlers has registered, programmer can customize the views according to the needs. For example, a typical UI design for a list of items, consists of a very common pattern. User can see a list of items and scroll through them. The list items are automatically inserted to the list using an adapter that pulls content from the source. In our case, following the common pattern, designer creates two views. The first view is regarding to how the list is illustrated and the second view that depicts how a single item seems, essentially it play the role of the adapter. Therefore, in our UI assembly process, the AST of a specific item and the AST of the list constitute two individual AST objects. Programmer has the responsibility to manner the two ASTs in a way to make possible the creation of a list of items.

```
var FunctionMappings = {
  "wxPanel"    :      function( xrc, parent, opts ) {
           var child = xrc.GetClild( 'object' );
           return child ? this[child.GetClass()](child, xrc, opts) : .< ; >.;
         },
  "wxButton"   :      function( xrc, parent, opts ) {
```

```
            var label = GetElemText( GetElem( xrc, 'label' ) );
            return .<
              ( function( parent ) {
                var divroot = document.createElement( 'button' );
                var textNode = document.createTextNode( .@label );
                divroot.appendChild( textNode );
                .~addAttributes( xrc.GetAttributes(), .< divroot; >. );
                parent.appendChild( divroot );
              } )( .~parent );
            >.;
          },
  //more functions creating html UI elements...
}
var XrcToHtml = function( xrc ) {
  return .<
    var divroot = document.createElement( 'div' );
    .~FunctionMappings[xrc.GetClass()]( xrc,  .< divroot; >. );
   >.;
}
var GenerateUIFragment = function( xrcPath ) {
  var xrcStr = ReadFile( xrcPath );
  var xrc = ParseXml( xrcStr );
  return XrcToHtml( xrc );
};
var GenerateAll = function() {
  return .<
    .~GenerateUIFragment('basicLayout.xrc');
    .~GenerateUIFragment('leftMenu.xrc');
    .~GenerateUIFragment('emailList.xrc');
    .~GenerateUIFragment('emailListItem.xrc');
    .~GenerateUIFragment('composeEmail.xrc');
    .~GenerateUIFragment('header.xrc');
  >.;
};
.!GenerateAll();
```

Above is outlined the source code for realizing this generative user-interface implementation approach. Essentially, the transformation procedure split in three parts: (i) loading of specifications into an array of objects carrying element creation information; (ii) a set of generator functions composing ASTs for the respective widget creation code, including statements which apply the visual attributes carried in the element argument; and (iii) a loop iterating on all elements and invoking generators while compositing the final AST that is eventually inlined to flush the respective code inside the program. The root AST object is inlined in the final application source code. Consequently, the source text results the staged procedure is a batch of enclosure anonymous functions, each function creates and appends to the parent, a DOM element enrich with the corresponding CSS and custom code (e.g. event handlers). The implementation size of the staged generator is about 350 lines, and remains constant in relationship to the target user-interface made with the builder and the final XRC specifications. However, the generated user-interface in JavaScript code for our e-mail client is around 1000 lines, something demonstrating the significant potential of staged generators. Finally, it should be mentioned that we realized the crucial of the staged debugging. We invested in supporting it, during the development of the e-mail client, since it resulted in a stage quite bigger than then tens of lines of code of our previous examples (really, nobody needs a stage debugger to make a correct staged power function). We briefly depict the outline of Litemail developing cycle in Figure 26.



```
<object class="wxButton" name="compose_email">
  <label>compose</label>
</object>
```

```
'compose_email': {
  evt: 'onclick',
  action: .< function(){
    pages.change('composePage');
  } >.
}
```

```
var elProportions = [];
divroot.setAttribute('class',(((divroot.getAttribute('class') || '')
+ ' ') + 'wx-vertical'));
var divItem = document.createElement('div');
divItem.setAttribute('class','wx-wxBoxSizer-item');
function(parent) {
  var divroot = document.createElement('a');
  divroot.setAttribute('class',(((divroot.getAttribute('class') ||
'') + ' ') + 'wx-wxHyperlinkCtrl ltm-menu-link-open-inbox'));
```

```
(function (parent) {
  var divroot = document.createElement("button");
  divroot['onclick'] = function(){
    pages.change('composePage');
  };
  divroot.setAttribute("class",
   (divroot.getAttribute("class") || "")
    + " " + "wx-wxButton compose_email");
  parent.appendChild(divroot);
  var textNode = document.createTextNode("compose");
  divroot.appendChild(textNode);
})
```

Figure 26 - Outline of Litemail developing cycle invlolving UI design, UI model, Javascript mapping process in staged level, debugging process, generated JavaScript, final application runtime at browser.

Chapter 6

Conclusions and Future Work

6.1. Summary

In our work, we implement a multi-staged metaprogramming system in JavaScript. We study the current implementation approaches in multi-staged programming languages. We conclude that the majority of programming languages separate the metaprogram from the normal program. Hence, they distinguish the programming language of metaprogram and the normal program, adapted particularly, custom languages for the metaprogramming requirements. We are aiming that metaprograms are essentially normal programs, consequently the metaprogramming language should be the same with the normal language. We adopt Spidermonkey JavaScript engine and fully reuse all the components. We indicate a methodological integration between the metaprogram and normal program, demonstrating that the development cost of such integration is minor. We reuse the SpiderMonkey lexer, parser, AST and runtime system, merely extending a minimum functionality to support the staging logic.

In this Thesis, we demonstrate a metaprogramming model that is at least as expressive as the traditional approaches, but offers significant advantages. Essentially, staging process involve the pipeline of parse source code, identify the staging depth, collect all the fragments of the same staging depth, assembly and evaluate respectively the staged program and finally unparse. This notions cause minimal extensions to the current SpiderMonkey implementation. Particularly, the only functionality that was omitted in SpiderMonkey was the unparse AST, henceforth we implemented from scratch, in an independent methodology.

JavaScript debugging systems are innovative and sophisticated. In our context, we reuse the JavaScript debugging facilities by exporting the staged functionality to a coherence program that is isolated from the JavaScript runtime environment aiming to externally debug the staging program. Even though the exported program is possible to run in any environment, we test the debugging consistent in the mostly-known web browsers such as Chrome, Mozilla Firefox and Internet Explorer. Moreover the traditional JavaScript debugging facilities, we add an AST inspection functionality, a corresponding AST view mechanism and a stage state indicator. Creating a collection of proper meta-debugging assisting tools.

We pass the boundaries of smaller-scale macro-related examples appearing in the literature, as we have accustomed from related work, by implementing a large scale application scenario. We address the MDE maintenance issues and offering a solution that exploits the metaprogramming principals. We implement *Litemail*, a pure email client, crossing and demonstrating all the portions of the production cycle. Additionally, we investigate JavaScript source code applications from the real world and industry, proposing source modifications in the aspects of metaprogramming, achieving runtime performance and reusability. Furthermore, studying the programming developing history, we conclude about the importance of design patterns. We show how we can modeled the design patterns as a design pattern generator library. Specifically, we emphasize in the notion of reusable design patterns, generated by metaprogramming, and indicate a variety of commonly used design patterns.

6.2. Future work

In this Thesis, we focus on the fundamental requirements indenting to integrate a coherence Multi-Staged metaprogramming environment. We show that the process of extending a current language in order to achieve metaprogramming is low and manageable. We are aiming that metaprogramming can be integrated in any language, considering thoughtful and tactical extensions. We select JavaScript for deploying metaprogramming because of the prosperity and reputation of the language. An interesting investigation could be the integration of multi-staged metaprogramming in further programming languages such as C++. A useful thought could be the reasoning of how the staged metaprogramming could replace the current C++ template custom language and the obsolete C++ preprocessor.

In the context of developing Multi-staged JavaScript, as we was coding the study cases, many requirements brought to the surface. Those requirements could massively integrated to a powerful IDE, providing meaningful and helpful assisting.

Firstly, a useful functionality is the autocomplete. The problems that came up with the implementation of autocomplete emanates from the nature of JavaScript. Autocomplete is a common issue for untyped languages such as JavaScript, since the type of a variable is not constantly determined, but defined in runtime. Even the most sophisticated JavaScript IDEs have not completely solve the specific issue. Hence, autocomplete is not a future requirement only for metaprogramming assisting, but firstly for pure JavaScript developing.

Another useful requirement is the interactive editing views. When a programmer is coding in a multi-staged metaprogramming language, the code is logically separated in staging levels. Respectively, the editor view could distinguish the staging levels with several methods, such as different background highlighting. Additionally, it would be very useful, to integrate a graphical tool that trying to identify the different transformations and flows of an AST, that created in an inner staged level, will follow until the embodiment to the final program.

Finally, an improved debugger would add a lot of possibilities in the developing process chain. Extended watchers could support powerful expressions like AST conditions regarding the concatenations and compositions. Furthermore, the enumeration of Inlines, escapes and Quazi-quotes (creation of AST) could instantly offer to the programmer options for breakpoints and proper inspections. We note that the above functionalities could build under our fundamental staged debugging system.

Bibliography

[1] Disney, T., Faubion, N., Herman, H., Flanagan, C. (2014). Sweeten your JavaScript: hygienic macros for ES5. DLS 2014: 35-44.

[2] Brian W. Kernighan and Dennis M. Ritchie. 1988. The C programming language. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition.

[3] J. Rafkind. Syntactic extension for languages with implicitly delimited and infix syntax. PhD thesis, 2013

[4] J. Rafkind and M. Flatt. Honu: Syntactic Extension for Algebraic Notation through Enforestation. Proceedings of the 11th International Conference on Generative Programming and Component Engineering, 2012.

[5] Sheard, T., Jones, S.P. (2002). Template metaprogramming for Haskell. SIGPLAN Not. 37, 12, pp. 60-75

[6] Laurence Tratt. 2005. Compile-time meta-programming in a dynamically typed OO language. In Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05). ACM, New York, USA, pp. 49-63. Available at: http://doi.acm.org/10.1145/1146841.1146846.

[7] Fleutot, F. (2007). Metalua Manual. http://metalua.luaforge.net/metalua-manual.html

[8] Subramaniam, V. (2013). Programming Groovy 2: Dynamic Productivity for the Java Developer. Pragmatic Bookshelf, ISBN 978-1-93778-530-7.

[9] JetBrains. 2013. IntelliJIDEA - Groovy and Grails. http://www.jetbrainscom/idea/features/groovy_grails.html. Accessed 11/2013.

[10] R. Kent Dybvig. 2009. The Scheme Programming Language (fourth edition). The MIT Press, ISBN 978-0-262-51298-5.

[11] Peter Seibel. 2005. Practical Common Lisp. Apress, ISBN 978-1590592397.

[12] Tim Sheard. 1998. Using MetaML: A Staged Programming Language. In: Advanced Functional Programming. 12-19 September, Braga, Portugal, Springer LNCS 1608, pp. 207-239. Available at: http://dx.doi.org/10.1007/10704973_5.

[13] Hygienic Macro System for JavaScript and Its Light-weight Implementation Framework. ILC 2014, 8th International Lisp Conference. http://dl.acm.org/citation.cfm?id=2635653

[14] Taha, W. (2004). A gentle introduction to multi-stage programming. In Domain-Specific Program Generation, Germany, March 2003, C.Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer LNCS 3016, 30-50

[15] Lilis, Y., Savidis, A. (2015). An integrated implementation framework for compile-time metaprogramming. Softw., Pract. Exper. 45(6): 727-763

[16] Mozilla SpiderMonkey Reflect.parse API https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API

[17] Chrome Developer Tools https://developer.chrome.com/devtools

[18] Firefox Developer Tools https://developer.mozilla.org/en-US/docs/Tools

[19] Libmongoose, an embedded HTTP and Websocket library https://cesanta.com/libmongoose.shtml

[20] Schramm, A., Preussner, A., Heinrich, M., Vogel, L. (2010). Rapid UI Development for Enterprise Applications: Combining Manual and Model-Driven Techniques. In proceed-ings of MODELS 2010, 13th International Conference on Model Driven Engineering Languages and Systems Oslo, Norway (October 3-8), Springer LNCS 6394, 271-285

[21] WxFormBuilder (2006). A RAD tool for wx GUIs. http://sourceforge.net/projects/wxformbuilder

[22] Yannis Valsamakis. 2013. Improved Model-Driven Engineering with Staged Code Generators. Master’s Thesis.