ABSTRACT

WRIGHT, DAVID R. Type-based Static Analysis of Web Documents with Embedded Executable Components. (Under the direction of Associate Professor S. Purushothaman Iyer)

Electronic documents have become an important medium for the exchange of information, and nowhere is this more evident than on the World Wide Web. Web-based documents can be dynamic and interactive with the inclusion of executable elements such as scripting languages, interactive multimedia components, and applets. This flexibility also brings the risks of incompatibilities between client applications, security and privacy concerns, and document integrity verification. Traditional document analysis has been restricted to examining the content of the document, as in spelling and grammar checkers, readability analyzers and keyword indexing, but these forms of analysis do not address the potential problems electronic documents may present. This thesis proposes that proven methods for analyzing computer programs may also be applied to documents with embedded executable elements. In particular, it is shown that type-based static analysis may be successfully and effectively employed to analyze a Web-based document to determine a set of browsers that will correctly render the document and provide all of the functionality incorporated into the document.

Type-based Static Analysis of Web Documents with Embedded Executable Components

by

David R. Wright

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Department of Computer Science

Raleigh

2003

Approved By:

Dr. Edward F. Gehringer	Dr. Matt Stallmann

Dr. S. Purushothaman Iyer
Chair of Advisory Committee

Dedication

This work is dedicated to my wife Sharon, without whom it would not have been possible. It was her instigation and impetus that got me started on this journey, and it has been her encouragement, patience, understanding, and sacrifice that has kept me going through all the difficult passages. With all my heart, I thank you, my darling Sharon!

Biography

I was born on Sunday, October 16, 1960, in Sewickley, Pennsylvania, a suburb of Pittsburgh, just three days after Bill Mazeroski smashed a home run in the bottom of the ninth inning to give the Pittsburgh Pirates a victory over the New York Yankees in game seven of the 1960 World Series. An auspicious beginning, I suppose! We lived in suburban Pittsburgh until June of 1971, when my father's job as an electronics engineer with Trion Corporation moved us to Sanford, North Carolina. My graduation ceremony from Sanford Central High School in June of 1978 was cut short by a sudden thunderstorm, complete with sharp lightning and booming thunder. After graduation, I started working as an electronic technician for a local TV shop, and attended NC State part-time for two semesters in 1979 and 1980. My dream in high school was to work as a researcher in the (then) new field of high-energy physics. The lack of time and money took its toll, however, and I did not continue to pursue that dream.

In 1981 I got married, and over the next five years became the proud father of two children. For many reasons, the marriage did not last, but I am still proud of my kids! My daughter, Leia (named after the princess of *Star Wars* fame), is now a sophomore majoring in Elementary Education and Sociology at UNC-Greensboro. My son, Jonathan, just turned 16 this year, and is a junior at Chatham Central High School in Bear Creek, NC, where he lives with his mother. He has been interested in computers since he was little, and is seriously considering NC State to pursue a degree in Computer Science.

In 1997, I married Sharon McCarthy, a New Jerseyite by way of Florida. We met in a bowling alley, of all places (there is hope for us bowling geeks!). Our wedding was a intimate affair in Lake Tahoe, Nevada, less than two weeks before I became a freshman student at State, and just three months after she graduated from Meredith College with a degree in Sociology. The adjustment to full-time college student from full-time employment was difficult at times, and the last five years have not been easy, but Sharon and I have grown much closer as a result. I could not have made it this far without her!

My initial plan when I restarted my collegiate career was to get through school as quickly as possible and get back to work (for the big bucks!). It did not take long, however, before the dream I had laid aside so long ago began to resurface. I realized that the field of high-energy physics had become an elite society due to the expense of research facilities, but the theoretical aspects of Computer Science offered a wealth of new and exciting challenges,

just as physics had done twenty years before. It was not long before I had decided to pursue a graduate degree, with the lofty goal of a Ph.D. and a new career in academia. This thesis brings me one step closer to that goal.

Acknowledgements

There are so many people I need to thank for their help, inspiration, and encouragement to get me to this point. At the top of the list, Dr. Purush Iyer, my advisor, without whose patience, guidance, and reality checks I would not be here. Some of the many instructors and professors I have taken classes from or worked with as a Undergraduate Teaching Assistant who helped inspire me to continue my education include Carol Miller, Jon Rossie, Matt Stallmann, Dana Lasher, Ed Gehringer, Joyce Hatch, Carla Savage, Bruce Weiand, Robert Fornaro, Margeret Heil, and many others. To my fellow students and TA's that I've worked with, thank you!

I must also thank my family, my children Leia and Jonathan, my parents Richard and Margery, my sisters and brother, Marilyn, Marion, and Rob, as well as my extended family locally and back in Pennsylvania and New York for their encouragement and understanding when I have not been able to spend as much time together as I would like.

Trademark Acknowledgements

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

2 Problem Definition 4

3 Related Work 7

4 Type Rules 12
4.1 Introduction . . . 12
4.2 Basic Concepts and Definitions . . . 14
4.3 Document Structure and Type Determination . . . 20
4.4 JavaScript Runability Analysis . . . 33
4.4.1 Source Elements . . . 34
4.4.2 Function Declarations . . . 38

5 Implementation 66
5.1 Determination of Type Classes . . . 66
5.2 Design Considerations . . . 68
5.2.1 HTML Parser Considerations . . . 69
5.2.2 JavaScript Parser Considerations . . . 69
5.3 Implementation Details . . . 70
5.3.1 Modified HTML Tree Structure Module . . . 72
5.3.2 Type System Module . . . 73
5.3.3 Environment Module . . . 73
5.3.4 Runability Type Analysis Module . . . 74

6 Case Study 79
6.1 Background . . . 79
6.2 Test Cases . . . 81
6.2.1 NCSU Homepage . . . 81

6.2.2 Microsoft Homepage 82
6.2.3 Mozilla JavaScript Testcase 82
6.2.4 Google Search Engine 83
6.2.5 CBS Homepage 84
6.2.6 WRAL TV Homepage 84

7 Conclusions and Future Work **86**

Bibliography **89**

A The `abstractDocument` Structure **91**

B Built-in Identifiers and Runability Types **94**

List of Figures

4.1 Example of Browser-sniffing JavaScript Code 13
4.2 Example of Browser-specific JavaScript Code 13
4.3 Browser-specific HTML Code 22
4.4 Browser-specific HTML Code with JavaScript 23
4.5 Non-browser-dependent HTML Code 24
4.6 Context-dependent, Browser-specific HTML Code with JavaScript 24
4.7 Browser-specific HTML Code with JavaScript 25
4.8 Examples of JavaScript Pseudoprotocol URLs 29
4.9 Examples of JavaScript Entity References 30
4.10 Example of Browser-dependent Attribute Value 32

5.1 Implementation of Runability Analysis 71
5.2 Simple Visitor Implementation in Java 75

List of Tables

6.1 Test Case Runability Results 85

B.1 HTML Tag Name Groups 94
B.2 HTML Runability Types by Group 95
B.3 Built-in Core JavaScript Identifier Groups 95
B.4 Built-in Core JavaScript Runability Types by Group 96
B.5 Built-in Client-side JavaScript Identifier Groups 97
B.6 Built-in Client-side JavaScript Runability Types by Group 98

Chapter 1

Introduction

In a general sense, a document is a collection of related information, presented in an orderly manner. A document might be a receipt for a business transaction, correspondence between two individuals, an audio or video recording, or a written report, among many other possibilities. Traditionally, a document's presentation is fixed by the author at the time of creation, and the structure, content, and presentation of the document are unchangeable without the active intervention of an external agent, if at all. In recent years, a new kind of document has emerged. Electronic documents can be dynamic and interactive, with presentation and content customized to the device used to view the document and to the viewer's preferences. Word processing files, spreadsheets, World Wide Web (WWW) pages, and electronic mail (e-mail) messages fall into this category, and may contain elements that are designed to be executed by the viewing device.

These executable components bring a new set of challenges with respect to security and integrity, both to the document itself and to the device used to present the document. Executable elements incorporated into the document must be correctly processed by the viewing application if the document is to be rendered as the author intended. Users who access such a document have the expectation that the document will represent the author's intent, and will not modify or harm the device used to access the document. Standards for scripting languages, either published or ad-hoc, generally only specify the syntax and core functionality of a language. Different implementations of scripting languages may also incorporate additional features and functionality, particularly with respect to the abstract

model used to access and manipulate a document. Authors may rely on implementation-specific features in their documents, often without knowing that their embedded script has introduced dependencies and/or incompatibilities into the document.

The responsibility for ensuring and maintaining the integrity of a document's presentation rests solely on the creator of the document at this time. A user viewing a dynamically constructed document must trust that the device used to access the document will correctly present it as the author intended. From the author's perspective, the only way to ensure that a document will be correctly rendered is to view the document on all the possible devices a user might employ to access the document. Obviously, this can be a time-consuming process, at best, since the document would need to be tested on a possibly broad array of hardware and software platforms . With the rapid pace of change in computers, personal data assistants (PDAs), wireless telephones, and many other devices that may access a WWW-based dynamic document, actually testing a document's presentation on even a modestly representative sampling of these platforms is a difficult proposition. Someone accessing such a document must rely on the creator to know the capabilities and limitations of the device used to view the document, and to have incorporated appropriate elements to ensure that when the document is accessed, it is presented in the manner intended.

This thesis introduces DRAT, a Document Runability Analysis Tool intended to analyze a HyperText Markup Language (HTML) document to determine a set of Web browsers that will correctly render the document. This tool could be used by both document authors to easily determine the compatibility of their documents without requiring the maintenance of a potentially large collection of hardware and software to test the correctness of document display on different browsers and platforms. If an author can quickly and reliably determine which browsers correctly render a document and which do not, corrections can be made to enable a wider audience to view the document. Alternatively, an author could also provide an indication of any browser-specific elements within the document. End users could also use this tool to determine if the browser they are using to display a document is capable of correctly rendering it as the author intended. Such a tool could make the WWW a more user-friendly and reliable source of information, entertainment, and communication.

There are many challenges in designing such a tool, including rapidly changing browser versions, the evolution of new platforms such as wireless telephones, internet ap-

pliances like dedicated email terminals, and the incorporation of internet and WWW communication capabilities into other devices not normally considered to be communication devices, such as refrigerators with internet access that maintain inventories of their contents and allow users to order groceries directly from the unit. There are several scripting languages that may be incorporated into a Web page, and the implementation of these languages can vary from from platform to platform as well as between versions of a particular browser. The lack of detailed documentation on these scripting language implementations also contributes to the lack of compatibility of web-based documents, as well as to the lack of thorough test suites for browsers that could be used to determine which languages and language constructs are correctly rendered by a particular browser.

The second chapter of this thesis will examine in more detail the problems associated with document security and integrity mentioned earlier in this section. Chapter three discusses related work that provided the basis for this research and influenced the design process. Chapter four covers in detail the theoretical basis for DRAT, establishing the groundwork for the implementation discussed in chapter five. Case studies using DRAT are examined in chapter six, and the thesis concludes with a summary and discussion of future work.

Chapter 2

Problem Definition

Maintaining the accuracy and integrity of a document has always been a problem for document distribution. Before the invention of the printing press, documents had to be copied by hand, and transcription errors were possible as a result of both illegible originals and the possibility of editing by the individual making the copy. The printing press allowed the preparation of many identical copies, although the accuracy was still dependent upon the typesetter. In both cases, however, the copies could be reviewed by the original author for their accuracy before distribution. Once that had been verified, the author could be assured that anyone viewing one of those copies would be viewing the correct content, as the author intended. The document was, in essence, fixed in the medium of distribution.

Audio and video recordings introduced another variable into the chain of distribution, particularly in the case of recordings made on magnetic tape. While the accuracy of the copies of the original could be verified and the distribution medium is fixed at the time the copies are made, incompatibilities and defects in reproduction devices could introduce artifacts into the document that did not exist in the original, or the devices might not reproduce the document as the creator intended. The establishment of recording and reproduction standards, along with readily available test recordings, allowed the document viewer to determine what limitations their reproduction equipment might have, as well as how it would affect the playback of a recording. Document creators could also determine how their recordings would be reproduced on equipment conforming to these standards, and have the opportunity to modify their recording to make it accessible to a wider audience

via a broader set of reproduction devices, minimizing the risks to document integrity from distribution.

Electronic documents such as e-mails and Web pages add another, more complex problem to the mix. These documents do not have a fixed medium for distribution in the same sense that earlier document forms have, since they contain *markup* intended to specify the presentation of the document. Organizations such as the Internet Engineering Task Force (IETF), the Institute for Electrical and Electronic Engineers (IEEE), and the International Telecommunications Union (ITU) publish standards for communication networks. These standards relate to the actual stream of bits that represent a document in the process of transmission, and devices that conform to these standards can reliably distribute the basic data that makes up a document. Thus an author may be reasonably assured that the actual data is received accurately and intact by someone wishing to view the document. These standards include issues such as error correction and recovery schemes and encryption/decryption mechanisms that further enhance the reliability and integrity of the distributed document.

The actual display or rendering of the document is done by a software agent at the recipient's end of the distribution. This software might be a Web browser, an e-mail client, or some other application designed to decode the stream of transmitted bits into a copy of the original document. In a sense, this software might be considered analogous to the scribe or typesetter responsible for making copies of a document, including the problems those kinds of copying may introduce. The rendering software agent might not "understand" all of the constructs in the transmitted data, and either ignore them or attempt to display those elements in another form. Different programs may be written to render the same element in different ways. Users might also be able to force their own preferences on the display of the document, overriding the author's intent. Finally, the device used to view the document might not be able to display the document at all. The important consideration is that the creator of the document cannot review the proposed presentation to ensure its accuracy and integrity, as was possible with a hand-copied or typeset text. Additionally, the recipient may not be aware of differences between the displayed document and the original. The author's only recourse is to test the rendering of the document with as many possible software agents on as many different platforms as possible. As noted earlier, this is increasingly difficult as the number of possibilities grows at an ever-increasing pace.

Further complicating this situation are dynamic documents containing executable

scripts or other code. Not only do authors of these documents have to be concerned with the rendering of the document's data, they must contend with variations in the implementations of the software that processes these executable components. Incompatible code may be ignored or misinterpreted on the client side of the distribution, resulting in an incomplete or incorrectly displayed document at best, and may cause fatal or non-fatal errors in the rendering program. These errors may cause the client program to "crash", or even have serious effects on the recipient's entire device. Of course, a malicious author might intentionally create document of this sort for the purpose of causing problems on client devices. The security of the client system may also be compromised without the user's knowledge or consent, allowing the document's author to access private data about the user, i.e. credit card and bank account numbers, private correspondence, and other information the user would not wish to be shared. Potential viewers have no means to determine how such a document will affect their device or how a dynamic executable document will affect their system when it is loaded.

Most Web browsers and similar programs allow disabling of scripts and other executable code in the user preferences, but this is a global setting, disabling all executable content within displayed documents. Word processors and spreadsheet applications offer similar settings, but with the same consequences, although as these programs are more file-oriented, embedded executables may be disabled only for a particular file. In any case, the user usually does not know what effects the document will have until the document is actually loaded, but by then it is too late if the document contains executable elements that may cause problems on the client system. A tool that can evaluate the behavior and compatibility of a document with respect to a particular rendering or displaying program and environment is clearly needed. Such a tool would be equally useful to both document developers and end users, and would enhance the reliability, accuracy, and safety of distributed documents and the systems used to communicate and display them.

Chapter 3

Related Work

As noted in the previous chapter, the problem of verifying the integrity and accuracy of a distributed document is not new, although recent technological advances have introduced new challenges for authors and readers. Delegating the responsibility for rendering a document to the client requesting it weakens the chain of control a document creator has over the intended presentation. This work seeks to show that it is possible to discover characteristics about a document that can be used to determine whether or not a document will be presented as the author intended. Remarkably, there seems to be little interest in this verification process, as exhaustive searching of related literature has been unsuccessful at finding any references to the subject. The current work in this area has been directed towards developing more detailed standards for document construction and markup, and specifying how client systems interpret and render these markup elements. The leader in this effort is the World Wide Web Consortium (W3C), a collaboration of industry and research organizations whose mission is to "lead the Web to its full potential" [5] through the development of technologies supporting an secure, interoperable, and accessible Web environment.

With respect to Web-based documents, the W3C has published several recommended standards. The HTML 4.01 [25] and XHTML [7] recommendations specify standards for structured HTML documents with a limited but well-defined set of tag names and a mechanism to extend the set of tag names in a machine-understandable manner. The recommendation includes guidelines for how each tag should be rendered. Supporting

these recommendations are standards concerning document style sheets [6] and a Document Object Model (DOM) [19] that defines a model for programmatically accessing elements in an HTML document. Included in the DOM recommendation are suggested bindings for the DOM elements in the JavaScript language. While these standards are a necessary step towards the goal of achieving reliable presentation of a document with respect to the author's intent, they are limited by several factors. First, most Web browsers and other software applications for rendering these documents are not fully compliant with these standards. Second, document authors must understand these standards in order to prepare and publish conforming documents. Third, assuming fully compliant browsers, client users must be using these programs to access and view conforming documents. Finally, there are many Web pages already published which are not conformant to these standards. Reworking these pages can be expensive in terms of both money and time, an expense which many authors might choose not to accept until there is sufficient demand, if at all. These standards do not address the complications introduced by dynamic documents that incorporate executable components, other than how those components may be embedded into a HTML document.

The European Computer Manufacturers Association (ECMA) addressed part of that problem by publishing a specification for the ECMAScript scripting language [2]. This specification defines the syntax and semantics of ECMAScript, which forms the basis for JavaScript, developed by Sun Microsystems and Netscape Communications, and for JScript, developed by Microsoft. However, the specification does not address the client-side implementation of these languages, defining only the core constructs and native objects a conforming implementation should provide. Client interpreter implementations are at the discretion of the client developers. Since the actual work of rendering a document is performed by a client application, differences between client implementations of the language, and in particular, the differences in the modeling of a document to allow access and modification of document elements are critical. Prior to the development of the W3C DOM, Netscape Navigator and Microsoft Internet Explorer, the most popular Web browsers, employed two significantly different and incompatible document models, requiring authors to incorporate "browser sniffing" code into their documents to detect which browser was in use and brand code execution appropriately. Of course, not all authors took the time to include this code, and their document will only be correctly rendered on a limited set of browsers. With the release of version 6 browser software, Netscape abandoned their earlier document model in favor of the W3C DOM, although this has yet to be fully implemented. Microsoft

began supporting some of the W3C DOM bindings in version 5.5 of Internet Explorer, but has not eliminated support for their earlier document model. Thus, at this time, the problem has actually become more complex with the addition of the DOM standard. Instead of two models to manage, there are now three, and the third is implemented to different extents in different browsers and versions. Netscape and Internet Explorer, while accounting for most of the browsers in use, are not the only players in the game. Web browsing programs may be embedded into other devices like cellular telephones and appliances as noted earlier. The additional code required to detect and distinguish between the major browsers (Netscape and Internet Explorer) is becoming more complex without considering other possible applications, and making the reliable testing of a document's rendering even more difficult

An alternative to more extensive and time consuming testing of documents is to analyze a document with respect to the browser-specific elements it incorporates to determine which browsers should correctly render the document. This work treats a document as a program, which is not an unreasonable point of view. Even a HTML document that does not contain any embedded executable script may be abstracted as a program, since the HTML tags delimiting different elements in the document can be considered commands to the rendering application, specifying how each element is to be displayed. Tag names have been standardized in the W3C recommendations, but there are several of these tags that have limited or no support in most browsers. Netscape and Microsoft have also introduced other tags that are ignored by each other's browsers, for example Netscape's layer and Microsoft's marquee tags. Embedded script may also include browser-specific statements or expressions that can affect how the document will behave and be rendered by a client application. Examination of language constructs, function names, and document model identifier names can provide information about which browsers can correctly evaluate expressions that access these identifiers.

This work builds upon existing work in proving the correctness of programs, structural operational semantics, type systems, and type inference, developing a system to determine the behavioral characteristics of a document. The characteristic of interest is a set of Web browsers that will correctly "execute" a HTML document that may contain embedded JavaScript code to dynamically alter the document based on client-side criteria determined at runtime. As Hantler and King note in [18], the desire to prove that computer programs behave as intended has existed since the first modern computers were built and

programmed. While their work centered on proving a program's correctness and required the programmer to include "correctness assertions" to enable their analysis, the symbolic execution model they used provided guidance for developing some of the type rules defined in Chapter 4. In particular, a correct analysis of conditional constructs such as `if` statements requires evaluating the constituent parts of the statement as with respect to all possible execution paths.

Fundamental to this analysis of a document is the concept of structural operational semantics (SOS), introduced by Plotkin [24]. The interesting parts of the document, insofar as this analysis is concerned, are not the actual content or data to be displayed, but the mechanisms used by the document author to instruct the client software about how to render the document. These instructions are in the form of HTML tags and JavaScript statements, and in order to correctly analyze a document, an abstraction must be generated that models the command structure the author specified. Semantics for HTML and ECMAScript are included in their respective language specifications [7, 2], but the interaction that occurs between the two languages is the key to understanding and correctly constructing the abstraction to be analyzed. Plotkin's method [24] was invaluable in this process, which formed the foundation upon which the type determination rules could be defined.

Investigating the rendering incompatibilities among various Web browsers, I found that the key differences were not in the implementation of the basic HTML or JavaScript parsers, but in the abstract document model used to manipulate the document contents during the rendering process. For example, in Netscape Navigator 4.x browsers, the `layer` tag is the only HTML element whose contents the JavaScript interpreter can dynamically alter. The Netscape 4.x document model implementation also includes an array of all layers in the document, accessible either by a zero-based index, or using the `name` attribute of the `layer` tag. Microsoft Internet Explorer, beginning with version 4.0, maintains an array of all elements in the document, accessed using an integer index or by specifying the `id` attribute of the desired element. Internet Explorer also completely ignores any `layer` tags in the document. These behavioral differences were the inspiration for developing a set of types denoting the set of browsers that correctly "understands" particular identifiers. These identifiers include HTML tag names, document object model objects and properties, and other function names built in to the interpreter. By determining the "Runability types" of all identifiers in a document combined with their context, it would then be possible to determine the set of browsers that would correctly reproduce the document.

In this respect, the language of the document structure is a statically-typed programming language, using inferred types rather than explicitly stated types, analogous to languages like ML and Scheme. Construction of a type system, as described by Schwatrzbach [26] and Milner, Tofte, and Harper [21], would provide a mechanism to infer the "Runability Type" of a document. The following chapter details the construction of this type system and defines a set of type rules to be used to determine the set of browsers capable of properly rendering a particular document. This type system is specific to HTML documents that may contain embedded JavaScript elements. While targeted to a specific set of documents, the basic framework is extensible to other document types and/or other embedded scripting languages. Modification of the type classes themselves could also allow this same basic analysis technique to provide other additional information about a document. One example would be to examine a document in order to determine if different content, i.e., the data between HTML tags, would be displayed on different browsers.

Chapter 4

Type Rules

4.1 Introduction

Precisely defined rules are necessary to accurately analyze a JavaScript program or JavaScript based hypertext document. These rules will relate grammatically correct JavaScript or HTML code to a *runability type* that denotes the set of web browsers that will correctly render or process the source code. Perhaps the most significant problem we face in the analysis of this source code is that a document may contain code that denotes compatibility with a particular browser, but will execute on other browsers. Figure 4.1 is a JavaScript code segment typically used to "sniff" a browser type and set a variable that other code can use to execute browser-specific code. This code attempts to access implementation-specific objects to determine the browser type. Each of the `document` object accesses is specific to a particular set of web browsers. The default behavior of JavaScript is to return a `null` value if an object is not implemented, which is interpreted as a *false* boolean value in the parenthesized part of the conditional expressions. This allows any one of these statements to execute without errors on a JavaScript-enabled browser, yet each statement implies the requirement for a particular browser or browsers.

There are also cases where a particular code segment will not execute on some browsers at all, triggering an error condition. The JavaScript `try` statement, illustrated in Figure 4.2, is an example of this situation. The `try` statement causes Microsoft Internet

```
var w3c=(document.getElementById)?true:false;
var ns4=(document.layers)?true:false;
var ie4=(document.all && !w3c)?true:false;
var ie5=(document.all && w3c)?true:false;
var ns6=(w3c && navigator.appName.indexOf("Netscape")>=0)?true:false;
var ieX=(!ns6 && (ie4 || ie5));
```

Figure 4.1: Example of Browser-sniffing JavaScript Code

```
try {
    var n = prompt("Please enter a positive integer", "");
    var f = factorial(n);
    alert(n + "! = " + f);
}
catch(ex) {  //if the user's input was not valid, we end up here
    alert(ex);
}
```

Figure 4.2: Example of Browser-specific JavaScript Code

Explorer 4.0 to halt both the execution of JavaScript code and the rendering of the HTML document containing the embedded JavaScript. Netscape 4.x and Internet Explorer 5.0 generate errors, but may continue to process and/or render the document, depending upon where the `try` statement occurred. Clearly, both of these situations place restrictions on the overall runability of a document, but they do so in different ways, especially when these restricting code segments are combined with other code that may or may not place other restrictions on the runability of the overall document. The analysis rules must correctly distinguish between these two kinds of code restrictions.

In this chapter I develop a set of rules for determining the runability type of a JavaScript program or an HTML document with embedded JavaScript code. Following this introduction, Section 4.2 establishes the notation that will be used to define these rules, and defines the constructs that will be fundamental to the runability analysis. Section 4.3 defines the abstract structure of a document and the rules to determine a document's runability type. Rules for determining the runability type of JavaScript code are detailed in Section 4.4.

4.2 Basic Concepts and Definitions

Before developing the type analysis rules, some basic definitions and assumptions must be established. This section will describe the fundamental concepts necessary for a runability type analysis. The notation and terminology used in the development of typing rules is defined, as are the basic structures upon which the these rules will depend. The section will conclude with the introduction of the first runability type rule.

The term *document* will be used to refer to an input source under analysis. In the most general terms, such an input source is a sequence of characters that may be interpreted as HTML or JavaScript source code. More precisely, I define a document as follows:

Definition 4.2.1. The Set of Documents

> A **document** is a web-based resource, identified by a Uniform Resource Locator (URL), consisting of a sequence of ASCII- or Unicode-encoded characters, and define $\mathcal{D}$ to be the set of all **document**s.

In order to analyze a document, an abstract structure must be applied to the sequence of characters. This structure allows the document to be decomposed into discrete elements for which a runability type can be determined. For this analysis, this structure may be a collection of HTML elements, a collection of JavaScript source elements, or a simple sequence of characters containing no HTML or JavaScript code. Formal definitions of this abstract structure and parsing function is given in Definitions 4.2.2 and 4.2.3.

Definition 4.2.2. The Set of `abstractDocuments`

> An **abstractDocument** is a structure that may be applied to a document. Let $\mathcal{P}$ be the set of all possible **abstractDocument**s.

The parse tree-like structure of a **abstractDocument** will be developed throughout the remainder of this chapter, and a complete listing of the structure may also be found in Appendix A.

Definition 4.2.3. The Parsing Function

> The parsing function
>
> $$\triangleright : \mathcal{D} \rightarrow \mathcal{P}$$
>
> written as $d \triangleright p$, where for each $d \in \mathcal{D}$ the function identifies a $p \in \mathcal{P}$ as the **abstractDocument** structure applied to a document d.

The purpose of this analysis is to determine the set of browsers required to correctly process and/or render a document. In general, the analysis described in this chapter may be applied to any finite set of web browsers. In order to simplify this discussion, I consider a limited set of browsers, namely Netscape and Microsoft Internet Explorer versions 4 and later. There are two reasons for choosing this set. First, browsers prior to version 4 did not implement any JavaScript mechanism enabling dynamic modification of a web document. Second, this set of browsers accounts for over 95% of of the web browsers currently in use [8].

Definition 4.2.4. The Browser Set for Document Analysis

> Define β to be an arbitrary set of browsers that are the basis for a document analysis.
>
> $$\beta = \{\text{Netscape4}, \text{Netscape4.5}, \text{Netscape6}, \text{MSIE4}, \text{MSIE5}, \text{MSIE5.5}, \text{MSIE6}\}$$

As was mentioned earlier, the purpose of this analysis is to determine two pieces of information about a document with respect to a set of web browsers. First is a set of browsers, each of which will correctly process and render the document. Second is another set of browsers, each of which will process the document without generating an error condition that halts processing. Note that this second set may include browsers that do not correctly render the document. The concept of runability type, and the symbols that will be used to denote a runability type and its constituent set is given in Definition 4.2.5.

Definition 4.2.5. The Runability Type

> Define runability type to be $\mathcal{R} = 2^{\beta} \times 2^{\beta}$. A particular runability type will be denoted by the symbol Γ, and is defined as $\Gamma = (\delta, \eta) \in \mathcal{R}$, and having the additional constraint that $\delta \subseteq \eta$, capturing the fact that δ is the set of browsers that *correctly* execute and/or render a **document** and η is the set of browsers that will process a **document** without failing.
>
> As a consequence of this definition, note that $\Gamma = (\beta, \beta)$ implies browser independence with respect to the browser set β.

Many of the type rules developed in this chapter involve operations to merge types. Tow of the most common operations are the least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) which are defined below.

Definition 4.2.6. The Least Upper Bound of Two Runability Types

If $\Gamma_i = (\delta_i, \eta_i)$, then the least upper bound of two runability types is given by:

$$\Gamma_1 \sqcup \Gamma_2 = (\delta_1 \cup \delta_2, \eta_1 \cup \eta_2)$$

Definition 4.2.7. The Greatest Lower Bound of Two Runability Types

If $\Gamma_i = (\delta_i, \eta_i)$, then the greatest lower bound of two runability types is given by:

$$\Gamma_1 \sqcap \Gamma_2 = (\delta_1 \cap \delta_2, \eta_1 \cap \eta_2)$$

A notion of an evaluation context or environment is also necessary for a runability type analysis. This environment should include runability type information about constants such as HTML tag names and built-in JavaScript identifiers, a facility to augment the environment with user-defined identifiers defined within JavaScript code, and a runability type of the context. The runability types of HTML tag names and built-in identifiers are dependent only upon the tag or identifier name itself, and is not influenced or modified by any particular document. I begin building a definition of an environment by defining the set of names associated with HTML tag names and a relation from these tag names to their runability types.

Definition 4.2.8. HTML Tag Names

Let $\mathcal{H}$ be the set of HTML tag names, ranged over by h. Let $\gamma_{tag} : \mathcal{H} \rightarrow \mathcal{R}$ be the function mapping tag names to their runability types.

The ECMAScript specification[2] defines a set of standard built-in objects and functions forming the core of a web browser's JavaScript implementation. A particular interpreter implementation may also define additional built-in objects. These objects include, but are not limited to: built-in functions, client-side objects that provide an interface to the host system, Document Object Model (DOM) objects, and DOM object properties and methods. Within JavaScript code elements, a document author may declare and define additional identifiers associated with variables and functions. Definition 4.2.9 defines this set of identifier names and a relation from these identifier names to their runability types.

Definition 4.2.9. Built-in JavaScript Identifiers

Let Σ be the set of all built-in JavaScript identifier names, ranged over by n. Let $\gamma_{id} : \Sigma \rightarrow \mathcal{R}$ be the function mapping JavaScript identifier names to their runability types.

Another component of the evaluation environment is the set of user-defined identifier names, in particular the set of identifiers associated with user-defined variables and functions within a JavaScript program. The language specification [2] defines the syntax of correctly named user-defined identifier names. Definition 4.2.10 formally defines this set of identifiers.

Definition 4.2.10. User-defined Identifiers

> Let $\mathcal{I}$ be the set of all syntactically correct user-defined JavaScript identifier names, ranged over by i. Let $\gamma_{uid} : \mathcal{I} \rightarrow \mathcal{R}$ be the function mapping user-defined identifier names to their runability types.

An evaluation environment includes all of the identifiers available in a particular context, including HTML tag names, built-in JavaScript identifiers, and user-defined identifiers. As noted above, the environment also has a runability type describing the browser requirements for the environment's context.

Definition 4.2.11. The Analysis Environment

> The set of all environments
>
> $$\Phi = (\mathcal{H} \rightarrow \mathcal{R}) \times (\Sigma \rightarrow \mathcal{R}) \times (\mathcal{I} \rightarrow \mathcal{R}) \times \mathcal{R}$$
>
> is a set of 4-tuples, where each environment $\phi = (\gamma_{tag}, \gamma_{id}, \gamma_{uid}, \Gamma)$ has functions to associate tag names and JavaScript identifiers with runability types, as well as capturing the composite runability type of the environment. Since the context disambiguates between tag names, built-in JavaScript identifiers, and user-defined identifiers, $\phi(n)$ is defined as:
>
> $$\phi(n) = \begin{cases} \gamma_{tag}(n) & \text{if } n \text{ is a HTML tag name} \\ \gamma_{id}(n) & \text{if } n \text{ is a built-in JavaScript identifier} \\ \gamma_{uid}(n) & \text{if } n \text{ is a user-defined JavaScript identifier} \end{cases}$$

Because an author may define variables and functions within JavaScript code, notation must be defined to indicate the augmentation of the environment with the identifier associated with the variable or function. This allows the runability type of a previously initialized identifier to be "looked up" later in the analysis when the identifier is accessed. The definition must also specify the semantics of the augmentation. In particular, adding an identifier to the environment does not require the runability type of the environment to be modified. At the time of variable or function definition, the only modification to the environment that is necessary is the addition of the identifier name to the environment's identifier set Σ.

Definition 4.2.12. Augmentation of the Environment with a Single Identifier

For an environment ϕ, let $\phi[id/\Gamma]$ denote a new environment ϕ', such that

$$\phi'(n) = \begin{cases} \phi(n) & \text{if } n \text{ is a tag name} \\ \phi(n) & \text{if } n \text{ is a JavaScript identifier and } n \neq id \\ \Gamma & \text{if } n \text{ is a JavaScript identifier and } n = id \end{cases}$$

To clarify the notation in the runability type rules that comprise the remainder of this chapter, I make several notational definitions. As noted earlier in this section, the analysis of a document requires the application of an abstract structure to enable the decomposition of the document into discrete elements. In order to indicate that abstract structure and denote how these structures are built from, and can be decomposed into, other structures and/or constant terms, I shall use a "*constructor*" notation.

Definition 4.2.13. Constructor Notation for Structure Elements

The notation `structName`(`term`$_1$, `term`$_2$, ..., `subStruct`$_n$) denotes the *constructor* of an abstract structure element `structName` from one or more abstract structure elements or constant terms `term`$_1$ ... `term`$_n$.

A structure will always have at least one constructor. I use the symbol "`::=`" to associate an abstract structure element with the constructor(s) for that element. I will also use the symbol " | " to separate multiple constructors associated with the same abstract structure. Definition 4.2.14 formally defines this notation.

Definition 4.2.14. Constructor Set Notation

Let **structName** be an abstract structure element with k constructors. The notation

$$\begin{aligned} \textbf{structName} \quad ::= \quad & C_1(t_1^1, \ldots, t_n^1) \\ | \quad & C_2(t_1^2, \ldots, t_n^2) \\ | \quad & \vdots \\ | \quad & C_k(t_1^k, \ldots, t_n^k) \end{aligned}$$

denotes a set of constructors for the structural element **structName**, where C_i are constructors and t_j^i are terms over other constructors and constants.

A *list* is an ordered collection or sequence of elements, with the requirement that each element in the list has the same abstract structural type. Definition 4.2.15 presents the inductive definition of a list and defines the *cons* operator, ::, that is used to add an element to a list.

Definition 4.2.15. List Notation

Let α be an abstract structural type. An α list is defined as:

The empty list, denoted by [], is an α `list`.
An element e cons'd to an α `list`, denoted as e**::**[], is an α `list`.

Implicit in this definition is that a list is a homogeneous connection of items. As such, a list of a given kind of items may be written as `item list`.

The **cons** operator may also be used to "deconstruct" a list structure. By the definition above, if L is an α `list`, then L may be replaced by the expression `x::xs`, where `x` is of type α and is the first element of L, and `xs` is an α `list` containing all of the elements of L immediately following the first element. It will also be convenient at times to append the contents of one list onto another, and so I define the `append` function as follows:

Definition 4.2.16. The Append Function for Lists

Let L_1 and L_2 be α `lists`. The function

$$\texttt{append} : (\alpha\,\texttt{list}, \alpha\,\texttt{list}) \rightarrow \alpha\texttt{list}$$

appends the contents of L_2 to L_1, maintaining the order of both lists and placing the first element of L_2 immediately after the last element of L_1, as:

$$\begin{array}{lcl} \texttt{append}(\,L_1,\,[\,]) & \rightarrow & L_1 \\ \texttt{append}(\,[\,],\,L_2) & \rightarrow & L_2 \\ \texttt{append}(\,\texttt{x} :: \texttt{xs},\,[\,]) & \rightarrow & x :: \texttt{append}(\,\texttt{xs},\,L_2) \end{array}$$

The type rules that will be developed in the following text require a determination to be made about an expression, and the determination may be dependent upon the typability of other expressions. Definition 4.2.17 below defines the notation I shall use to express this determination or judgement in the runability type rules that follow.

Definition 4.2.17. Determination Notation

The notation

$$\phi' \vdash d \,:\, t$$

is defined to mean that d, in the context of environment ϕ', is determined to have the runability type t. The $\vdash$ operator may be subscripted to indicate that the type determination has been made with respect to a particular structural classification, i.e., the expression $\phi' \vdash_{HTML} h \,:\, t$ means that h, in the context of environment ϕ', is determined with respect to HTML type rules to have the runability type t.

To denote the dependence of a conclusion upon a set of hypotheses, I shall use a "fractional" notation. The "denominator" will be a conclusion or result to be determined , while the "numerator" will be zero or more hypotheses required to be true for the conclusion to also be true. I formalize this notation in Definition 4.2.18. While the primary use of this notation will be for writing rules for making runability type determinations, I will use it in other situations, and the meaning shall remain the same.

Definition 4.2.18. Hypothesis - Conclusion Notation

Let $\mathcal{S}_{hyp}$ be a set of hypotheses and $Concl$ be a conclusion. The notation

$$\frac{\mathcal{S}_{hyp}}{Concl}$$

is used to denote the implication

$$\bigwedge_{s \in \mathcal{S}_{hyp}} s \quad \Rightarrow Concl$$

4.3 Document Structure and Type Determination

This section will begin the discussion of runability type determination for a Web document. I start with a detailed explanation of the abstract structure that the parsing function of Definition 4.2.3 applies to a document. This leads to a statement of the rule for determining the runability type of a document. The remainder of the section examines the structural components of the abstract document representation and runability type determination for these elements.

The runability type of a document depends upon the `abstractDocument` structure that can be applied to that document. Fundamental to an `abstractDocument` is the idea that a document can be considered a list of elements or `docSeg`s, loosely analogous to paragraphs or sentences in a "traditional" paper document. The parsing function in Definition 4.2.3 builds an `abstractDocument` by decomposing the source document into these elements. Each `docSeg` in the `abstractDocument` may be distinguished as one of three distinct types, based on whether the element represents HTML code, JavaScript code, or neither in the original document. Definition 4.3.1 below summarizes these ideas and and provides the top level of the grammatical definition that will be used to decompose and type documents.

Definition 4.3.1. Abstract Document and Document Segment Structure

```
abstractDocument ::= docSeg list

docSeg           ::= htmlNode(tag, docSeg list, attribute list)
                 |   scriptNode(sourceElt list, langAttr)
                 |   noCodeNode
```

The `htmlNode` element is obviously a recursive structure as it contains a list of `docSeg` elements. The runability type of a `htmlNode` is dependent upon the browser compatibility of the HTML tag itself, the runability type of the contained `docSeg` list, and any browser dependencies introduced by the attributes associated with the particular tag instance. The `scriptNode` element may also have a recursive component, since it possible to have JavaScript statements that contain character strings representing HTML code which influences the runability type determination of the JavaScript component. The runability type of a `scriptNode` is in turn dependent upon the runability determination of the enclosed list of `sourceElt`s, as well as the attribute specifying the scripting language. A `noCodeNode` element contains neither HTML or JavaScript code, and this should be represented by a browser independent runability type. Based on this abstract structure, the first two runability type rules are stated below. Rule 4.3.1 defines the runability type of a document as the runability type determined for the `abstractDocument` applied to the document by the parsing function $\triangleright$. The second rule (4.3.2) defines the runability type of an `abstractDocument` in terms of the runability type determined for the list of `docSeg`s that make up the `abstractDocument`.

Rule 4.3.1.

$$\frac{document \ \triangleright \ absDoc \qquad \phi \vdash_{AD} absDoc : t}{\phi \vdash_{DOC} document : t}$$

Rule 4.3.2.

$$\frac{\phi \vdash_{DSL} dsl : t}{\phi \vdash_{AD} \texttt{abstractDocument}(dsl) : t}$$

To develop a rule for determining the runability type of a `docSeg` list, the relationship between the abstract structure and the actual document must be examined. Observations made from this comparison will lead to a basic assumption that is critical to the runability type analysis of a list of `docSeg`s, and in turn to the analysis of the document itself.

There are two important considerations that need to be made to accurately determine the runability type of a `docSeg` list. First, the de facto standard for rendering HTML documents allows a web browser to ignore any HTML tags it does not know how to render. A document may contain browser-specific tags but still be rendered, incompletely, by other browsers that do not support those particular tags. This applies only to non-script tags, however, since script tags contain JavaScript code that is executed distinctly from the rendering of the document, and this code may not execute without errors on all browsers. Second, a document author may use browser-specific tags in a document and embed JavaScript code to provide alternate content that will be displayed by browsers that cannot correctly render the browser-specific tags. The mere presence of both browser-specific HTML tags and JavaScript code does not necessarily mean that the JavaScript is there to replace the dependent tags. The structural relationship between the JavaScript and the browser-specific tags is critical.

Some examples will serve to illustrate these ideas. In Figure 4.3, the `<ilayer>` tag is only rendered by Netscape version 4.x browsers. The runability type of the `htmlNode` corresponding to the `<ilayer>...</ilayer>` segment to be restricted by this constraint. Furthermore, this node is nested in the `<li>` node, and while the `<li>` node itself is compatible with all browsers, because it contains a restricting HTML element, a particular browser is required to correctly render the node. Suppose, however, that the document's author

```
<ul>
  <li>First list item</li>
  <li><ilayer>
    ⋮
    </ilayer></li>
  <li>Another list item</li>
</ul>
```

Figure 4.3: Browser-specific HTML Code

wishes to provide alternate content renderable by other browsers. This may easily be done by using JavaScript to write out browser-specific HTML code in the same context as the browser-dependent HTML tag. Figure 4.4 illustrates this situation. In this example, the condition of the `if` statement, `IE`, is a user-defined variable set using browser-sniffing code similar to that shown in Figure 4.1, and `printIEListItem`() is a JavaScript function that

prints out IE-specific HTML for a list item. Since both the `ilayer` node and the `script` node share the same logical position in the list, it would seem that the author's intent was to allow both Netscape 4.x and Internet Explorer browsers to correctly render the list item, and by extension, the list as a whole. These examples lead to an important assumption

```
<ul>
  <li>First list item</li>
  <li><ilayer>
    ⋮
    </ilayer>
    <script> if(IE) printIEListItem(); </script></li>
  <li>Another list item</li>
</ul>
```

Figure 4.4: Browser-specific HTML Code with JavaScript

that must be made about the author's intent for particular document constructions. This assumption will then guide the determination of a runability type for a `docSeg` list. As noted earlier, a web browser will ignore a HTML tag it does not know how to render. This behavior allows the content contained in a unrecognized tag to be replaced with other content in a form that the browser can correctly render, as illustrated by the example in Figure 4.4. Thus, a document author may include JavaScript code, enclosed within one or more `<script> ... </script>` tags that are siblings to one or more browser-dependent tags, to allow a document's content to be viewed on additional browsers. A web browser may then ignore the content of any unsupported tags, while generating alternate content using the embedded JavaScript code. For runability type analysis, I make the assumption that a document author understands how to logically relate browser-dependent HTML code with JavaScript code that will allow other browsers to render the document in a manner comparable to the dependent HTML code. This logical relation applies only to HTML and JavaScript code in nodes that are siblings.

Based on this assumption, I begin developing the type rules for lists of HTML nodes by analyzing the cases where the rules will be applied. The first and simplest case is that of an empty `docSeg` list. This is illustrated in Figure 4.3 by the first list item. In this case, the `htmlNode` contains a **li** tag and a`docSeg` list containing only a `noCodeNode`. The node list is empty because there are no additional HTML tags preceding the closing `</li>`

tag in the code. Since an empty list has no elements which may be browser dependent, its runability type will be (β, β), as shown in Rule 4.3.3(a). A second case is shown in Figure 4.5. In this example, the parent node is the `<ul>` node, and the `docSeg` list contains the three `<li>` nodes. These list item nodes are not browser dependent for two reasons. First, the `<li>` tag is rendered correctly by all browsers in the browser set β. Second, each list item node has an empty child node list, so no additional browser dependencies are introduced.

```
<ul>
  <li>First list item</li>
  <li>Second list item</li>
  <li>Third list item</li>
</ul>
```

Figure 4.5: Non-browser-dependent HTML Code

Figure 4.3 illustrates a third case, that of a browser-dependent tag in the `docSeg` list. In this case, the runability type of the nested `docSeg` list in the second **li** tag is limited by the browser dependence of the **ilayer** tag. Figure 4.4 modifies this scenario by adding a **script** tag as a sibling of the **ilayer** tag, with the embedded JavaScript providing Internet Explorer compatible content as an alternative to the content enclosed in the Netscape 4.x dependent **ilayer** tag, based on the assumption stated earlier. In this case, the runability type of the `docSeg` list that is the child of the **li** node should reflect the additional browser compatibility provided by the embedded JavaScript.

```
<ul>
  <li>First list item</li>
  <li><ilayer> ...  </ilayer>
    <script> if(IE) printIEContent(); </script>
    <p> ...  </p>
    <marquee> ...  </marquee>
    <script> if(NS6) printNS6Content();
             else if(NS4) printNS4Content();</script> </li>
  <li>Another list item</li>
</ul>
```

Figure 4.6: Context-dependent, Browser-specific HTML Code with JavaScript

A more complex case is illustrated by the example code in Figure 4.6. In this case, the **ilayer** and **marquee** tags are both browser dependent, and are "paired" with JavaScript code to extend the compatibility. The addition of the **p** tag between the two pairs separates their contexts, since this tag is not browser dependent, and should be rendered correctly by any browser. The runability type in this case should reflect the combined compatibility of the first tag-script pair as well as the second pair, while keeping the type determination of the two pair distinct. This example also identifies the need to distinguish between browser dependent and non-dependent HTML tags when determining the runability type of a `docSeg` list.

```
<ul>
  <li>First list item</li>
  <li><ilayer>
    ⋮
    </ilayer></li>
    <li>    <script> if(IE) printIEListItem(); </script></li>
  <li>Another list item</li>
</ul>
```

Figure 4.7: Browser-specific HTML Code with JavaScript

Before fully developing the type rule for a `docSeg` list, I first define two functions to simplify the construction of this rule. The first is the function `nondepTag`, given in Definition 4.3.2, that takes a `tag` from a `htmlNode` as an argument, and returns a **true** value if the HTML tag represented by the node is not browser dependent and false otherwise. The second function, `splitInitDepNodes`, removes a sequence of browser dependent elements from the beginning of a `docSeg` list, and is defined in Definition 4.3.3.

Definition 4.3.2. Function `nondepTag`

Let t be a `tag` in the definition of the function

$$\texttt{nondepTag} : \texttt{tag} \rightarrow \{\textbf{true}, \textbf{false}\}$$

given below:

$$\texttt{nondepTag}(t) = \begin{cases} \texttt{true} & \text{if} \quad \phi(t) = (\beta, \beta) \\ \texttt{false} & \text{if} \quad \text{otherwise} \end{cases}$$

Definition 4.3.3. Function `splitInitDepNodes`

Let *atl* be a `attribute list`, *la* be a `langAttr`, *ds* be a `docSeg` and *dsl* be a `docSeg` list in the definition of the function

$$\texttt{splitInitDepNodes} : \texttt{docSeg list} \rightarrow \texttt{docSeg list} \times \texttt{docSeg list}$$

that maps a `docSeg` list to an ordered pair of lists. The first list of the pair is an `docSeg` list containing the longest possible sequence of browser dependent nodes at the front of the given list. The second element of the pair is also a `docSeg` list that is the rest of the original list following the browser dependent nodes.

$$\begin{aligned}
&\texttt{splitInitDepNodes}(\texttt{noCodeNode} :: dsl) \quad = \quad ([], \texttt{noCodeNode} :: dsl) \\
&\texttt{splitInitDepNodes}(\texttt{htmlNode}(tag, dsl', atl) :: dsl) \quad = \\
&\qquad \texttt{if } (\texttt{nonDepTag}(tag)) \texttt{ then } ([], \texttt{htmlNode}(tag, dsl', atl) :: dsl) \\
&\qquad \texttt{else let } (init, tail) = \texttt{splitInitDepNodes}(dsl) \\
&\qquad\qquad \texttt{in } (\texttt{htmlNode}(tag, dsl') :: init, tail) \\
&\texttt{splitInitDepNodes}(\texttt{scriptNode}(sel, la) :: dsl) \quad = \\
&\qquad \texttt{let } (init, tail) = \texttt{splitInitDepNodes}(dsl) \\
&\qquad \texttt{in } (\texttt{scriptNode}(sel, la) :: init, tail)
\end{aligned}$$

Based on the assumption that an author will use a script element adjacent to and in parallel with a browser-dependent element to broaden the compatibility of a particular segment of a document, this function allows sequential browser-dependent elements in a `docSeg list` to be typed together. With these functions, it is now possible to state a rule for determining the runability type of a `docSeg` list.

Rule 4.3.3.

Let dsl be a `docSeg`, la be a `langAttr`, and atl be a `attribute list` list in the following.

$$\frac{}{\phi \vdash_{DSL} [] : (\beta, \beta)}$$

$$\frac{\phi \vdash_{DSL} dsl : \Gamma_{dsl}}{\phi \vdash_{DSL} \texttt{noCodeNode} :: dsl : \Gamma_{dsl}}$$

$$\frac{\texttt{nonDepTag}(tag) \qquad \phi_{DS}\texttt{htmlNode}(tag, dsl', atl) :: dsl : \Gamma_1 \qquad \phi \vdash_{DSL} dsl : \Gamma_2}{\phi \vdash_{DSL} \texttt{htmlNode}(tag, dsl', atl) : \Gamma_1 \sqcap \Gamma_2}$$

$$\frac{\begin{array}{c}\texttt{splitInitDepNodes}(\texttt{htmlNode}(tag, dsl', atl) :: dsl) = (init, tail) \\ \neg\texttt{nonDepTag}(tag) \qquad \phi \vdash_{DDSL} init : \Gamma_1 \qquad \phi \vdash_{DSL} tail : \Gamma_2\end{array}}{\phi \vdash_{DSL} \texttt{htmlNode}(tag, dsl', atl) :: dsl : \Gamma_1 \sqcap \Gamma_2}$$

$$\frac{\begin{array}{c}\texttt{splitInitDepNodes}(\texttt{scriptNode}(sel, la) :: dsl) = (init, tail) \\ \phi \vdash_{DDSL} init : \Gamma_1 \qquad \phi \vdash_{DSL} tail : \Gamma_2\end{array}}{\phi \vdash_{DSL} \texttt{scriptNode}(sel, la) :: dsl : \Gamma_1 \sqcap \Gamma_2}$$

The first case states that an empty `docSeg` list is not browser dependent. The second case occurs when a `noCodeNode` is the first element in the list. Since the `noCodeNode` is not browser dependent, the runability type will be that of the remainder of the list. The case where the first element is `htmlNode` with a non-dependent tag name is similar, although the `docSeg` list enclosed within the `htmlNode` may introduce dependencies that must be reflected in the overall determination. In this case, the runability type will be the least upper bound of the enclosed list's type with the remainder of the outer list. If the tag name in a leading `htmlNode` is browser dependent, the sequence of dependent elements in the `docSeg` list is separated and the runability types of the resulting lists are determined separately, with the list containing the dependent elements analyzed using a different procedure which is defined in Rule 4.3.4. The final case occurs when the first element in the list is a `scriptNode`, and is analyzed similar to the previous case since the script may have been embedded to extend the runability of subsequent elements in the list.

Based on the assumption stated earlier, an author will use JavaScript to provide alternate content to replace content within browser dependent HTML tags. To determine the runability type of a list of browser dependent nodes, the greatest lower bound of the nodes is the list is taken. This also requires that an empty list of dependent nodes be compatible with no browsers at all. Rule 4.3.4 below states the rule for determining the runability type of a browser dependent `docSeg` list.

Rule 4.3.4.

Let *dsl* be a `docSeg` list, *hn* be a `htmlNode`, and *sn* be a `scriptNode` in the following.

$$\frac{}{\phi \vdash_{DDSL} [] : (\emptyset, \emptyset)}$$

$$\frac{\phi \vdash_{DS} hn : \Gamma_{hn} \qquad \phi \vdash_{DHNL} dsl : \Gamma_{dsl}}{\phi \vdash_{DDSL} hn :: dsl : \Gamma_{hn} \sqcup \Gamma_{dsl}}$$

$$\frac{\phi \vdash_{DS} sn : \Gamma_{sn} \qquad \phi \vdash_{DHNL} dsl : \Gamma_{dsl}}{\phi \vdash_{DDSL} sn :: dsl : \Gamma_{sn} \sqcup \Gamma_{dsl}}$$

The runability type of a `docSeg` depends first upon the kind of `docSeg` it is. As noted earlier, and implicitly defined in Rule 4.3.3(b), a `noCodeNode docSeg` is not browser dependent, and thus has the runability type (β, β). The runability type of a `htmlNode` is based on three factors: the browser dependency of the node's tag name, as determined by γ_{tag} in Definition 4.2.8, the runability type of the nested `docSeg` list, and any browser dependencies introduced by JavaScript attribute values. As noted in the earlier discussion about `docSeg` lists, the runability type of a parent node is restricted by the runability of the child node list, so the runability type of an `htmlNode` will be the greatest lower bound of the types determined for the tag name, the `attribute` list, and the nested `docSeg` list. Before formalizing the type rule for a `docSeg`, the structure of a HTML tag attribute and the rules to determine an attribute's runability type must be stated, as well as the semantics of the `langAttr` of a `scriptNode`.

Definition 4.3.4. HTML Tag Attribute Structure

```
attribute ::= eventAttribute(attrName, sourceElt list)
          |   urlAttribute(url)
          |   otherAttribute(attrName, value)

url       ::= javascriptURL(sourceElt list)
          |   normalURL()

value     ::= javascriptEntity(sourceElt list)
          |   stringValue()
```

For this analysis, the set of attributes is divided into three disjoint subsets, specified in Definition 4.3.4. The `eventAttribute`s are a set of attribute names associated with events that a browser or the user may trigger, such as moving the mouse over an element, clicking a

mouse button, or pressing a key. The event handling models employed by the Web browsers under consideration share some event attribute names. As there is not a common set of event attributes, the existence of a particular event attribute within a HTML tag may alter the runability type of the tag if that attribute is not universally recognized. Additional browser dependency may be introduced by the JavaScript source element list representing the value of the attribute. Rule 4.3.5 defines the runability type analysis for an event attribute.

Rule 4.3.5.

Let *name* be an event attribute identifier and *sel* be a JavaScript `sourceElt list`in the following.

$$\frac{\phi(name) = \Gamma_{name} \qquad \phi \vdash_{JSEL} sel : \Gamma_{sel}}{\phi \vdash_{ATTR} \texttt{eventAttribute}(name,\ sel) : \Gamma_{name} \sqcap \Gamma_{sel}}$$

A `urlAttribute` is an HTML tag attribute whose value is a URL, such as the `href` attribute used in a `<a>` tag to create a hyperlink to another document or resource. The particular set of URL attribute values of interest to the runability type analysis of a document are those using the `javascript:` *pseudoprotocol*, examples of which are shown in Figure 4.8 below.

```
javascript:5%2
javascript:d = new Date(); typeof d;
javascript:s=""; for(i in document) s+=i+":"+document[i]+"\n"; alert(s);
```

Figure 4.8: Examples of JavaScript Pseudoprotocol URLs

The `javascript:` URL is used when a document author wishes to have JavaScript code executed when a browser loads the URL, i.e., to open a "pop-up" window when an link is selected. The runability type of a `javascriptURL` attribute is dependent upon the type determined for the `sourceElt list` following the `javascript:` pseudoprotocol. All other URL attribute values, either full or partial URLs, are not browser dependent. Rule 4.3.6 summarizes this analysis.

Rule 4.3.6.

Let *url* be a `url` and *sel* be a `sourceElt list` in the following.

$$\frac{}{\phi \vdash_{URL} \texttt{normalURL}() : (\beta, \beta)}$$

$$\frac{\phi \vdash_{JSEL} sel : \Gamma_{sel}}{\phi \vdash_{URL} \texttt{javascriptURL}(sel) : \Gamma_{sel}}$$

$$\frac{\phi \vdash_{URL} url : \Gamma_{URL}}{\phi \vdash_{ATTR} \texttt{urlAttribute}(url) : \Gamma_{url}}$$

The last attribute value that may alter the runability type of a HTML tag is the JavaScript Entity reference, which may be used with any attribute. The JavaScript Entity is supported only by Netscape 4.x browsers; all other browsers will ignore an attribute using this construct as the attribute value. Figure 4.9 below illustrates the usage of a JavaScript Entity as an attribute value.

```
<table border="&{getBorderWidth()};">
<body text=&{favorite_color()};>
```

Figure 4.9: Examples of JavaScript Entity References

Because the JavaScript Entity is only supported on Netscape 4.x browsers, this imposes a constraint on the runability type of the value expression in addition to any limitations induced by the JavaScript code in the Entity. Rule 4.3.7 formally states the analysis rule for all other attribute values which may be JavaScript Entity references.

Rule 4.3.7.

Let *sel* be a `sourceElt list` in the following.

$$\frac{}{\phi \vdash_{ATTR\_VAL} \texttt{stringValue}() : (\beta, \beta)}$$

$$\frac{\phi \vdash_{JSEL} sel : \Gamma_{sel}}{\phi \vdash_{ATTR\_VAL} \texttt{javascriptEntity}(sel) : (\{\text{Netscape4, Netscape4.5}\}, \beta) \sqcap \Gamma_{sel}}$$

$$\frac{\phi \vdash_{ATTR\_VAL} val : \Gamma_{val}}{\phi \vdash_{ATTR} \texttt{otherAttribute}(val) : \Gamma_{val}}$$

The following rule summarizes the runability type analysis for a list of HTML tag attributes, in which an empty list induces no browser dependency, and the runability type of a non-empty list is the greatest lower bound of the types of each attribute in the list.

Rule 4.3.8.

$$\frac{}{\phi \vdash_{ATTR\_LIST} [] : (\beta, \beta)}$$

$$\frac{\phi \vdash_{ATTR} attr : \Gamma_{attr} \qquad \phi \vdash_{ATTR\_LIST} attrl : \Gamma_{attrl}}{\phi \vdash_{ATTR\_LIST} attr :: attrl : \Gamma_{attr} \sqcap \Gamma_{attrl}}$$

As indicated in Definition 4.3.1, a `scriptNode` also contains an attribute that is used to specify the script language encapsualted by the `script` tags. In practice, this value may be associated with either of two different HTML attributes, `type` or `language`. The `type` attribute is the W3C standard [25] for specifying the script language, but all of the browsers under consideration here also support the older, non-standard `language` attribute. While any scripting language might possibly be specified, for this analysis there are only three languages supported and commonly used, as given in Definition 4.3.5. JScript is Microsoft's client-side implementation and extension of ECMAScript, while VBScript is a script-based version of Microsoft Visual Basic that is supported only by the Internet Explorer family of Web browsers.

Definition 4.3.5. Structure of the `language` Attribute for Script Tags

```
langAttr    ::= javascript
            |   jscript
            |   vbscript
```

For the purposes of this analysis, JavaScript and JScript have been treated as the same language, and in most respects they are, since both are based upon and adhere to the ECMAScript language specification. They differ primarily in their document object and event handling models and identifiers, the very differences that provide the basis for this inferential analysis. VBScript has a markedly different syntax and overall language structure, and this runability analysis does not attempt to distinguish between different versions of the language with respect to browser dependence at this time. Instead, if a `script` tag `type` or `language` attribute has a "`vbscript`" value, the entire `scriptNode` is given a runability type indicating a browser dependence strictly upon the four Internet

Explorer browser versions in the set under consideration. No attempt is made to parse and analyze the VBScript source. Rule 4.3.9 formalizes the analysis of the `langAttr` itself, in preparation for the specification of an analysis rule for a `docSeg`.

Rule 4.3.9.

Let ι be the set {MSIE4, MSIE5, MSIE5.5, MSIE6} in the following.

$$\frac{}{\phi \vdash_{LANG\_ATTR} \texttt{javascript} : (\beta, \beta)}$$

$$\frac{}{\phi \vdash_{LANG\_ATTR} \texttt{jscript} : (\beta, \beta)}$$

$$\frac{}{\phi \vdash_{LANG\_ATTR} \texttt{vbscript} : (\iota, \iota)}$$

Returning to the `abstractDocument` structure given in Definition 4.3.1, it is now possible to build the rule for analyzing the runability type of a `docSeg`. As noted earlier in this section, a `noCodeNode` contains no HTML or JavaScript code that can introduce browser dependency, and so will always have a browser-independent runability type. The runability type of an `htmlNode` is influenced by three factors: the browser dependence of the tag name, browser dependencies introduces by attributes of the HTML tag, and the runability type of any children of the `htmlNode`.

An HTML tag may contain a list of one or more attributes. If the list is present, it may affect the runability analysis of the tag structure as a whole. Since any behaviors or rendering characteristics dynamically introduced by a JavaScript-valued attribute are integral to the particular instance of the tag, they cannot be compensated for by parallel HTML or JavaScript code. Figure 4.10 illustrates such a situation. In this example, a mouseover event is associated with a hyperlink, so that when a user moves the mouse pointer over the link, the `displayCat1Menu` function is invoked to display an expanded list of links to specific areas within or accessible from the `category1.html` target.

```
<a href="category1.html" onmouseover="displayCat1Menu()">
```

Figure 4.10: Example of Browser-dependent Attribute Value

The `onmouseover` attribute is not supported by Netscape 4.x browsers, and a sibling to this tag cannot add comparable behavior for the unsupported browsers, in the manner previously described in the discussion of document segments, above. A cross-browser

solution in this case would be to generate the tag as a scripted component, using JavaScript events to trigger the desired behavior, based on the detected browser, instead of relying on HTML events. Thus the runability type of a `htmlNode` is then the greatest upper bound of the runability types of the tag name, the attribute list, and the child `docSeg` list. Finally, as discussed above, the runability type of a `scriptNode` with a `langAttr` of `javascript` or `jscript` is dependent upon the runability type of the `sourceElt list` contained in the node, while a `langAttr` of `vbscript` restricts the runability of the `scriptNode` strictly to the set of Internet Explorer browsers. Rule 4.3.10 formalizes the runability analysis for a `docSeg`.

Rule 4.3.10.

Let *tag* be a HTML tag name, *dsl* be a `docSeg list`, *atl* be a `attribute list`, *sel* be a `sourceElt list`, *la* be a `langAttr`, and ι be the browser set {MSIE4, MSIE5, MSIE5.5, MSIE6} in the following.

$$\frac{}{\phi \vdash_{DS} \texttt{noCodeNode} : (\beta, \beta)}$$

$$\frac{\gamma(tag) = \Gamma_{tag} \quad \phi \vdash_{ATTR\_LIST} atl : \Gamma_{atl} \quad \phi \vdash_{DSL} dsl : \Gamma_{dsl}}{\phi \vdash_{DS} \texttt{htmlNode}(tag, dsl, atl) : \Gamma_{tag} \sqcap \Gamma_{atl} \sqcap \Gamma_{dsl}}$$

$$\frac{la = \texttt{javascript} \wedge la = jscript \quad \phi \vdash_{JSEL} sel : \Gamma_{sel}}{\phi \vdash_{DS} \texttt{scriptNode}(sel, la) : \Gamma_{sel}}$$

$$\frac{la = \texttt{vbscript}}{\phi \vdash_{DS} \texttt{scriptNode}(sel, la) : (\iota, \iota)}$$

The rules presented in this section provide the means to analyze the abstract structure of a Web document to determine which web browsers will correctly render the document. The next section will cover the rules for determining the runability type of the JavaScript part of a document, extending the third and final case of Rule 4.3.10.

4.4 JavaScript Runability Analysis

In this section, I will develop the rules for determining the runability type of JavaScript source code. In the preceding discussion, I have touched upon the two occasions when JavaScript code may be encountered with respect to runability type analysis. The first is as the *document* itself, i.e. the URL identifying the resource to be analyzed references

a JavaScript program. The second is within `scriptNode`s embedded in an `docSeg` list representing a document. The rules defined here apply transparently to both situations.

To decompose a JavaScript program, I shall use a simplified version of the grammar defined in the ECMAScript version 3 specification[2] of the language. ECMAScript is the "core" of the JavaScript language, and the specification defines the syntactic and semantic construction of a JavaScript program, including a set of objects and functions that should be included in an implementation meeting the standard's specification. It should be noted that this specification is considered a baseline, and that implementors may choose to extend that capabilities of the language. This is most commonly done through the inclusion of additional objects and functions in a particular implementation. These extensions, coupled with different levels of implementation of the ECMAScript standard, give rise to the browser compatibility problems this analysis seeks to identify.

4.4.1 Source Elements

As noted in Definition 4.2.1, a `scriptNode` contains a list of JavaScript source elements. A `sourceElt` may be either an executable statement or a function declaration, formally stated in Definition 4.4.1.

Definition 4.4.1. JavaScript `sourceElt` Structure

```
sourceElt      ::=  funcDecl(identifier, optFormalParams, body)
                |   statement
```

Statements are executed when they are encountered by the JavaScript interpreter. Function declarations are not executed at the time of definition and establish their own execution context at the time the function is called. It should also be noted that the existence of a function definition does not imply that there is an execution path that will actually execute the function. Because of this, the runability type of a function declaration does not affect the runability type of a sequence of source elements, but must still be determined at the point of declaration. The runability type of the function declaration is determined and bound to the function's name identifier in the environment ϕ to be used for type determination of a function call at another location in the code.

The ECMAScript specification [2] requires the scope chain of a newly entered execution context to be modified with the identifier names of formal function arguments (if

the new context is that of a function), then with the function identifier name, and finally with any variable identifiers in the list of source elements corresponding to the execution context. To correctly analyze the runability type of a list of JavaScript source elements, these the runability context *phi* should also be augmented in the same manner. When a function declaration is processed, including the formal parameters to the function, the runability type of the function body is also determined, analogous to the construction of a `Function` object during the evaluation of a JavaScript program. When a function is called, a new execution context is created, using the context defined by the function declaration and the context existing at the call point.

Since JavaScript support global variables, it is possible that a function declaration may contain references to variables that have not been declared before the function definition. These global variables may influence the runability type of the function at the call point, however, and must also be accounted for. I define a function $\texttt{undeclIds} : \Phi \times \texttt{sourceElt} \rightarrow \texttt{identifier list}$ that "extracts" any undeclared identifiers from a source element and returns these identifiers in a list.

Definition 4.4.2. Function `undeclIds`

The function

$$\texttt{undeclIds} : \Phi \times \texttt{sourceElt} \rightarrow \texttt{identifier list}$$

defines a mapping from an environment and a JavaScript source element to a list of identifiers that have not been declared in the environment. In the definition below, let fd be a function declaration, *stmt* be a statement, *id* be an identifier, and *ids* be an identifier list.

$$\begin{array}{lcl}
\texttt{undeclIds}(\phi,\ fd) & = & [] \\
\texttt{undeclIds}(\phi,\ stmt) & = & \texttt{let}\ ids = []\ \texttt{in} \\
 & & \quad \texttt{foreach}\ id\ \texttt{in}\ stmt\ \texttt{do} \\
 & & \quad\quad \texttt{if}\ (\phi(id) = \texttt{null})\ \texttt{then}\ ids = id :: ids \\
 & & \texttt{return}\ ids
\end{array}$$

This definition can then be extended to a `sourceElt list` as defined below, where *se* is a `sourceElt` and *sel* is a `sourceElt list`.

$$\begin{array}{lcl}
\texttt{undeclIds}(\phi,\ [],\ ids) & = & ids \\
\texttt{undeclIds}(\phi,\ se :: sel,\ ids) & = & \texttt{undeclIds}(\phi,\ sel,\ \texttt{undeclIds}(\phi,\ se))
\end{array}$$

At the point where a function is called, it will be necessary to retrieve this list of identifiers to determine their runability types in the context of the call point, and from that determine the runability type of the function call expression. The function κ : `identifier` $\rightarrow$ `identifier list` defined below describes this mapping.

Definition 4.4.3. The *kappa* (κ) Mapping

> Let κ : `identifier` $\rightarrow$ `identifier list` define a mapping between a function name `identifier` and a list of undeclared `identifier`s referenced in the body of the function. This mapping is implicit in the environment when the environment is augmented with the function `identifier`.

I now define a function Λ : $\Phi \times$ `sourceElt` $\rightarrow \Phi$ that traverses a list of JavaScript source elements, augmenting the environment ϕ with the identifier closure and runability type of each function declaration encountered during the traversal.

Definition 4.4.4. The *Lambda* (Λ) Function

> The function Λ : $\Phi \times$ `sourceElt` $\rightarrow \Phi$ defines a mapping from an environment and a JavaScript source element to a new environment that has been augmented with the function identifier closure and runability type if the source element is a function declaration. The semantics of the Λ function are as follows:
>
> $$\frac{}{\Lambda(\phi, \texttt{statement}) = \phi}$$
>
> $$\frac{\phi \vdash_{JFD} \texttt{funcDecl}(id, params, body) : \Gamma' \qquad \texttt{undeclIds}(\phi, body, []) = ids \qquad \phi[id/\Gamma'] = \phi'}{\Lambda(\phi, \texttt{funcDecl}(id, params, body)) = \phi'}$$
>
> The behavior of the Λ function is easily extensible to a list of source elements. This behavior is defined below, where *se* and *sel* are a `sourceElt` and a `sourceElt` list, respcetively.
>
> $$\frac{}{\Lambda(\phi, []) = \phi}$$
>
> $$\frac{\Lambda(\phi, se) = \phi' \qquad \Lambda(\phi', sel) = \phi''}{\Lambda(\phi, se :: sel) = \phi''}$$

Statements are the executable units in a JavaScript program. Because they will be executed sequentially in a list of source elements, all statements in the list must be runable on the same browser. As mentioned earlier, it is possible to write statements that imply dependence on a particular browser, but will execute correctly on others. This is the basis

for "browser sniffer" code segments, and should be incorporated into web documents that are intended to be rendered on multiple browsers. It is also possible to write statements that will only execute on a particular browser, and will cause an error condition on others. An example of this is the `try` statement, which will halt JavaScript execution completely on Internet Explorer 4.0, cause Netscape 4.x and Internet Explorer 5.0 browsers to report an error, but executes without errors on Netscape 6 and Internet Explorer 5.5 and 6.0 browsers. The development of our type rules reflects the need to accommodate both of these situations and illustrates the need to remember two sets of browsers in the runability type.

There are two cases to consider when merging a statement with a source element list. The first case is where the statement, the list, or both is not browser dependent, i.e. $\delta = \beta$ for the statement, list, or both. In the other case, both the statement and the list are browser dependent. Before stating the rules for determining the runability type of a source element list, I develop the logic for the rules and show their correctness.

There are three possibilities for the first case. First, both the statement and the list are not browser dependent. Then the δ of both will be β, and since $\delta \subseteq \eta$, the η of both will also be β. Second, the list is not browser dependent, but the statement is. In this situation, the statement limits the runability of the list. By definition, $\delta_{stmt} \subseteq \eta_{stmt} \subseteq \beta$. So we know that the combined statement and list will execute without errors in the same context as the list itself, but indicates a "preference" for a browser in δ_{stmt}. The resulting runability type will be the runability of the statement. The third possibility is when the list is browser dependent and the statement is not, and using the same logic as was used for the second possibility, the resulting runability type will be that of the list.

In the second case, both the statement and the list are browser dependent. There are two situations to consider - whether or not the dependency set of one is compatible with the run set of the other (and vice versa). More formally, let $\phi \vdash_{STMT} stmt : (\delta_1, \eta_1)$ and $\phi \vdash_{JSEL} list : (\delta_2, \eta_2)$. If $\delta_1 \cap \eta_2 \neq \emptyset$ and $\delta_2 \cap \eta_1 \neq \emptyset$, the statement and list are compatible as they will both execute without errors on the same browser, as long as that browser is in the run set of both the statement and the list, $\eta_1 \cap \eta_2$. Since $\delta_1 \cap \eta_2 \neq \emptyset$ and $\delta_2 \cap \eta_1 \neq \emptyset$, the merged statement and list will have $\delta = (\delta_1 \cap \eta_2) \cup (\delta_2 \cap \eta_1)$, allowing the author to write code that can execute without error on multiple browsers. The second situation occurs when either $\delta_1 \cap \eta_2 = \emptyset$ or $\delta_2 \cap \eta_1 = \emptyset$. Here there are no common dependent browsers between the statement and the list, so the resulting δ will be the empty set.

Combining the results of this analysis with the definition of the Λ function, a rule for determining the runability type of a list of `sourceElt`s may now be formally stated.

Rule 4.4.1.

Let fd be a function declaration, *stmt* be a statement, and *sel* be a source element list in the following:

$$\frac{}{\phi \vdash_{JSEL} [] : \Gamma_\phi}$$

$$\frac{\Lambda(\phi,\, fd) = \phi' \qquad \phi' \vdash_{JSEL} sel : \Gamma'}{\phi \vdash_{JSEL} fd :: sel : \Gamma'}$$

$$\frac{\begin{array}{cc}\Lambda(\phi,\, sel) = \phi' & \phi' \vdash_{STMT} stmt : \Gamma_{stmt} \\ \phi' \vdash_{JSEL} sel : \Gamma_{sel} & (\beta \subseteq \delta_{stmt} \vee \beta \subseteq \delta_{sel})\end{array}}{\phi \vdash_{JSEL} stmt :: sel : \Gamma_{stmt} \sqcap \Gamma_{sel}}$$

$$\frac{\begin{array}{ccc}\Lambda(\phi,\, sel) = \phi' & & \phi' \vdash_{STMT} stmt : (\delta_1,\, \eta_1) \\ \phi' \vdash_{JSEL} sel : (\delta_2,\, \eta_2) & \delta_1 \cap \eta_2 \neq \emptyset & \delta_2 \cap \eta_1 \neq \emptyset\end{array}}{\phi \vdash_{JSEL} stmt :: sel : ((\delta_1 \cap \eta_2) \cup (\delta_2 \cap \eta_1),\, (\delta_1 \cap \eta_2) \cup (\delta_2 \cap \eta_1))}$$

$$\frac{\begin{array}{cc}\Lambda(\phi,\, sel) = \phi' & \phi' \vdash_{STMT} stmt : (\delta_1,\, \eta_1) \\ \phi' \vdash_{JSEL} sel : (\delta_2,\, \eta_2) & \delta_1 \cap \eta_2 = \emptyset \vee \delta_2 \cap \eta_1 = \emptyset\end{array}}{\phi \vdash_{JSEL} stmt :: sel : (\emptyset,\, \eta_1 \cap \eta_2)}$$

4.4.2 Function Declarations

Before considering how to determine the runability type of a function declaration, I complete the grammatical definition of a function declaration started in Definition 4.4.1.

Definition 4.4.5. The Function Declaration Structure

Restating part of Definition 4.4.1 for continuity, followed by the full definition of a function declaration.

```
sourceElt       ::=  funcDecl(identifier, optFormalParams, body)

optFormalParams ::=  identifier list

body            ::=  sourceElt list
```

Runability type analysis of a function declaration is done at the point of declaration by analyzing the source elements that make up the body of the function. Because JavaScript is a dynamically typed language, the formal parameters to the function are not associated with type information. At first glance, this might seem to be an argument to postpone our analysis until the function is called. This is not necessary, however, since we can determine a runability type for the source elements in the function body, and any manipulation of the identifiers associated with the formal parameters that restricts the runability type of a parameter will contribute to the runability type of the body. At the site of a function call, the composite runability type of the actual arguments to the function will be combined with the runability type of the function. The parameter identifiers must still be accounted for in the analysis, however. This is accomplished by augmenting the environment used in the analysis of the function body with the identifiers in the formal parameter list, if it is present. These identifiers are initially bound to a runability type of (β, β).

Before summarizing the runability type rule for a function declaration, I define an environment augmentation for adding a list of identifiers with the same runability type to an environment. This definition uses the basic augmentation in Definition 4.2.12. Note that an empty list does not change the state of the environment.

Definition 4.4.6. Augmentation of the Environment with a List of Identifiers

$$\frac{}{\phi[\,[]\,/\,\Gamma] = \phi}$$

$$\frac{\phi[id\,/\,\Gamma] = \phi_{id} \qquad \phi_{id}[idList\,/\,\Gamma] = \phi'}{\phi[id :: idList\,/\,\Gamma] = \phi'}$$

Using this definition, a rule for augmenting an environment with the formal parameters of a function declaration can be written as follows:

Rule 4.4.2.

$$\frac{}{\phi[\texttt{optFormalParams}([])] = \phi}$$

$$\frac{\phi[id :: idList\,/\,(\beta,\,\beta)] = \phi'}{\phi[\texttt{optFormalParams}(id :: idl)] = \phi'}$$

The runability type determination for a function declaration requires that the identifiers in the formal parameter list be added to the environment before processing the

body of the function. The body is a `sourceEltList`, and determination of its runability type is done using Rule 4.4.1, defined above.

Rule 4.4.3.

$$\frac{\phi[ofp] = \phi' \qquad \phi' \vdash_{JSEL} sel : t}{\phi \vdash_{JFD} \texttt{funcDecl}(id,\, ofp,\, \texttt{body}(sel)) : t}$$

Statements

Statements are the basic unit of execution in a JavaScript program. Runability type determination is based upon the kind of statement under consideration and its underlying structure. Before developing rules for these various kinds of statements, they must first be identified. Below I present a simplified grammar that I will use to decompose a statement as a basis for the development of runability type rules. More detailed grammars for each kind of statement will be presented at the point where the runability type rule is developed.

Definition 4.4.7. JavaScript Statement Structure

```
statement  ::=  blockStmt(statement list)
            |   variableStmt(varDecl list)
            |   emptyStmt()
            |   expressionStmt(expression)
            |   ifStmt(expression, statement)
            |   ifStmt(expression, statement, statement)
            |   do-whileStmt(statement, expression)
            |   whileStmt(expression, statement)
            |   forStmt(optExpr, optExpr, optExpr, statement)
            |   forStmt(varDecl list, optExpr, optExpr, statement)
            |   for-inStmt(expression, expression, statement)
            |   for-inStmt(varDecl, expression, statement)
            |   continueStmt()
            |   breakStmt()
            |   returnStmt()
            |   returnStmt(expression)
            |   withStmt(expression, statement)
            |   switchStmt(expression, caseBlock)
            |   labelledStmt(statement)
            |   throwStmt(expression)
            |   tryStmt(blockStmt, catch, finally)
```

Block Statement

A block statement provides the means to group a sequence of zero or more statements into a single statement. All of the statements in such a sequence must be able to run without errors on a common browser set, just as the statements in a source element list must also execute in a common browser set. The rules for determining the runability type of a statement list are very similar to those for a source element list, and follow the same reasoning. Definition 4.4.8 below states the structure of a block statement in advance of developing the runability determination rule.

Definition 4.4.8. Block Statement

```
statement      ::=  blockStmt(statement list)
```

Since a block statement is just a "wrapper" for a statement list, the runability type of a block statement is exactly the composite runability type of the statement list.

Rule 4.4.4.

$$\frac{\phi \vdash_{STMT-LIST} sl : \Gamma'}{\phi \vdash_{STMT} \texttt{blockStmt}(sl) : \Gamma'}$$

The runability type of a statement list is determined in much the same manner as a source element list, without the need to distinguish function declarations from statements. Additionally, the runability type of an empty statement list is defined to be the runability type of the environment ϕ.

Rule 4.4.5.

Let *stmt* be a `statement` and *sl* be a `statement` list in the following.

$$\frac{}{\phi \vdash_{STMT-LIST} [] : \Gamma_\phi}$$

$$\frac{\phi \vdash_{STMT} stmt : \Gamma_{stmt} \qquad \phi \vdash_{STMT-LIST} sl : \Gamma_{sl} \qquad (\beta \subseteq \delta_{stmt} \vee \beta \subseteq \delta_{sl})}{\phi \vdash_{STMT-LIST} stmt :: sl : \Gamma_{stmt} \sqcap \Gamma_{sl}}$$

$$\frac{\begin{array}{cc}\phi \vdash_{STMT} stmt : (\delta_1,\, \eta_1) & \phi \vdash_{STMT-LIST} sl : (\delta_2,\, \eta_2) \\ (\delta_1 \cap \eta_2) \neq \emptyset & (\delta_2 \cap \eta_1) \neq \emptyset\end{array}}{\phi \vdash_{STMT-LIST} stmt :: sl : (\,(\delta_1 \cap \eta_2) \cup (\delta_2 \cap \eta_1),\ (\delta_1 \cap \eta_2) \cup (\delta_2 \cap \eta_1))}$$

$$\frac{\phi \vdash_{STMT} stmt : (\delta_1,\, \eta_1) \qquad \phi \vdash_{STMT-LIST} sl : (\delta_2,\, \eta_2) \qquad (\delta_1 \cap \eta_2 = \emptyset \vee \delta_2 \cap \eta_1)}{\phi \vdash_{STMT-LIST} stmt :: sl : (\emptyset,\, \eta_1 \cap \eta_2)}$$

Variable Statement

A variable statement is a statement that is used to declare a new variable or a list of variables, and to optionally initialize these variables with a value. An uninitialized variable has an JavaScript type of *undefined*, and in terms of this analysis, is not dependent upon any particular browser implementation for execution. A variable that has been initialized will incorporate the runability type of the initializing expression. As with function names, a variable identifier must be added to the environment when it is declared. The abstract structure for a variable statement used in this analysis is defined below.

Definition 4.4.9. Variable Statement

```
statement      ::=  variableStmt(varDecl list)

varDecl        ::=  identifier
                |   identifier assignExpr
```

As defined above, a `varDecl` list is a list of variable declarations, each of which consists of an identifier optionally paired with an initializing assignment expression. Because all declarations in the list will be executed in the same context, all initializing expressions must be able to be evaluated on the same set of browsers.

Rule 4.4.6.

$$\frac{\phi \vdash_{VDL} vdList : t}{\phi \vdash_{STMT} \texttt{variableStmt}(vdList) : t}$$

Rule 4.4.7.

Let vd be a `varDecl` and vdl be a `varDecl` list in the following.

$$\frac{}{\phi \vdash_{VDL} [] : \Gamma_\phi}$$

$$\frac{\phi \vdash_{VAR-DECL} vd : \Gamma_{vd} \qquad \phi \vdash_{VDL} vdl : \Gamma_{vdl}}{\phi \vdash_{VDL} vd :: vdList : \Gamma_{vd} \sqcap \Gamma_{vdl}}$$

An individual variable declaration may or may not be initialized by the value of an assignment expression. In either case, the variable name (identifier) is added to the environment. If the expression is present, the runability type bound to the identifier in the environment will be the type determined for the expression. If the expression is not present, the variable is assumed to be browser independent. If a later assignment to the variable

introduces a browser dependency, the variable will be bound to that restricted type at that time.

Rule 4.4.8.

$$\frac{\phi[id \diagup (\beta,\, \beta)]}{\phi \vdash_{VAR-DECL} \texttt{varDecl}(id) : (\beta,\, \beta)}$$

$$\frac{\phi \vdash_{EXPR} expr : t \qquad \phi[id \diagup t]}{\phi \vdash_{VAR-DECL} \texttt{varDecl}(id, expr) : t}$$

Empty Statement

An empty statement is exactly what the name states — a statement with no executable source code. The empty statement is browser independent.

Rule 4.4.9.

$$\frac{}{\phi \vdash_{STMT} \texttt{emptyStmt}() : (\beta, \beta)}$$

Expression Statement

An expression statement provides the means, in JavaScript, to build a statement from an expression, as defined below. The runability type of an expression statement is the runability type of the embedded expression.

Definition 4.4.10. Expression Statement

```
statement   ::=  expressionStmt(expression)
```

Rule 4.4.10.

$$\frac{\phi \vdash_{EXPR} expr : t}{\phi \vdash_{STMT} \texttt{expressionStmt}(expr) : t}$$

`if` Statement

`If` statements allow computation to branch based on the value of a boolean branching expression. An `if` statement may also contain an "else" clause with a statement that is executed if the branch expression evaluates to false value. To accurately analyze an `if` statement, both cases must be considered, as well as whether or not the condition expression is browser dependent or not. In particular, if the condition is browser dependent and the statement has an `else` clause, the resulting runability type of the `if` statement must

reflect both the dependency of the expression and the complement of that dependency. As usual, my analysis begins with a definition of the abstract structure of an `if` statement.

Definition 4.4.11. If Statement

```
statement      ::=  ifStmt(expression, statement)
                |   ifStmt(expression, statement, statement)
```

There are four cases to consider in the determination of the runability type of an `if` statement. In the first case, the condition expression is browser independent, and there is no `else` clause. In this case, the resulting runability type is determined by the type of the statement that will be executed if the expression evaluates to true. The second case also includes a browser-independent conditional expression, and also has an `else` clause containing a second statement. In this case, I assume that the author is not attempting to introduce behavior that might be browser dependent, and that a browser executing this `if` statement should be able to correctly execute either substatement. In the third and fourth cases the condition expression is browser dependent. The third case has no `else` clause, and the assumption is that this is a browser test, i.e., the substatement represents an execution path to be taken only if the program is executing on a particular set of browsers. The expression and the substatement should thus share a common browser dependence. The fourth and final case has both an `if` statement and an `else` statement with a browser-dependent condition expression. As in the third case, the assumption is that this is a browser test. The reasoning used for the third case also applies to this case, but the implications of the `else` clause must also be considered. The first statement represents the execution path taken if the program is running on one of the set of browsers the expression is dependent upon. However, if one of those browsers is not detected, the `else` statement is executed, implying that this statement is the execution path for the set of browsers that is the complement of the browser set of the expression.

Rule 4.4.11.

$$\frac{\phi \vdash_{EXPR} expr : (\beta,\, \beta) \qquad \phi \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{ifStmt}(expr,\, stmt) : \Gamma_{stmt}}$$

$$\frac{\phi \vdash_{EXPR} expr : (\beta,\, \beta) \quad \phi \vdash_{STMT} stmt_1 : \Gamma_1 \quad \phi \vdash_{STMT} stmt_2 : \Gamma_2}{\phi \vdash_{STMT} \texttt{ifStmt}(expr,\, stmt_1,\, stmt_2) : \Gamma_1 \sqcup \Gamma_2}$$

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \neq (\beta,\, \beta) \qquad \phi \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{ifStmt}(expr,\, stmt) : \Gamma_{expr} \sqcap \Gamma_{stmt}}$$

$$\frac{\phi \vdash_{EXPR} expr : (\delta_{expr}, \eta_{expr}) \neq (\beta,\, \beta) \quad \phi \vdash_{STMT} stmt_1 : (\delta_1, \eta_1) \quad \phi \vdash_{STMT} stmt_2 : (\delta_2, \eta_2)}{\phi \vdash_{STMT} \texttt{ifStmt}(expr,\, stmt_1,\, stmt_2) : ((\delta_{expr} \cap \delta_1) \cup (\overline{\delta}_{expr} \cap \delta_2),\ (\eta_{expr} \cap (\eta_1 \cup \eta_2)))}$$

`do-while` and `while` Statements

The `do-while` iterative statement will always execute the loop statement once, looping only if the condition expression is true. The `while` statement evaluates the condition expression and will execute the loop statement if it is true. With respect to runability type determination, the two statements are the same. The analysis of `if` statements without an `else` also applies to the `do-while` and `while` statements, as both have a condition expression and a single statement, as defined below.

Definition 4.4.12. Iterative Statement

```
statement      ::=  do-whileStmt(statement, expression)

statement      ::=  whileStmt(expression, statement)
```

Rule 4.4.12.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{do} - \texttt{whileStmt}(stmt,\, expr) : \Gamma_{expr} \sqcap \Gamma_{stmt}}$$

Rule 4.4.13.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{whileStmt}(expr,\, stmt) : \Gamma_{expr} \sqcap \Gamma_{stmt}}$$

`for` Statement

The `for` statement is similar to the `while` statement, except that it contains three optional expressions, the first of which may be a variable declaration list, and the second is the actual loop condition, in addition to the loop statement. Also similar to the `while` statement, the runability of a `for` statement is the common set of elements between the expressions and the statement.

Definition 4.4.13. For Statement

```
statement  ::= forStmt(optExpr, optExpr, optExpr, statement)
           |   forStmt(varDecl list, optExpr, optExpr, statement)

optExpr    ::= ()
           |   expression
```

To simplify the rule for a `for` statement, I first state a rule for determining the runability type of an optional expression. As the term implies, an optional expression is an expression that may be omitted from the source code. An optional expression that is not present is compatible with all browsers, since there is no code to introduce a dependency.

Rule 4.4.14.

$$\frac{}{\phi \vdash_{OPT-EXPR} \texttt{optExpr}() : (\beta, \beta)}$$

$$\frac{\phi \vdash_{EXPR} expr : \Gamma'}{\phi \vdash_{OPT-EXPR} \texttt{optExpr}(expr) : \Gamma'}$$

Execution of the embedded statement is dependent upon successful evaluation of the three optional expressions, or the variable declaration list and two expressions. Therefore, the expressions (and the variable declaration list, if present) must all execute on a common set of browsers. As with the `while` statement, the expressions must also be common with the embedded statement. If the variable declaration list is present, the declared variables augment the environment that the statement and the other expressions are evaluated in, and the rule below reflects that augmentation. In the second part of the rule below, I use ϕ' to denote the state of the environment after typing the variable declaration list, which also augments the environment as stated in Rule 4.4.7.

Rule 4.4.15.

$$\frac{\begin{array}{cc}\phi \vdash_{EXPR} expr_1 : \Gamma_1 & \phi \vdash_{EXPR} expr_2 : \Gamma_2 \\ \phi \vdash_{EXPR} expr_3 : \Gamma_3 & \phi \vdash_{STMT} stmt : \Gamma_{stmt}\end{array}}{\phi \vdash_{STMT} \texttt{forStmt}(expr_1, expr_2, expr_3, stmt) : \Gamma_1 \sqcap \Gamma_2 \sqcap \Gamma_3 \sqcap \Gamma_{stmt}}$$

$$\frac{\begin{array}{cc}\phi \vdash_{VDL} vdl : \Gamma_{vdl} & \phi' \vdash_{EXPR} expr_2 : \Gamma_2 \\ \phi' \vdash_{EXPR} expr_3 : \Gamma_3 & \phi' \vdash_{STMT} stmt : \Gamma_{stmt}\end{array}}{\phi \vdash_{STMT} \texttt{forStmt}(vdl, expr_2, expr_3, stmt) : \Gamma_{vdl} \sqcap \Gamma_2 \sqcap \Gamma_3 \sqcap \Gamma_{stmt}}$$

`for-in` Statement

The `for-in` statement provides a simple mechanism to iterate through the enumerable properties of an object, performing some computation on each one. The abstract structure is defined in Definition 4.4.14 below. The first expression may be either a variable declaration or an expression identifying a particular enumerable property of the object value of the second expression. Determination of the runability type is much the same as the `for` statement, except that the expressions are not optional.

Definition 4.4.14. For-in Statement

```
statement    ::=  for-inStmt(expression, expression, statement)
              |   for-inStmt(varDecl, expression, statement)
```

Browser dependency analysis is basically the same as that for the `for` statement.

Rule 4.4.16.

$$\frac{\phi \vdash_{EXPR} expr_1 : \Gamma_1 \qquad \phi \vdash_{EXPR} expr_2 : \Gamma_2 \qquad \phi \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{for-inStmt}(expr_1, expr_2, stmt) : \Gamma_1 \sqcap \Gamma_2 \sqcap \Gamma_{stmt}}$$

$$\frac{\phi \vdash_{VAR-DECL} vd : \Gamma_{vdl} \qquad \phi' \vdash_{EXPR} expr : \Gamma_{expr} \quad \phi' \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{forStmt}(vd, expr, stmt) : \Gamma_{vdl} \sqcap \Gamma_{expr} \sqcap \Gamma_{stmt}}$$

`continue` Statement

The `continue` statement is used only within iteration statements to control the flow of execution. It has no executable components, and is compatible with all browsers.

Rule 4.4.17.

$$\frac{}{\phi \vdash_{STMT} \texttt{continueStmt} : (\beta, \beta)}$$

`break` Statement

The `break` statement is used only within iteration or `switch` statements to control execution flow, and like the `continue` statement, has no executable content.

Rule 4.4.18.

$$\frac{}{\phi \vdash_{STMT} \texttt{breakStmt} : (\beta, \beta)}$$

`return` Statement

The `return` statement is only used within the body of a function, and includes an optional expression whose value is that which is returned by the function. The abstract structure of a `return` statement is given in Definition 4.4.15 below.

Definition 4.4.15. Return Statement

```
statement    ::= returnStmt()
             |   returnStmt(expression)
```

The runability type of the `return` statement will be dependent upon the runability of the current environment ϕ if there is no expression value to be returned. Since the `return` statement may only be used within a function body, this environment will be that of the function body itself. If the expression is present, the runability type of the statement will be that of the expression, and because that runability is determined within the current environment, it will include any dependencies that exist in that environment.

Rule 4.4.19.

$$\frac{}{\phi \vdash_{STMT} \texttt{returnStmt}() : \Gamma_{\phi}}$$

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr}}{\phi \vdash_{STMT} \texttt{returnStmt}(expr) : \Gamma_{expr}}$$

`with` Statement

The `with` statement, whose structure is defined in Definition 4.4.16 below, adds the computed result of an expression to the current execution context, executes a statement within the augmented context, then removes the result of computing the expression from the context before passing control to the next statement. The runability of the expression and the statement must be compatible, and the runability of the `with` statement will be the largest set of browsers compatible with both.

Definition 4.4.16. With Statement

```
statement     ::=  withStmt(expression, statement)
```

Rule 4.4.20.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi \vdash_{STMT} stmt : \Gamma_{stmt}}{\phi \vdash_{STMT} \texttt{withStmt}(expr, stmt) : \Gamma_{expr} \sqcap \Gamma_{stmt}}$$

`switch` Statement

A `switch` statement is used to control the flow of execution based on a switch expression compared to the values of one or more `case` expressions. The `switch` statement itself is made up of an expression and a block of zero or more `case` clauses, which may include a `default` clause, as the abstract structure in Definition 4.4.17 illustrates. Each `case` clause has an expression that is evaluated and compared to the switch expression, and an optional statement list that is executed if the case expression and the switch expression are equivalent. The `default` clause does not have an expression, and its optional statement list is executed if none of the preceding case expressions (in source text order) have tested equivalent to the switch expression value.

Definition 4.4.17. Switch Statement

```
statement      ::=  switchStmt(expression, caseBlock)

caseBlock      ::=  optCaseClauses optDefaultClause optCaseClauses

optCaseClauses ::=
               |    caseClause list
```

```
caseClause     ::=  expression optStatementList

optDefaultClause ::= ()
                |    defaultClause

defaultClause  ::=  optStatementList

optStatementList ::= ()
                |    statement list
```

The runability type of a `switch` statement will be determined by the types of the switch `expression` and the `caseBlock`. If the `expression` is not browser dependent, the runability type of the `switch` statement will be that of the `caseBlock`. If the `expression` is browser dependent, and the `caseBlock` is not, the statement will have the same dependency as the `expression`. Finally, if both the `expression` and the `caseBlock` are browser dependent, the runability type of the `switch` statement must then reflect the set of browsers compatible with both of these elements. These three cases can be combined in one rule, where the runability type of the `switch` statement is the intersection of the runability types of the `expression` and the `caseBlock`, and is formally stated in Rule 4.4.17.

Definition 4.4.18.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi \vdash_{CASEBLK} cb : \Gamma_{cb}}{\phi \vdash_{STMT} \texttt{switchStmt}(expr, cb) : \Gamma_{expr} \sqcap \Gamma_{cb}}$$

A `caseBlock` is a collection of zero or more `caseClause`s and a single, optional `defaultClause`, in any order. Each `caseClause` contains an `expression` and an optional `statementList`, while the `defaultClause` contains only an optional `statementList`. During execution, the `expression`s in the `caseClause`s are evaluated in source text order, and each value is compared to the value of the outer switch `expression`. If the values are equal, the optional `statementList` for that clause is executed. Unless a `break` statement is encountered in the `statementList`, the next `caseClause` or `defaultClause` in the sequence will be evaluated. Because this is the defined behavior of a `switch` statement, all of the `caseClause`s and the `defaultClause` in a `switch` statement must be able to execute on a common set of browsers. This relation is defined in Rules 4.4.21. The rules for optional case clauses and optional default clauses are self-explanatory.

Rule 4.4.21.

$$\frac{\phi \vdash_{OPTCCS} occs_1 : \Gamma_{occs_1} \qquad \phi \vdash_{OPTCCS} occs_2 : \Gamma_{occs_2} \qquad \phi \vdash_{OPTDC} odc : \Gamma_{odc}}{\phi \vdash_{CASEBLK} \texttt{caseBlock}(occs_1,\, odc,\, occs_2) : \Gamma_{occs_1} \sqcap \Gamma_{odc} \sqcap \Gamma_{occs_2}}$$

Rule 4.4.22.

$$\frac{}{\phi \vdash_{OPTCCS} \texttt{optCaseClauses}() : (\beta,\, \beta)}$$

$$\frac{\phi \vdash_{CCL} ccl : t}{\phi \vdash_{OPTCCS} \texttt{optCaseClauses}(ccl) : t}$$

Rule 4.4.23.

$$\frac{}{\phi \vdash_{OPTDC} \texttt{optDefaultClause}() : (\beta,\, \beta)}$$

$$\frac{\phi \vdash_{DC} dc : t}{\phi \vdash_{OPTDC} \texttt{optDefaultClause}(dc) : t}$$

Because execution may proceed from one case clause to the next if the first does not contain a `break` statement to force an exit from the `switch` statement, the runability type of a sequence of case clauses must reflect the common set of browsers that will correctly execute each case. Rule 4.4.24 states this relation for a list of `caseClause`s.

Rule 4.4.24.

$$\frac{}{\phi \vdash_{CCL} [] : (\beta,\, \beta)}$$

$$\frac{\phi \vdash_{CC} cc : \Gamma_{cc} \qquad \phi \vdash_{CCL} ccl : \Gamma_{ccl}}{\phi \vdash_{CCL} cc :: ccl : \Gamma_{cc} \sqcap \Gamma_{ccl}}$$

A `caseClause`, as noted earlier, has an `expression` that is used to determine if the optional statement list in the clause will be executed. Since the `expression` must be evaluated first, it may place restrictions of the set of browsers that can execute the statements. Similarly, the statement list itself may also impose restrictions on the runability type of the clause as a whole. Rule 4.4.25 below defines the runability type of an `optStatementList` using Rule 4.4.5 for statement lists, and is followed by the type rule for a `caseClause`. The `defaultClause` is similar to a `caseClause`, except that it does not have an expression to determine if execution will branch to the enclosed statement list. The runability type is then simply the runability type of the enclosed `optStatementList`, with this relation given in Rule 4.4.27.

Rule 4.4.25.

$$\frac{}{\phi \vdash_{OPTSL} \texttt{optStatementList}() : (\beta, \beta)}$$

$$\frac{\phi \vdash_{STMT-LIST} sl : t}{\phi \vdash_{OPTSL} \texttt{optStatementList}(sl) : t}$$

Rule 4.4.26.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi \vdash_{OPTSL} osl : \Gamma_{osl}}{\phi \vdash_{CC} \texttt{caseClause}(expr, osl) : \Gamma_{expr} \sqcap \Gamma_{osl}}$$

Rule 4.4.27.

$$\frac{\phi \vdash_{OPTSL} osl : t}{\phi \vdash_{DC} \texttt{defaultClause}(osl) : t}$$

Labelled Statement

A labelled statement is a statement with a label that allows a `continue` or `break` statement to force a jump in the execution path to the labelled statement. The label has no effect on runability, so the type of the labelled statement is the runability type of the statement itself.

Definition 4.4.19. Labelled Statement

```
statement   ::=  labelledStmt(statement)
```

Rule 4.4.28.

$$\frac{\phi \vdash_{STMT} stmt : t}{\phi \vdash_{STMT} \texttt{labelledStmt}(stmt)}$$

`throw` and `try` Statements

The `throw` statement allows an author to raise an exception to indicate an error condition or other abnormal computational state. In conjunction with a `try` statement or statements, exceptions can be used to trap errors and handle them in an orderly fashion at the author's discretion. The `throw` and `try` statements are browser dependent, as they were not present in the early versions of the JavaScript language, and are not supported by earlier browser versions. Of the browsers in the set β used in this analysis, only Netscape6,

MSIE5.5, and MSIE6 provide support for the `throw` statement, and for convenience, I define a runability type Γ_t in Definition 4.4.20 that expresses this dependency. Definition 4.4.21 defines the abstract structure applied to the `throw` and `try` statements for runability type analysis.

Definition 4.4.20. Set of Browsers Supporting Try and Throw Statements

$$\Gamma_t = (\{\text{Netscape6}, \text{MSIE5.5}, \text{MSIE6}\}, \{\text{Netscape6}, \text{MSIE5.5}, \text{MSIE6}\})$$

Definition 4.4.21. Try and Throw Statements

```
statement      ::=  throwStmt(expression)

statement      ::=  tryStmt(blockStmt, catch, finally)

catch          ::= ()
                |    identifier blockStmt

finally        ::= ()
                |    blockStmt
```

The `throw` statement contains an expression that evaluates to the exception the statement raises. The runability type of this expression may further restrict the runability of the statement as a whole, and this is reflected in the type rule for the `throw` statement.

Rule 4.4.29.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr}}{\phi \vdash_{STMT} \texttt{throwStmt}(expr) : \Gamma_t \sqcap \Gamma_{expr}}$$

As with the `throw` statement, the `try` statement construct itself has a restricted browser dependency set. The `try` statement contains a block statement and one or both of a `catch` clause and a `finally` clause. The browser dependency of the `try` statement is determined by the runability types of the block statement, the `catch` and `finally` clauses, and the set of browsers whose JavaScript interpreters implement the `try` statement construct. The `catch` and `finally` clauses are syntactically correct only within a `try` statement. The scope of the identifier in the `catch` clause is limited to the clause's block statement.

Rule 4.4.30.

$$\frac{}{\phi \vdash_{CATCH} \texttt{catch}() : (\beta,\, \beta)}$$

$$\frac{\phi[id \diagup (\beta,\, \beta)] \vdash_{EXPR} expr : t}{\phi \vdash_{CATCH} \texttt{catch}(id, expr) : t}$$

Rule 4.4.31.

$$\frac{}{\phi \vdash_{FIN} \texttt{finally}() : (\beta,\, \beta)}$$

$$\frac{\phi \vdash_{STMT} blk : t}{\phi \vdash_{FIN} \texttt{finaly}(blk) : t}$$

Rule 4.4.32.

$$\frac{\phi \vdash_{STMT} blk : \Gamma_{blk} \qquad \phi \vdash_{CATCH} cc : \Gamma_{cc} \qquad \phi \vdash_{FIN} fc : \Gamma_{fc}}{\phi \vdash_{STMT} \texttt{tryStmt}(blk, cc, fc) : \Gamma_t \sqcap \Gamma_{blk} \sqcap \Gamma_{cc} \sqcap \Gamma_{fc}}$$

Expressions

Expressions in JavaScript are program elements that evaluate to a value. As with statements, the runability type of an expression is dependent upon the kind of expression under consideration and the underlying structure. Definition 4.4.22 is a simplified grammar that will be used to decompose an expression as a basis for the development of runability type rules. When necessary, these productions will be expanded to clearly define the abstract structure of particular expressions.

Definition 4.4.22. JavaScript Expression Structure

```
expression  ::=  thisExpr
             |   identifier
             |   literalExpr(primitiveLiteral)
             |   stringLiteral(abstractDocument)
             |   arrayLiteral(expression list)
             |   objectLiteral(propVal list)
             |   parenExpr(expression)
             |   conditionalExpr(expression, expression, expression)
             |   newExpr(expression, expression list)
             |   arrayAccExpr(expression, expression)
             |   objPropAccExpr(expression, identifier)
             |   functionCallExpr(expression, expression list)
```

```
|    functionDefExpr(identifier, optFormalParams, body)
|    functionDefExpr(optFormalParams, body)
|    assignExpr(expression, expression)
|    binaryExpr(expression, BINOP, expression)
|    unaryExpr(UNOP, expression)
```

`this` Expression

The value of a `this` expression is bound to the **this** value of the execution context. That context may be global or may be the context within a function call. The runability type of a `this` expression is then the runability type of the current environment ϕ, denoted by Γ_ϕ.

Rule 4.4.33.

$$\overline{\phi \vdash_{EXPR} thisExpr : \Gamma_\phi}$$

Identifier Expression

Definitions 4.2.9 and 4.2.10 specify sets of user-defined and built-in identifiers and functions to map identifiers to runability types, and these mappings are included in the current environment by Definition 4.2.11. An identifier expression is simply the referencing of an identifier name, as shown in Definition 4.4.22. Determination of the runability type of an `idExpr` is performed using Rule 4.4.34.

Rule 4.4.34.
Let id be an `identifier` in the following.

$$\overline{\phi \vdash_{EXPR} id : \phi(id)}$$

Literal Expressions

The ECMAScript standard [2] specifies syntax for seven kinds of literal expressions, as defined in the abstract grammar below. Of these seven, four are "primitive" type literals, namely `undefinedLiteral, nullLiteral, numberLiteral` and `booleanLiteral`. These literal types are supported by all browsers, as reflected in Rule 4.4.35 below.

Definition 4.4.23. Literal Expressions

```
expression     ::=  literalExpr(primitiveLiteral)

primitiveLiteral ::= undefinedLiteral
                 |    nullLiteral
                 |    numberLiteral
                 |    booleanLiteral
```

Rule 4.4.35.
Let pl be a `primitiveLiteral` in the following.

$$\frac{}{\phi \vdash_{EXPR} \texttt{literalExpr}(pl) : (\beta, \beta)}$$

The runability type of a string literal depends upon what the string represents. A string might contain HTML markup to be written out into a document to provide alternative content, as described earlier in this section. This HTML code may itself be browser dependent. Based on the assumption that an author will use JavaScript to provide alternate HTML code, a string literal containing browser-dependent HTML should be interpreted as being browser dependent because if it is written into a HTML document, the resulting document would have browser dependency at that point. In another case, a string might also contain a HTML `SCRIPT` tag and accompanying JavaScript code. To correctly determine the runability type of a string literal, the string literal must be interpreted as an `abstractDocument` using Rule 4.3.1.

Definition 4.4.24. String Literals

```
expression  ::=  stringLiteral(abstractDocument)
```

Rule 4.4.36.

$$\frac{\phi \vdash_{DOC} doc : t}{\phi \vdash_{EXPR} \texttt{stringLiteral}(doc) : t}$$

An array literal or initializer is a sequence of zero or more expressions separated by commas and enclosed in square brackets. Because the array is an object itself, each expression in the sequence should share a common browser dependency, and that shared dependency will be the dependency of the array as a whole. Definition 4.4.25 specifies the abstract structure used for runability type analysis, while Rules 4.4.37 and 4.4.38 specify the runability of an expression list and an array literal.

Definition 4.4.25. Array Literals

```
expression     ::=  arrayLiteral(expression list)
```

Rule 4.4.37.

$$\frac{}{\phi \vdash_{EXPR-LIST} [] : (\beta, \beta)}$$

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi \vdash_{EXPR-LIST} el : \Gamma_{el}}{\phi \vdash_{EXPR-LIST} expr :: el) : \Gamma_{expr} \sqcap \Gamma_{el}}$$

Rule 4.4.38.

$$\frac{\phi \vdash_{EXPR-LIST} el : t}{\phi \vdash_{EXPR} \texttt{arrayLiteral}(el) : t}$$

Object literals have a syntax similar to array literals, and the runability type is determined in much the same manner. An object literal is initialized by zero or more property name — value expression pairs. The property name(s) are identifiers allowing access to property values within the new object, and do not have a runability type until they are bound to the result of the value expression. As with the array literal, all value expressions included in the initialization of an object must share a common browser dependency. The environment must also be augmented with the property name identifiers. The abstract structure and runability type rule for an `objectLiteral` are shown in Definition 4.4.26 and 4.4.39, respectively.

Definition 4.4.26. Object Literals

```
objectLiteral  ::=  objectLiteral(propVal list)

propVal        ::=  identifier expression
```

Rule 4.4.39.

$$\frac{\phi \vdash_{PVL} pvl : t}{\phi \vdash_{EXPR} \texttt{objectLiteral}(pvl) : t}$$

$$\frac{}{\phi \vdash_{PVL} [] : (\beta, \beta)}$$

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr} \qquad \phi[id \diagup \Gamma_{expr}] \vdash_{PVL} pvl : \Gamma_{pvl}}{\phi \vdash_{PVL} \texttt{propVal}(id, expr) :: pvl : \Gamma_{expr} \sqcap \Gamma_{pvl}}$$

Parenthesized Expression

The parenthesized expression is a grouping syntax in JavaScript. The runability type is that of the enclosed expression.

Definition 4.4.27. Parenthesized Expression

```
expression    ::= parenExpr(expression)
```

Rule 4.4.40.

$$\frac{\phi \vdash_{EXPR} expr : t}{\phi \vdash_{EXPR} \texttt{parenExpr}(expr) : t}$$

Conditional Expression

The conditional expression is similar to the `if-else` statement in terms of determining the runability type. In particular, if the first expression is not browser dependent, the expression will be the greatest lower bound of the runability types of the second and third expressions. If the first expression is browser dependent, the second expression must be compatible with the runability of the first, and the third expression must be runable in the complement of the first expression.

Definition 4.4.28. Conditional Expression

```
expression  ::= conditionalExpr(expression, expression, expression)
```

Rule 4.4.41.

$$\frac{\phi \vdash_{EXPR} e_1 : (\beta,\, \beta) \qquad \phi \vdash_{EXPR} e_2 : \Gamma_2 \qquad \phi \vdash_{EXPR} e_3 : \Gamma_3}{\phi \vdash_{EXPR} \texttt{conditionalExpr}(e_1, e_2, e_3) : \Gamma_2 \sqcup \Gamma_2}$$

$$\frac{\phi \vdash_{EXPR} e_1 : (\delta_1,\, \eta_1) \quad \delta_1 \neq \beta \quad \phi \vdash_{EXPR} e_2 : (\delta_2,\, \eta_2) \quad \phi \vdash_{EXPR} e_3 : (\delta_3,\, \eta_3)}{\phi \vdash_{EXPR} \texttt{conditionalExpr}(e_1, e_2, e_3) : ((\delta_1 \cap \delta_2) \cup (\overline{\delta_1} \cap \delta_3),\, (\eta_1 \cap (\eta_2 \cup \eta_3)))}$$

`new` Expression

The `new` expression is used to construct or allocate a new object in the current execution context. The object type to be allocated is identified by an expression, and

arguments, in the form of an `expressionList` may or may not be present in the `new` expression. In the case where no arguments are given, the runability type is determined by the runability of the object identification expression. If arguments are present, they must share a common browser dependency that must also be compatible with the runability of the object identification expression.

Definition 4.4.29. New Expression

```
expression  ::= newExpr(expression, expression list)
```

Rule 4.4.42.

$$\frac{\phi \vdash_{EXPR} e_{obj} : \Gamma_{obj} \qquad \phi \vdash_{EXPR-LIST} el : \Gamma_{args}}{\phi \vdash_{EXPR} \texttt{newExpr}(e_{obj}, el) : \Gamma_{obj} \sqcap \Gamma_{args}}$$

Array Access Expression

The runability type of an array access expression depends upon the runability of the array object itself as well as that of the index expression. Either subexpression may restrict the runability of the entire expression.

Definition 4.4.30. Array Access Expression

```
expression   ::=  arrayAccExpr(expression, expression)
```

Rule 4.4.43.

$$\frac{\phi \vdash_{EXPR} e_{array} : \Gamma_{array} \qquad \phi \vdash_{EXPR} e_{index} : \Gamma_{index}}{\phi \vdash_{EXPR} \texttt{arrayAccExpr}(e_{array},\ e_{index}) : \Gamma_{array} \sqcap \Gamma_{index}}$$

Object Property Access Expression

The runability type of an object property access expression is determined using both the runability type of the parent object itself, and the property name expression. The expression can take either of two different forms. First is the "dot" notation, i.e. `window.document`, where `window` is the object expression and `document` is the property name identifier. The second syntax is similar in appearance to an array access, except that

a string literal is used as the index expression. In this case, the string literal is the property name identifier, and should be interpreted as an identifier and not as a string literal.

Definition 4.4.31. Object Property Access Expression

```
expression  ::=  objPropAccExpr(expression, identifier)
```

Rule 4.4.44.

$$\frac{\phi \vdash_{EXPR} expr : \Gamma_{expr}}{\phi \vdash_{EXPR} \texttt{objPropAccExpr}(expr, id) : \Gamma_{expr} \sqcap \phi(id)}$$

Function Call Expression

As noted in the discussion of function definitions, the runability type of a function is based on the function's definition. At the site of a function call, the runability of the function may be further modified based on the runability types of any actual arguments in the function call. This new runability type cannot be any less restrictive than the type determined from the function definition, because the source elements in the body of the function are the limiting or restricting factor. Actual arguments may further restrict the runability of the call expression due to the dependencies of the argument expressions. As described earlier in the discussion of function declarations, global variables referenced in the body of a function may also affect the runability type of a function. When the function is defined prior to the declaration of such variables, a list of the variable identifiers is mapped to the function name identifier during the processing of the function declaration. When the function is called, these variables are looked up in the current environment and their runability types are included in the runability type determination of the function call expression. Before stating the rule for analyzing a function call expression, I define a rule for determining the composite runability type of a list of identifiers.

Rule 4.4.45.

$$\frac{}{\phi \vdash_{IDL} [] : \Gamma_{\phi}}$$

$$\frac{\phi(id) = \Gamma_{id} \qquad \phi \vdash_{IDL} ids : \Gamma_{ids}}{\phi \vdash_{IDL} id :: ids : \Gamma_{id} \sqcap \Gamma_{ids}}$$

A function's name is an `identifier`, which may be part of a more complex expression, i.e. an object or array access or the return of another function call. The actual arguments are in the form of an expression list.

Definition 4.4.32. Function Call Expression

```
expression ::= functionCallExpr(expression, expressionList)
```

Rule 4.4.46.

$$\frac{\phi \vdash_{EXPR} e_{funcId} : \Gamma_{funcId} \quad \phi \vdash_{IDL} \kappa(e_{funcId}) = \Gamma_{\kappa} \quad \phi \vdash_{EXPR-LIST} el : \Gamma_{args}}{\phi \vdash_{EXPR} \texttt{functionCallExpr}(e_{funcId}, el) : \Gamma_{funcId} \sqcap \Gamma_{\kappa} \sqcap \Gamma_{args}}$$

Function Definition Expression

In addition to function definitions in the context of a source element, JavaScript also provides the syntax to define a function as an expression. The value of a function definition expression is a function object that can be bound to an identifier, allowing an author to provide a function definition as a function argument, for example. A function definition expression also allows a function to be defined "anonymously" without a function identifier. If it is defined anonymously, the function object is assumed to be bound to another identifier through assignment or argument passing. Furthermore, a function defined as an expression does not have global scope as one defined with a function declaration, and any variables referenced within the body of the function should have been defined prior to the function definition expression. Thus the need to examine the function body for undeclared variable identifiers does not exist for function definition expressions and the runability type can be determined based only on the body of the function.

Definition 4.4.33. Function Definition Expression

```
expression   ::= functionDefExpr(identifier, optFormalParams, body)
             |   functionDefExpr(optFormalParams, body)
```

Rule 4.4.47.

$$\frac{\phi[ofp] \vdash_{JSEL} sel : t \qquad (\phi[ofp])[id \diagup t]}{\phi \vdash_{EXPR} \texttt{functionDefExpr}(id, ofp, \texttt{body}(sel)) : t}$$

$$\frac{\phi[ofp] \vdash_{JSEL} sel : t}{\phi \vdash_{EXPR} \texttt{functionDefExpr}(ofp, \texttt{body}(sel)) : t}$$

Assignment Expression

Assignment expressions allow a value to be bound to an expression that may be a variable identifier, assignable object property, etc. To determine the runability type of the assignment expression, we must consider the runability type of the expressions on both sides of the assignment operator. The runability type of the entire expression will be the common elements of both expressions.

Definition 4.4.34. Assignment Expression

```
expression   ::=  assignExpr(expression, expression)
```

Rule 4.4.48.

$$\frac{\phi \vdash_{EXPR} e_{left} : \Gamma_{left} \qquad \phi \vdash_{EXPR} e_{right} : \Gamma_{right}}{\phi \vdash_{EXPR} \texttt{assignExpr}(e_{left},\, e_{right}) : \Gamma_{left} \sqcap \Gamma_{right}}$$

Binary Expressions

Binary expressions allow the comparison or manipulation of the values of two expressions. Of particular interest in determining runability types are the equality and inequality operators, as they are frequently used to test or "sniff" for a browser's capabilities. Generally, this involves testing the `null` value against an object access expression that is dependent upon a particular browser or set of browsers. If the interpreter implementation does not support the particular expression, it returns a `null` value. Binary expressions other than the equality and inequality expressions do not present the opportunity to perform such a test.

In general, binary operators work strictly on numeric values, or on objects that can be converted to numeric values. The exceptions to this are the `instanceof`, `in`, conjunction (&&), disjunction(||), equality (==, !=), and strict equality or identity operators (===, !==). The general case for the runability of a binary expression is the common elements of the two subexpressions. The `instanceof` and `in` operators also adhere to this rule, although they operate strictly on object types. The identity operators compare primitive, string, and object type, and in the case of object comparisons, test if the two subexpressions refer to the same object. The conjunction and disjunction operators are used to combine boolean-valued expressions. The equality operators also compare primitive, string, and object types,

but in the case of object comparisons they test for equivalent valued expressions, enabling the "browser-sniffing" code described above.

Determination of the runability type of conjunction and disjunction expressions should be considered in two cases. First is the case where both subexpressions are not browser dependent. In this case, the resulting runability type will also be non-dependent. In the second case, one or both of the subexpressions is browser dependent. In this situation, the runability type of the overall expression should reflect the conjunction or disjunction of the runability types of the subexpressions.

Examining the equality operators more closely with respect to runability typing, we consider three cases: both subexpressions are not browser dependent; both are browser dependent; and one is dependent while the other is not. In the first case, if both subexpressions are not browser dependent, the comparison expression is also not browser dependent. Likewise, if both subexpressions are browser dependent, the runability type of the comparison expression will be the common δ and η elements of the two subexpressions. The case where one subexpression is browser dependent and the other is not is potentially a browser detection expression. If the comparison is for equality and the non-dependent expression is a null literal, the runability of the comparison will be the type of the dependent subexpression, because the code is performing a "positive" test. However, if the non-dependent expression is not a null literal, the resulting type will be that of the dependent expression. Similarly, if the comparison is for inequality and the non-dependent expression is a null literal, the test is a "negative" test, and the runability of the entire expression is the complement of the dependent expression. If the non-dependent expression is not a null literal, the resulting runability type is the complement of the dependent expression. If both subexpressions are browser dependent in either an equality or inequality comparison, the resulting runability type will be the least upper bound of the types of the subexpressions. The following rules formalize these concepts.

Definition 4.4.35. Binary Expressions

```
expression   ::=  binaryExpr(expression, BINOP, expression)

BINOP        ::=  &&
              |   ||
              |   ==
              |   !=
```

```
|    OTHER-OP
```

Rule 4.4.49.

$$\frac{\phi \vdash_{EXPR} e_1 : \Gamma_1 \qquad \phi \vdash_{EXPR} e_2 : \Gamma_2}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ \&\&,\ e_2) : \Gamma_1 \sqcap \Gamma_2}$$

$$\frac{\phi \vdash_{EXPR} e_1 : \Gamma_1 \qquad \phi \vdash_{EXPR} e_2 : \Gamma_2}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ \|,\ e_2) : \Gamma_1 \sqcup \Gamma_2}$$

$$\frac{(\phi \vdash_{EXPR} e_1 : (\beta,\ \beta) \wedge e_1 = \texttt{nullLiteral}) \qquad \phi \vdash_{EXPR} e_2 : \Gamma_{e_2}}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ ==,\ e_2) : \Gamma_{e_2}}$$

$$\frac{\phi \vdash_{EXPR} e_1 : \Gamma_{e_1} \qquad (\phi \vdash_{EXPR} e_2 : (\beta,\ \beta) \wedge e_2 = \texttt{nullLiteral})}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ ==,\ e_2) : \Gamma_{e_1}}$$

$$\frac{\phi \vdash_{EXPR} e_1 : \Gamma_{e_1} \qquad \phi \vdash_{EXPR} e_2 : \Gamma_{e_2}}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ ==,\ e_2) : \Gamma_{e_1} \sqcap \Gamma_{e_2}}$$

$$\frac{(\phi \vdash_{EXPR} e_1 : (\beta,\ \beta) \wedge e_1 = \texttt{nullLiteral}) \qquad \phi \vdash_{EXPR} e_2 : (\delta_2,\ \eta_2)}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ !=,\ e_2) : (\overline{\delta_2},\ \beta)}$$

$$\frac{\phi \vdash_{EXPR} e_1 : (\delta_1,\ \eta_1) \qquad (\phi \vdash_{EXPR} e_2 : (\beta,\ \beta) \wedge e_2 = \texttt{nullLiteral})}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ !=,\ e_2) : (\overline{\delta_1},\ \beta)}$$

$$\frac{\phi \vdash_{EXPR} e_1 : \Gamma_{e_1} \qquad \phi \vdash_{EXPR} e_2 : \Gamma_{e_2}}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ !=,\ e_2) : \Gamma_{e_1} \sqcap \Gamma_{e_2}}$$

$$\frac{\phi \vdash_{EXPR} e_1 : \Gamma_1 \qquad \phi \vdash_{EXPR} e_2 : \Gamma_2}{\phi \vdash_{EXPR} \texttt{binaryExpr}(e_1,\ \texttt{OTHER} - \texttt{OP},\ e_2) : \Gamma_1 \sqcap \Gamma_2}$$

Unary Expressions

In general, the runability type of a unary expression will be the same as the runability type of the subexpression to which the unary operator is applied. The exception to this is the logical negation operator (!) when it is applied to an expression that is browser dependent. In this case, the resulting runability type should reflect the complement of the dependency of the subexpression.

Definition 4.4.36. Unary Expressions

```
expression   ::=  unaryExpr(UNOP, expression)

UNOP         ::=  !
              |   OTHER-UNOP
```

Rule 4.4.50.

$$\frac{\phi \vdash_{EXPR} expr : (\delta',\, \eta') \qquad \delta' \neq \beta}{\phi \vdash_{EXPR} \texttt{unaryExpr}(!, expr) : (\overline{\delta'},\, \beta)}$$

$$\frac{\phi \vdash_{EXPR} expr : (\beta,\, \beta)}{\phi \vdash_{EXPR} \texttt{unaryExpr}(!, expr) : (\beta,\, \beta)}$$

$$\frac{\phi \vdash_{EXPR} expr : t}{\phi \vdash_{EXPR} \texttt{unaryExpr}(\texttt{OTHER} - \texttt{UNOP}, expr) : t}$$

Conclusion

This chapter presents a unified, inductively defined rule set for determining the runability type of a document containing HTML and/or JavaScript code. Although a a limited set of browsers was used to define a non-dependent code segment, the rule set itself is not dependent upon the elements of the β set. The browser set β was defined as it was based on the the most used browsers and operating system platforms, to be representative of the majority of web users. This set can be expanded to include other browsers and platforms, provided dependency data is accumulated for both HTML tag names and built-in JavaScript identifiers. The accumulation of this data is discussed in Chapter 5.

Chapter 5

Implementation

This chapter details the implementation of the Document Runability Analysis Tool (DRAT). The first section covers the construction of type classes for HTML tag names and JavaScript identifiers for built-in functions, objects, and properties. This section also comments on the process by which browser compatibility was established for these identifiers. The second section delves into several significant design issues, including the choice of implementation language, the use of existing tools, and an overview of the data flow in the analysis process. The third section explains the actual implementation of the tool. This section also explains a significant problem that occurred during the implementation and preliminary testing of the tool, and how that problem was resolved.

5.1 Determination of Type Classes

Determination of runability type classes involved three separate tasks. First, a list of all identifiers available as built-in names in the target browsers had to be compiled. Next, the compatibility characteristics for each identifier had to be determined, identifying which browsers correctly processed each name in its usage context and which browsers generated errors when a particular name was encountered. Finally, these characteristics were grouped together based on common browser sets. I cover each of these steps below. The process of determining these type classes was the most time consuming part of the

DRAT implementation.

One might assume that browser vendors would provide detailed documentation of the behavior of their products to enable Web authors to take full advantage of a browser's features and capabilities, particularly considering the competitive nature of the Web browser market. This is not the case, as reliable, first-hand documentation is generally not available, and when the information is provided, it is fragmented and poorly organized. Developing a comprehensive list of identifiers for the limited set of browsers given in Rule 4.2.4 in the previous chapter proved to be a significant challenge. A variety of sources had to be culled through to develop a comprehensive list of identifier names. The W3C Recommendations for HTML [25] and XHTML [7] include the standard HTML tag names, but do not mention non-standard tag names implemented by some versions of Netscape and Microsoft browsers. *HTML & XHTML: The Definitive Guide*[22] listed these tag names along with their compatibility characteristics. The ECMAScript language specification [2] provided the identifier names for the core implementation of JavaScript, but did not include any documentation of possible client-side object, property, or function names. The Mozilla.org website includes a test matrix for client-side JavaScript at [13], but there are many property and method identifiers that are listed without test case implementations. These tests also do not include identifiers no longer supported by Mozilla/Netscape browsers, or those implemented specifically by Microsoft browsers. To complete the list, I used *JavaScript: The Definitive Guide*[14].

The process of determining which browsers are compatible with the accumulated identifier names was as cumbersome as building the list itself. The Mozilla test matrices [12, 23, 13] were helpful, but as noted above, were incomplete with respect to the list of identifiers. Also, the test framework includes JavaScript code that is not supported by Netscape 4.x browsers or Internet Explorer version 4.0 and 5.0, limiting their usability for these browsers. Another resource that had limited use is a W3C DOM Level 1 test suite developed by the National Institute of Standards and Technology [11]. This test suite was written specifically for Internet Explorer version 5.0 and later browsers, and will not function at all on any Netscape browsers. The two reference books that proved useful in building the list of identifier names, *HTML & XHTML: The Definitive Guide*[22] and *JavaScript: The Definitive Guide*[14] were the final resource for determining which browsers were compatible with the identifiers.

Complicating this process was the need to know not only if a browser correctly

processed a particular identifier name, but also to determine whether or not a reference to a name caused an error in the browser. For HTML tag names, this was trivial, since a Web browser will ignore any HTML tag names not supported by the browser. For JavaScript, the situation was not so clear. In some cases, unsupported identifiers are coerced by the interpreter to a **null** or **undefined** value and avoid generating an error. In other cases, this coercion does not occur, and errors result. This is the case where an identifier is associated with a function with a non-void return type, or refers to an object that serves as a container for other values or objects. In these cases, the interpreter expects to be able to manipulate the object the name refers to, and a **null** or **undefined** value cannot be manipulated as an object value, generating an error condition. As noted in the previous chapter, it is necessary to identify all three kinds of behavior because browser detection code generally relies upon testing some value for a null value.

At total of five hundred eighty-eight identifiers were cataloged with their browser compatibility data: one hundred four HTML tag names and four hundred eighty-four built-in JavaScript identifiers. With this compatibility data, I could then assign a runability type to each identifier, and for convenience in the implementation, grouped them by common types. HTML tag names groups are listed in Table B.1, with their respective runability types listed in Table B.2, both of which may be found in Appendix B. JavaScript core and client-side identifiers are similarly listed in Tables B.3 and B.5, respectively, with their associated runability types in Tables B.4 and B.6. These groups could then be used to initialize a symbol table enabling the analyzer to look up runability types for all possible built-in identifiers. The HTML tag names and built-in JavaScript identifiers were grouped separately, as Rule 4.2.11 implies a three-step lookup process.

5.2 Design Considerations

An important design consideration for the implementation was to demonstrate that the analysis could be performed using the same kind of infrastructure already existing in Web browsers and other HTML document rendering software, and that no extraordinary manipulation of the target document would be required. Such an design could also be readily translated to other document classes like word processing documents, spreadsheets, etc. For this reason, I chose to build this implementation on top of existing HTML and

JavaScript parsers rather than to construct these parsers from scratch. A second design consideration was the implementation language. The choice of parsers would influence this decision, but support for accessing and manipulating Web-based resources was considered to be very important, as the goal of the implementation was to access and analyze a Web-based document. A final consideration for the parser tools was to be able to access the source code for the HTML and JavaScript parsers, in particular the generated Abstract Syntax Tree representation generated by the parsers, since this tree representation is what will be analyzed. The remainder of this section discusses these considerations in more detail.

5.2.1 HTML Parser Considerations

There are many HTML parsers available that meet the basic criteria outlined above, i.e. the source code is available, they construct well-defined tree representations, and these trees are easily traversable. However, nearly all of these are "standards-based" parsers that require HTML documents to adhere to the W3C HTML specification [25]. These parsers do not recognize non-standard, browser-specific tag names will also fail when various end tags are omitted. Since the objective of this tool is to determine browser dependencies, browser-specific tag names must not be ignored by the parser. Also, because most HTML rendering applications do not fail as a result of omitted end tags, this should also not cause this tool's parser to fail, either. What was needed was a simple parser that would build a basic parse tree from a HTML document, include the tag names as part of the tree structure, and synthesize missing end tags when necessary. Two parsers fit these criteria, along with being easily portable across hardware platforms and operating systems: a Perl HTML parser module [1] and a HTML parser [4] written in Java and generated using the JavaCC [3] compiler generator.

5.2.2 JavaScript Parser Considerations

There were fewer choices for a JavaScript parser. The ECMAScript language specification [2] provides a complete grammar for the core language, which would be sufficient as it defines the syntax for JavaScript, as implemented within Web browsers and similar applications. The Scriptonite project [16] intended to use this grammar to build a complete, stand-alone ECMAScript interpreter, and their website yielded a SableCC [15] specification

based on version 3 of the ECMAScript standard. The Rhino Project [10] at Mozilla was another consideration, as it is an interpreter written in Java based on the C implementation used in the Mozilla and Netscape Web browsers. This connection made this choice an early favorite. The other possibility was the Free EcmaScript Interpreter project (FESI)[20], but this choice was eliminated early as it only conformed to JavaScript version 1.1, and the current version, 1.5, has made significant structural changes to the language, including the incorporation of **switch**, **throw**, and **try** statements, and support for regular expressions.

The Rhino interpreter had two problems, with respect to its incorporation in this tool, that eliminated it as a choice. The lexer, parser, and interpreter are very tightly integrated in the Rhino code, and no interface to a parse tree structure is exposed outside of the interpreter itself. Also, the classes in the interpreter are not public, other than the methods to create an interpreter and related objects, and to evaluate an input script object. These concerns eliminated the Rhino code base from consideration at this phase of the implementation. This left the Scriptonite SableCC specification, which was a thorough and accurate reproduction of the ECMAScript grammar, with one exception. The Scriptonite grammar required whitespace characters to immediately follow a postfix operator (++ or –) in a postfix expression, which the ECMAScript grammar did not. For the purpose of this implementation, the assumption is that the input source code is syntactically correct and executable on the appropriate interpreter, and has been written unambiguously with respect to postfix/prefix expressions, i.e., that another intervening token would clearly and positively disambiguate a questionable expression in the manner the code author intended. With this assumption, it was possible to eliminate the whitespace requirement from the grammar specification.

5.3 Implementation Details

The choice of using the Scriptonite specification and SableCC to generate the core of the JavaScript parser also resolved the choice of the HTML parser code in favor of the Java-based parser. There are several additional benefits of using the Java language. Java's strong type system, Unicode character data support, and core library support for accessing network-based resources simplified the coding process. Java Runtime Environments (JREs) are available for many different platforms, allowing the application to be run in a variety

of different computing environments. The dynamic linking system used by Java allows the extension of existing code without requiring changes to the existing code. The remainder of this section will cover specific details of the implementation, starting with an overview of the application design and modularization.

The design of the Document Runability Analysis Tool incorporates six principal modules, including the HTML and JavaScript parsers discussed earlier in this chapter. The JavaCC HTML parser code was slightly modified from the distribution form to provide a simplified interface to several attributes of the HTML node classes in the tree structure the parser generates. The code segments that were modified were hand-coded and were not generated by JavaCC, and the functionality of the parser itself was unchanged. SableCC was used to generate the JavaScript parser from the Scriptonite grammar specification, and the resulting code was used unchanged. The four additional modules include the modified HTML tree structure based on the structural specification given in Definition 4.3.1, the type system, the environment, and the analysis modules.

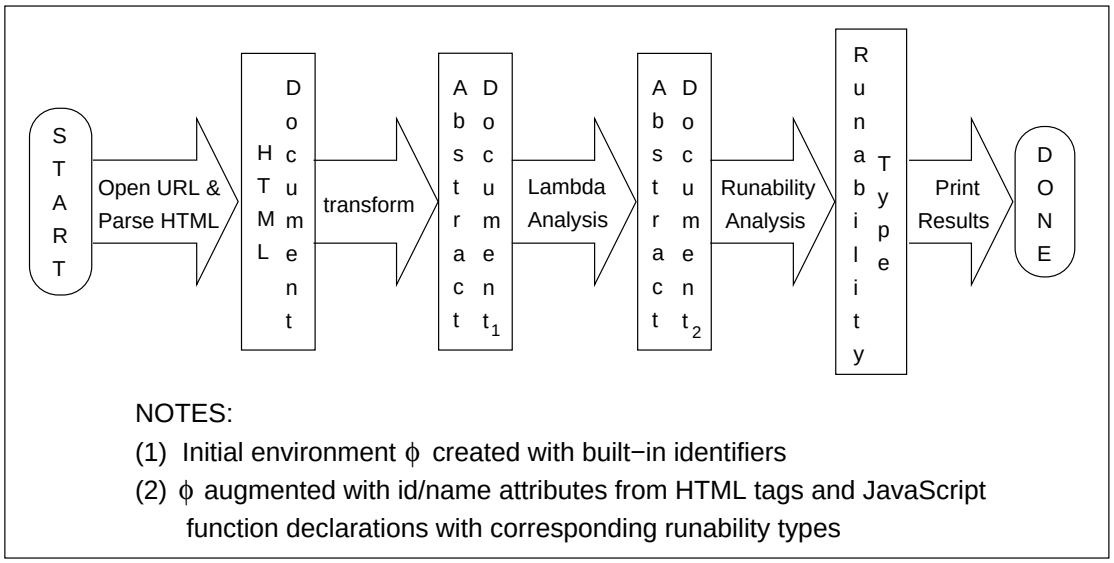

Figure 5.1: Implementation of Runability Analysis

Figure 5.1 illustrates the sequence of operations in the DRAT design. Execution begins by opening the URL specified by the user as the command-line argument. The input character stream associated with this URL is passed to the HTML parser, which generates a tree-structured HTMLDocument. This structure contains nodes corresponding to HTML

tags and tag blocks, HTML comments, text data, and new lines. While this structure is an accurate abstraction of a HTML document, it does not match the **abstractDocument** structure as defined in Chapter 4. Maintaining this structure through the analysis process adds unnecessary complexity to the design of the analysis, so the HTML document is transformed into an **abstractDocument**-based tree structure. In this transformation, HTML comment, text data, and new line nodes in the original structure become **noCodeNodes** in the new **abstractDocument** tree. Tag nodes and tag block nodes not delimited by **script** tags are transformed to **htmlNodes**, and script blocks become **scriptNodes**. At this time, the initial environment is also created, including the lookup table for all built-in JavaScript identifiers and HTML tag names.

Two separate operations are performed during the Lambda Analysis phase. First, the Λ function of Definition 4.4.4 is applied to JavaScript source code contained in the **scriptNodes** of the **abstractDocument**. Second, all other HTML tags are checked for either the *id* or *name* attributes. These user-defined identifiers allow JavaScript code to access and manipulate elements within the HTML document outside the JavaScript code itself. The environment is augmented with the identifier and the Runability Type of the HTML tag name. The Runability Analysis phase directly implements the type rules of Chapter 4, traversing the **abstractDocument** and calculating the overall Runability Type of the document. The results are printed to the screen as two lists of Web browsers: a set that will correctly execute and render the document, and a set that will load the document without alerting the user to any error conditions. The times required to load/parse the document and to perform the analysis are also listed.

5.3.1 Modified HTML Tree Structure Module

As noted earlier in this section, the tree structure generated by the HTML parser, defined by **HtmlDocument.java**, does not match the **abstractDocument** structure defined in Chapter 4, and would have made coding the analysis awkward and cumbersome. To simplify this, I transformed this structure into the **abstractDocument** structure prior to performing the analysis. The source files **AbstractDocument.java**, **DocSegment.java**, **DocSegmentList.java**, **HTMLNode.java**, **NoCodeNode.java**, and **ScriptNode.java** define this structure and the transformation in the implementation. For tag block nodes in the original tree, the HTML parser collects the script source as a single block of text that

becomes a child of the tag block node. During the transformation, the JavaScript parser is called on this text to generate a parse tree based on the `ecmascript.scc` grammatical specification, and this parse tree becomes a child of the **ScriptNode** object created in the transformation. A common practice in the past, when browser support for scripting languages was not common, was to enclose script source code in an HTML comment. The comment start and end tags, `<!--` and `-->` should not be parsed as part of the JavaScript source, and are stripped off before the parser is called. The tree structure resulting from the transformation is exactly the structure defined in Chapter 4.

5.3.2 Type System Module

The type system module is a straightforward implementation of the design established in Chapter 4. Each Web browser in the set β (Definition 4.2.4) is modeled by a class implementing the **Browser** interface. The **BrowserSet** class describes a set of **Browser** objects, and includes static methods for calculating the union and intersection of two **BrowserSet** objects. Two **BrowserSets** make up a **RunabilityType** object as described in Definition 4.2.5. The **RunabilityType** class, similar to the **BrowserSet** class, defines two static methods for calculating the Least Upper Bound (LUB) and Greatest Lower Bound (GLB) of two **RunabilityType** objects.

5.3.3 Environment Module

The environment module centers on the **Phi** class, directly implementing Definitions 4.2.11, 4.2.12, and 4.4.6. The **Phi** class contains three **SymbolTable** objects, one holding HTML tag names and their respective runability types, one holding built-in JavaScript identifiers and their runability types, and the third for user-defined identifiers. The **SymbolTable**s for tag names and built-in identifiers are initialized from the **HTMLIds** and **JavaScriptIds** classes respectively. These two classes contain arrays of identifier names and methods to load a **SymbolTable** object with those names and appropriate runability types. The identifiers and types are taken directly from the data in the tables listed in Appendix B.

The **Phi** class, performs a maximum of three lookup operations when a client requests the runability type of an identifier name. As defined in Chapter 4, the set of

HTML tag names is searched first, followed by the set of built-in JavaScript identifiers, if the first lookup was unsuccessful, and ending with a lookup into the set of user-defined identifier names. The **SymbolTable** class extends the **Hashtable** class included in the Java SDK distribution to provide an interface tailored to this application while leveraging the optimized implementation of the library class. Entries in the table consist of a (**Identifier**, **RunabilityType**) pair, where the **Identifier** object is the key value for the hash table. It was necessary to implement the **Identifier** class rather than just using a **String** object to represent an identifier as a result of how JavaScript function declarations and function calls are analyzed. Since a function declaration may include a reference to global variables that have not been declared at the point where the function is defined. When the function is called later in the code, these variables have been declared, and their values accessed by the function's execution. To capture this situation, the analysis of a function declaration requires the accumulation of a list of undeclared identifiers encountered during the analysis. This identifier list is maintained as part of the function's identifier, and when a call is made, the previously undeclared identifiers are looked up in the current environment and their runability types merged with the type of the function and the actual arguments, as described in Rules 4.4.3 and 4.4.46. The **FunctionIdentifier** class extends the **Identifier** class to provide this closure functionality while allowing the function name to be used as the key for the hash table.

5.3.4 Runability Type Analysis Module

The code supplied with the JavaCC HTML parser, as well as that generated from the Scriptonite specification, incorporate the "Visitor" design pattern [17], allowing the actual analysis code to be implemented in just two classes, one a Visitor implementing the Lambda function of Definition 4.4.4, and the other a Visitor implementing the Runability Analysis. The Visitor pattern externalizes the implementation of an operation to be performed upon the elements of a data structure from those elements. Figure 5.2 illustrates a simple implementation of the Visitor pattern in Java. The **Main** class is the application in this example, and the **rootNode** object is assumed to be the root of some **Node**–based structure the programmer wishes to traverse depth-first. The **rootNode** object "accepts" the visitor by calling the **visit** method of the visitor object, passing a reference to itself as the argument to the call. The visitor then performs the desired computation on the

Node object, in this case, by "visiting" the child of the **Node** first, then performing some computation on the **Node** itself.

```
public interface Visitable {
    public void accept(Visitor vis);
}

public interface Visitor {
    public void visit(Node n);
}

public class Node implements Visitable {
    protected Node child;
    ...
    public Node getChildNode() {
        return child;
    }
    public boolean hasChild() {
        return (null != child);
    }
    public void accept(Visitor vis) {
        vis.visit(this);
    }
}

public class DepthFirstNodeVisitor implements Visitor {
    public void visit(Node n) {
        if (n.hasChild()) {
            (n.accept(this);   // visit the child first
        }
        fixNode(n);
    }
    private void fixNode(Node n) {
        // manipulate the Node in some manner
    }
}

public class Main {
    ...
    rootNode.accept(new DepthFirstNodeVisitor());
    ...
}
```

Figure 5.2: Simple Visitor Implementation in Java

In an extension of this very simple example, suppose that the **Node** class was a base class for a variety of more specialized classes, for instance, classes representing specific node types in a parse tree. The class implementing the visitor would then implement a different **visit** method for each **Node**–derived class in the parse tree hierarchy, relying on Java's

dynamic dispatching to match a particular visitor method based on the runtime–determined type of the object passed as the argument to the function call. The clear advantage to this traversal mechanism is that the classes making up the nodes of the parse tree do not have to be modified every time some new operation is to be performed upon the tree structure. The parse tree is solely a data or state storage device, while any manipulation or computation involving the tree is handled by an external class. From a software engineering perspective, this is also attractive, since all of the code required to perform a particular operation upon the tree is contained within a single class instead of being spread out across all the classes in the tree structure. Additional operations require writing a single class to perform the operation while implementing the Visitor interface instead of modifying each class within the tree hierarchy.

The SableCC Compiler Compiler automatically generates classes representing the tree structure defined by the grammatical specification given as input. These classes implement a variation on the Visitor pattern that provides the "look and feel" of a **switch** statement, but the fundamental pattern and behavior is the same. SableCC also generates a Java interface file providing all the methods necessary to implement the Visitor variant, as well as three classes that implement that interface: a basic Visitor that provides stub implementations of all interface methods, a depth–first visitor, and a reverse–depth–first visitor. These classes may then be extended to implement operations on the tree that require these traversal methods. The two classes used in this application are **LambdaAnalysis** and **RunabilityAnalysis**. The **RunabilityAnalysis** class extends the SableCC–generated **DepthFirstAdapter** class to leverage the traversal code in the generated class. The **RunabilityAnalysis** class also implements the **AbstractDocumentVisitor** interface, which allows traversal of the entire **AbstractDocument**, including both the HTML and JavaScript portions of the structure. The **LambdaAnalysis** class extends the **RunabilityAnalysis** class as it requires much of the analysis functionality to determine the runability type of function declarations. The analysis implementation was coded directly from the type rules developed in Chapter 4, employing both the tree structure of the **AbstractDocument** and the functionality provided by the static methods of the **RunabilityType** and **BrowserSet** classes.

The analysis implemented in the first version of the tool did not perform any analysis on the attributes of HTML tags, except for two cases. First, for any tags with a `name` or `id` attribute, the value associated with the attribute was augmented to the

environment with the runability type of the tag name, to allow proper analysis of JavaScript code that accessed a document element using this identifier. Second, the `type` or `language` attributes of each `script` tag were checked for the presence of either "`javascript`" or "`jscript`" before attempting to parse the source code in the element. Any other script language specifications cause the script node in the document to be ignored. Both of these simplifications caused the analysis to be inaccurate in certain circumstances, however. Ignoring the possibility of JavaScript values associated with HTML tag attributes also ignored the opportunity for such code to impart additional browser dependencies to the document. This analysis also assumed that document authors would not mix different scripting languages within the same document.

The analysis of the Microsoft and CBS Web pages were suspect for similar reasons. For the Microsoft page, the analysis indicated that the Netscape version 4 browsers should correctly render the document, which they did not. Similarly, the results for the CBS.com page also indicated, incorrectly, that in should be loaded without errors on version 4 Netscape browsers. Careful examination of the HTML and JavaScript code for both pages pointed towards two possible causes. First, the use of JavaScript code to dynamically generate values for HTML tag attributes could cause errors. If the attribute name was recognized by the browser, but the code associated with it was not compatible with the particular browser version, an error would be indicated. Second, VBScript, which is supported only by Microsoft browsers, was used in both documents in contexts that could affect the ability of Netscape browsers to correctly render them.

To overcome these problems, the analysis rules in Chapter 4 were modified to handle these situations. The definition of the top level of the `abstractDocument` structure (4.3.1) was changed to include HTML tag attributes in the `htmlNode` substructure, and to include a language attribute in the `scriptNode` substructure. Additional definitions and rules were also added to specify the analysis of these enhanced structures. Specifically, any HTML attribute value containing JavaScript code is now analyzed during the Runability Analysis phase, with the results merged with the runability analysis of any children of the node and the runability type tag name itself. Additionally, any `scriptNode`s with a "VBScript" language attribute are given a runability type restricting the compatibility to Internet Explorer browsers in terms of correct rendering, while allowing all browsers to load the node without errors, since the Netscape browsers will ignore `script` tags and their contents if the language specification is VBScript. The results after making these changes

in the implementation are now accurate with respect to confirming each browser's behavior with the test cases, as described in detail in Chapter 6.

The source code for the DRAT application is available for download by selecting this **link to drat-src.zip**. The Java SDK version 1.3 or later is required to compile the source code. A binary distribution is also downloadable by selecting this **link to Drat.jar**. Note that the binary distribution does not include the source code. See the README files in both distributions for installation, compilation, and execution details.

Chapter 6

Case Study

This chapter discusses six case studies of the use of the DRAT application. These six Web pages were chosen as representative of different levels of JavaScript usage within a Web document and having different browser compatibilities, including one document that does not incorporate any JavaScript code. This chapter is organized as follows: first, some background on the document selection process will be explained, followed by a discussion of each test case. The chapter concludes with a summary of the results.

6.1 Background

There were two reasons more test cases were not chosen. First, while installing different versions of Netscape browsers on the same computer is easily accomplished by installing to different directory locations, the same is not possible with Microsoft Internet Explorer. Installing a later version of Internet Explorer either completely overwrites an earlier version previously installed or updates common components of the earlier version, altering the functionality to be more compatible with the later version. Uninstalling a later version did not reverse any updates to common files, leaving the earlier version altered from its initial functionality. In particular, reinstalling version 4.0 required reinstalling the Windows operating system, since the browser installation program detected the presence of the newer version of Internet Explorer and would not replace it with an older version. To

correctly evaluate the correctness of a DRAT analysis required checking all of the results on one Internet Explorer version before installing the next version.

The second reason a limited set of Web document was chosen is due to many documents' dependence upon the "automatic semicolon insertion" functionality of the JavaScript interpreters incorporated in most Web browsers. This functionality is within the standard for the core language [2], but was not implemented in the JavaScript parser used in the DRAT application. The reasons for this will be discussed in the next chapter with regard to future work. The effect of this decision was that many Web pages containing JavaScript generated numerous parse errors because statements were not terminated with semicolons. Of the six cases studied in this thesis, three fell into this category. To enable testing of these cases, the Web pages and any linked JavaScript source files were saved locally, and the JavaScript source edited to correct the parse errors. The resulting documents were then served to the application and browsers via a local Web server.

As noted earlier, one of the test cases is a Web document that does not contain any JavaScript code. Two cases were chosen as representing a bias towards each browser family, and both are heavily scripted. One search engine Web page was chosen since search engines are popular Websites, and it contains only modest scripting. Finally, two media outlet Web pages were chosen. One of these contains a moderate amount of JavaScript code, while the other is very heavily scripted. The heavily scripted document includes not only JavaScript code, but also VBScript, Microsoft's alternative to JavaScript that provides alternative functionality incompatible with JavaScript, as well as a significantly different syntax. In the initial version of DRAT, script nodes containing VBScript code were ignored, as noted in the previous chapter on the implementation. The second version of the tool does not completely ignore VBScript script nodes in a document, but does not attempt to analyze the script source in detail. Instead, the node is given a runability type indicating correct functionality only with Internet Explorer browsers. Future versions of DRAT may incorporate more detailed source code analysis of VBScript, as well as other scripting languages that may become popular.

Two computing platforms were employed in the testing of the DRAT application. The primary client platform was a Intel Celeron-based computer with Microsoft Windows 2000 SP2 as the operating system. Netscape versions 4.07, 4.79, and 6.23 were installed on this system, as well as successive installations of Microsoft Internet Explorer versions 5.5 and 6.0. Internet Explorer 4.0 could not be installed with the Windows 2000 operating

system, and I had upgraded Internet Explorer to version 5.5 before beginning this research. The second computer was a Intel Pentium system with the Microsoft NT 4.0 SP6a operating system. This system also ran the Apache HTTP Server, version 2.0.43, used to serve the edited Web documents over a 100BaseT Ethernet connection between the two computers. This system was also used as a client system for Internet Explorer versions 4.0 and 5.0 for checking the results of the runability analysis. Internet connections for both computers were via an analog modem dialup connection at 45.2 Kbps.

6.2 Test Cases

This section details the test cases analyzed using the DRAT application and compares the results with actual browser performance. Additional background information is provided about each case. Analysis times for all test cases ranged from approximately one to five seconds, although there did not appear to be a direct relationship between the size of the document and the time required for analysis. These times also did not include load time, as this may vary widely depending upon the network traffic between the web server and the client system. It should be expected that this document analysis would take less time than that required to load and render multimedia-rich Web documents, since the analysis loads only the character data portion of the document. The analysis is fast enough to be practical and useful on a widespread basis.

6.2.1 NCSU Homepage

The North Carolina State University homepage (http://www.ncsu.edu) does not contain any script components, JavaScript or otherwise. The document was chosen for this reason, and because of the (previously unproven) belief that the university's homepage should be compatible with all Web browsers. The runability analysis of the document indicated that it should be correctly rendered by all of the Web browsers tested by the current version of DRAT. Checking this analysis with actual browsers confirmed the results – the page was rendered the same on all seven Web browsers.

6.2.2 Microsoft Homepage

The second Web page tested was the Microsoft homepage, located on the Web at (http://www.microsoft.com). This document was chosen as it was expected to show a bias towards the Microsoft family of Internet Explorer Web browsers. The page did not cause any parse errors, and thus was analyzable from its Web location. With the first version of the DRAT application, the runability analysis of the document indicated that it should be loaded without errors by all of the browsers under consideration, and correctly rendered by all browsers except Netscape 6. These results were found to be incorrect, as the page generated errors while loading with Netscape 4.07 and 4.79 browser versions, and did not provide the same functionality as it did on all of the Internet Explorer versions, all of which correctly rendered the document. Examining the source code for the Web page, several references to JavaScript objects not supported by the 4.x versions of Netscape were identified, correlating with the reported errors. These object references did not occur within a **script** block, but as attribute values in HTML tags. This prompted the design and implementation changes with respect to analysis of HTML tag attributes and generic runability typing of VBScript nodes.

Using the second version of the tool, the analysis of the document indicated that it could be correctly rendered only by the Internet Explorer browsers under consideration, and would also generate errors on all Netscape browsers. These results were confirmed by attempting to load the document with all browsers. The Netscape 4.x browsers indicated errors in the browser status area, and the rendered document did not exhibit all of the elements and functionalities that were present on the Internet Explorer browsers. Netscape 6 did not report any serious or fatal errors, but monitoring the browser's JavaScript console while loading the document showed several non-fatal errors occurred while parsing the embedded JavaScript code, due in all cases to references to objects in the Microsoft DOM that are not supported by Netscape 6.

6.2.3 Mozilla JavaScript Testcase

The Mozilla organization is dedicated to developing a standards-compliant Web browser and related tools, and the Netscape browser is derived from this effort. As noted in Chapter 5, the Mozilla test suites are fairly comprehensive, but do suffer from a bias

towards the Mozilla/Netscape implementation, or a lack of standards compliance on the part of the Internet Explorer browser family. The specific test case chosen was the test for the **setTimeout** function, a function that has been implemented by both browsers families from their earliest JavaScript implementations. Any incompatibilities in the document should thus be related to the script executed to perform the test rather than the function under test. The URL of the page is http://mozilla.org/quality/ngdriver/suites/javascript/win080.html, and was not analyzable from its Web location due to two instances of missing semicolons that would be inserted automatically by the interpreters incorporated in all of the tested browsers. The result of the analysis indicated that only the Netscape 6 Web browser should load the document without errors and render it correctly. Evaluating this result with individual browsers confirmed that Netscape 6 was the only browser to load the document without any errors. Internet Explorer 4.0 and Netscape 4.x browsers failed due to exception handling statements in the JavaScript code. The other Internet Explorer versions failed as a result of accessing properties of an object specified in the W3C DOM but not implemented by the Microsoft browsers.

6.2.4 Google Search Engine

Search engines are popular locations on the Web since they help users find information far more quickly than they could by browsing randomly. The Google search engine (http://www.google.com) has a simple and uncluttered main page with a minimal amount of scripting. Based on anecdotal personal usage of the site, I expected it to be compatible with all browsers, but the results of the analysis did not confirm that belief! As indicated in Table 6.1 on page 85, the Google website is correctly rendered and functional only on Netscape 6 browsers. I was suspicious of these results until I examined the page source in detail, and found that the JavaScript on the page made reference to the `getElementById` function that is implemented only by Netscape 6. The reference is used to limit execution of a particular code segments only to browsers implementing that function. These code segments were used to generate dynamic links to other documents based on information acquired at the time the triggering event occurred.

6.2.5 CBS Homepage

The CBS homepage (http://www.cbs.com) was the first of two documents analyzed that required editing to eliminate parse errors. The document was chosen as representative of an entertainment-based Website that an average user might use to check on upcoming broadcasts or find additional information on a telecast or news-related item. The document has nearly one thousand lines of JavaScript code, more than any other document tested, and had sixty-three missing semicolons that were manually inserted to enable the analysis. The results of the analysis were surprising, as DRAT reported that none of the browsers under consideration would correctly load the page, and all will generate errors of some kind. Checking against the browsers themselves confirmed the analysis. None of the browsers correctly rendered the document, either due to formatting problems like incorrectly positioned document elements, or to omission of one or more elements visible in other browsers. All seven browsers generated either fatal or non-fatal errors as well.

6.2.6 WRAL TV Homepage

WRAL is a local television station that has had a strong Web presence for several years with their homepage at http://www.wral.com. This page was chosen as representative of a well-established local news site, another frequent target of Web users. In the past year or so, I have noticed inconsistencies viewing the page on different browsers, making this document an interesting one to analyze with the DRAT application. Before the analysis could be performed, however, this document also required editing to insert missing statement-terminating semicolons into the JavaScript source code. Loading the edited document from the local Web server, DRAT reported that only Netscape 6 should correctly render the page, and that only Netscape 6 will load the document without errors. With respect to document loading without errors of any kind, the results were correct. However, the page had significant differences in layout and formatting across the various browsers. These problems may be the result of incorrectly written Cascading Style Sheets (CSS) [9] that are used to control document formatting on compatible browsers.

Examining the JavaScript source for this site, I could find only very limited use of browser-detection code, leading to my assumption that the effort was made, unsuccessfully, to produce a Web page that would be cross-browser compatible while using advanced

HTML and CSS features that are not implemented with compatible behavior by different browsers. This effort appears to have been made without attempting to dynamically generate document content and formatting based on the client browser loading the document, delegating the script code to event processing and dynamic content insertion, i.e. latest news headlines.

Table 6.1 summarizes the data generated during the execution of the Document Runability Analysis Tool as described above.

Table 6.1: Test Case Runability Results

Test Case	Correctly Rendering Browsers	Browsers Loading Without Errors
NCSU	Netscape4, Netscape4.5, Netscape6, MSIE4, MSIE5, MSIE55, MSIE6	Netscape4, Netscape4.5, Netscape6, MSIE4, MSIE5, MSIE55, MSIE6
Microsoft	MSIE4, MSIE5, MSIE55, MSIE6	MSIE4, MSIE5, MSIE55, MSIE6
Mozilla	Netscape6	Netscape6
Google	Netscape6	Netscape4, Netscape4.5, Netscape6, MSIE4, MSIE5, MSIE55, MSIE6
CBS.com	NONE	NONE
WRAL.com	Netscape6,	Netscape6

Chapter 7

Conclusions and Future Work

Electronic documents have become an integral part of modern life in our technological society, and while the vision of a "paperless" economy may still be a distant goal, it is unlikely that the role of electronic documents will decrease in the foreseeable future. Dynamic documents containing elements intended to be executed on the viewer's computing device bring a new level of flexibility, allowing an author to tailor the presentation of the document to the viewing platform and the user's preferences. They also present new challenges with respect to the integrity and security of these dynamic documents and the devices used to execute and display them. Traditional document analysis has focused on the content or text of the document, analyzing human-centric characteristics like spelling, grammar, or readability indices. Dynamic documents may be permitted to alter a user's computer, or to read information from it and transmit that data to another computer, possibly without the user's permission or knowledge. These challenges call for a new approach to document analysis.

This thesis has examined these problems, and presented a theoretical solution to one document analysis problem based on proven static type analysis techniques hitherto used to analyze computer programs. The implementation of this design, the Document Runability Analysis Tool, demonstrates the feasibility of programmatic analysis of HTML documents that may contain embedded and executable JavaScript source code to determine a set of Web browsers that will correctly render such a document. Based on the results of the test cases discusses in Chapter 6, such an analysis can also correctly identify this set of

browsers.

As there has been very little research in this area to date, there are many possible avenues open to exploration. There are obvious improvements that should be made to the DRAT application, in particular implementing automatic semicolon insertion as described in the ECMAScript language specification. Automatic semicolon insertion may occur only when the sequence of tokens on the top of the parser stack can be reduced to one of a specific set of JavaScript statements *and* the next token is a line terminator or a right brace (}) instead of the semicolon that would normally indicate the end of the statement. In all other cases, line terminators are treated as simple whitespace and ignored. A semicolon may also be inserted if the end of input is reached and the parser cannot reduce the stack to a *Program* without the insertion, or a within certain "restricted productions" as described in Section 7.9.1 of [2]. Implementation of this functionality is not trivial, and would require either significant modification of the generated parser currently used in the application, or a complete rewrite of the parser itself, perhaps in a form similar to that of the Rhino JavaScript interpreter [10].

A graphical user interface to the application that could provide additional functionality for the tool. For instance, a tree view of a document, highlighting browser-dependent code segments, could be incorporated into the application, allowing an author to quickly identify sections of a document that might be reworked to be more broadly compatible. A mechanism to allow the user to "customize" the list of built-in identifiers and browsers used in the analysis would also be a valuable addition to the tool. As noted previously, the ability to analyze JavaScript used as HTML tag attribute values was added in the second implementation version.

In a broader view, the techniques used in the design of DRAT can also be applied to other document analysis problems such as document integrity verification, analysis of metadata associated with and describing a document, and determination of document behavioral characteristics other than how it is rendered on a client device, i.e. resource requirements, client system read/write access, and calls to external executable components such as applets, ActiveX objects, or platform-native executable code.

In conclusion, the goal of this thesis was to demonstrate, theoretically and practically, that it is possible to successfully apply computer program analysis techniques to the new domain of dynamic documents. Chapter 4 presented the theoretical solution to a specific analysis problem, which was successfully implemented and tested, achieving this

goal.

Bibliography

[1] Aas, G. Perl HTML Parser Module. Web page. http://search.cpan.org/author/GAAS/HTML-Parser.

[2] Ang, M., Begle, C., Boyd, N., and others. ECMAScript Language Specification, Edition 3. Web, December 1999. http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM.

[3] Anon. JavaCC. Web page. http://www.webgain.com/products/java_cc.

[4] Anon. JavaCC HTML Parser. Web page. http://www.quiotix.com/docnloads/html-parser.

[5] Anon. W3C in 7 Points. Web page. http://www.w3c.org/Consortium/Points/.

[6] Anon. Web Style Sheets. Web page. http://www.w3c.org/Style.

[7] Anon. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Web page. http://www.w3c.org/TR/xhtml1.

[8] Anon. Browser Statistics. Web page, July 2002. http://www.w3schools.com/browsers/browsers_stats.asp.

[9] Bos, B. "Cascading Style Sheets". Web page, Feb 2002. http://www.w3.org/Style/CSS/.

[10] Boyd, N. Rhino: JavaScript for Java. Web page. http://www.mozilla.org/rhino/.

[11] Brady, M. "Document Object Model Conformance Tests". Web page, Dec 1999. http://xw2k.sdct.itl.nist.gov/BRADY/dom/index.html.

[12] Carpenter, J. "DOM Testcase Matrices". Web page, Apr 2000. http://mozilla.org/quality/browser/standards/dom1/tcmatrix/index.html.

[13] Desale, P. "Cleint Side Javascript Testcase Matrix". Web page, Apr 2000. http://mozilla.org/quality/browser/standards/javascript/tcmatrix/index.html.

[14] Flanagan, D. *JavaScript The Definitive Guide*, fourth ed. O'Reilly and Associates, Inc., Sebastopol, CA, 2002.

[15] Gagnon, E. Sable Research Group's Compiler Compiler. Web page. http://www.sablecc.org.

[16] Gagnon, E. Scriptonite ECMAScript Interpreter Project. Web page. http://www.scriptonite.org.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1998.

[18] Hantler, S. L., and King, J. C. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR) 8*, 3 (1976), 331–353.

[19] LeHegaret, P., Whitmer, R., and Wood, L. W3C Document Object Model. Web page. http://www.w3c.org/DOM.

[20] Lugrin, J.-M. FESI A Free EcmaScript Interpreter. Web page. http://home.worldcom.ch/jmlugrin/fesi/index.html.

[21] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[22] Musciano, C., and Kennedy, B. *HTML & XHTML The Definitive Guide, Fourth Edition*, fourth ed. O'Reilly and Associates, Inc., Sebastopol, CA, 2000.

[23] Peterson, C. "HTML 4.0 Transitional Matrix". Web page, Sep 2000. http://mozilla.org/quality/browser/standards/html/index.html.

[24] Plotkin, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sept. 1981.

[25] Raggett, D., LeHors, A., and Jacobs, I. "HTML 4.01 Specification". W3C Recommendation, December 1999. http://www.w3.org/TR/html4/.

[26] Schwartzbach, M. I. Polymorphic type inference. ls LS-95-3, BRICS, daimi, jun 1995. viii+24 pp.

Appendix A

The abstractDocument Structure

```
abstractDocument ::= docSeg list

docSeg    ::= htmlNode(tag, docSeg list, attribute list)
          |   scriptNode(sourceElt list, langAttr)
          |   noCodeNode

attribute ::= eventAttribute(attrName, sourceElt list)
          |   urlAttribute(url)
          |   otherAttribute(attrName, value)

url       ::= javascriptURL(sourceElt list)
          |   normalURL()

value     ::= javascriptEntity(sourceElt list)
          |   stringValue()

langAttr  ::= javascript
          |   jscript
          |   vbscript

sourceElt ::= funcDecl(identifier, optFormalParams, body)
          |   statement

optFormalParams ::=  identifier list

body      ::= sourceElt list

statement ::= blockStmt(statement list)
          |   variableStmt(varDecl list)
          |   emptyStmt()
          |   expressionStmt(expression)
```

```
          |   ifStmt(expression, statement)
          |   ifStmt(expression, statement, statement)
          |   do-whileStmt(statement, expression)
          |   whileStmt(expression, statement)
          |   forStmt(optExpr, optExpr, optExpr, statement)
          |   forStmt(varDecl list, optExpr, optExpr, statement)
          |   for-inStmt(expression, expression, statement)
          |   for-inStmt(varDecl, expression, statement)
          |   continueStmt()
          |   breakStmt()
          |   returnStmt()
          |   returnStmt(expression)
          |   withStmt(expression, statement)
          |   switchStmt(expression, caseBlock)
          |   labelledStmt(statement)
          |   throwStmt(expression)
          |   tryStmt(blockStmt, catch, finally)

varDecl   ::= identifier
          |   identifier assignExpr

optExpr   ::= ()
          |   expression

caseBlock ::= optCaseClauses optDefaultClause optCaseClauses

optCaseClauses ::=()
          |   caseClause list

caseClause ::= expression optStatementList

optDefaultClause ::=
          |   defaultClause

defaultClause ::= optStatementList

optStatementList ::= ()
          |   statement list

catch     ::= ()
          |   identifier blockStmt

finally   ::= ()
          |   blockStmt
```

```
expression ::= thisExpr
          |   identifier
          |   literalExpr(primitiveLiteral)
          |   stringLiteral(abstractDocument)
          |   arrayLiteral(expression list)
          |   objectLiteral(propVal list)
          |   parenExpr(expression)
          |   conditionalExpr(expression, expression, expression)
          |   newExpr(expression, expression list)
          |   arrayAccExpr(expression, expression)
          |   objPropAccExpr(expression, identifier)
          |   functionCallExpr(expression, expression list)
          |   functionDefExpr(identifier, optFormalParams, body)
          |   functionDefExpr(optFormalParams, body)
          |   assignExpr(expression, expression)
          |   binaryExpr(expression, BINOP, expression)
          |   unaryExpr(UNOP, expression)

primitiveLiteral ::= undefinedLiteral
          |   nullLiteral
          |   numberLiteral
          |   booleanLiteral

propVal   ::= identifier expression

BINOP     ::= &&
          |   ||
          |   ==
          |   !=
          |   OTHER-OP

UNOP      ::= !
          |   OTHER-UNOP
```

Appendix B

Built-in Identifiers and Runability Types

Table B.1: HTML Tag Name Groups

Group	Identifiers
HTML_0	A, APPLET, AREA, B, BASE, BIG, BLOCKQUOTE, BODY, BR, CAPTION, CENTER, CITE, CODE, DD, DIR, DIV, DL, DT, EM, EMBED, FONT, FORM, FRAME, FRAMESET, H1, H2, H3, H4, H5, H6, HEAD, HR, HTML, I, IMG, INPUT, ISINDEX, KBD, LABEL, LI, LINK, MAP, MENU, META, NOBR, NOEMBED, NOFRAMES, NOSCRIPT, OL, P, PARAM, PRE, S, SAMP, SCRIPT, SELECT, SMALL, SPAN, STRIKE, STRONG, STYLE, SUB, SUP, TABLE, TD, TEXTAREA, TH, TITLE, TR, TT, U, UL, VAR, WBR
HTML_1	BLINK, ILAYER, LAYER, MULTICOL, SPACER
HTML_2	KEYGEN
HTML_3	BASEFONT
HTML_4	BUTTON, COL, COLGROUP, DEL, DFN, FIELDSET, IFRAME, INS, LEGEND, TBODY, TFOOT, THEAD
HTML_5	BDO
HTML_6	OBJECT, OPTGROUP
HTML_7	OPTION, Q
HTML_8	BGSOUND, COMMENT, MARQUEE
HTML_9	ABBR, ACRONYM

Table B.2: HTML Runability Types by Group

Group	δ	η
HTML_0	β	β
HTML_1	Netscape4, Netsacpe4.5	β
HTML_2	Netscape4, Netsacpe4.5, Netscape6	β
HTML_3	Netscape4, Netsacpe4.5, MSIE4, MSIE5, MSIE55, MSIE6	β
HTML_4	Netscape6, MSIE4, MSIE5, MSIE55, MSIE6	β
HTML_5	Netscape6, MSIE5, MSIE55, MSIE6	β
HTML_6	Netscape6, MSIE6	β
HTML_7	Netscape6	β
HTML_8	MSIE4, MSIE5, MSIE55, MSIE6	β
HTML_9	$\emptyset$	β

Table B.3: Built-in Core JavaScript Identifier Groups

Group	Identifiers
JS_CORE_0	abs, acos, anchor, arguments, asin, atan, atan2, big, bold, callee, caller, ceil, charAt, concat, constructor, cos, E, escape, eval, exec, exp, fixed, floor, fontcolor, fontsize, fromCharCode, getDate, getDay, getFullYear, getHours, getMilliseconds, getMinutes, getMonth, getSeconds, getTime, getTimezoneOffset, getUTCDate, getUTCDay, getUTCFullYear, getUTCHours, getUTCMilliseconds, getUTCMinutes, getUTCMonth, getYear, getUTCSeconds, indexOf, isInfinite, isNaN, italics, join, lastIndexOf, length, link, LN10, LN2, log, LOG10E, LOG2E, match, max, MAX_VALUE, min, MIN_VALUE, NEGATIVE_INFINITY, parse, parseFloat, parseInt, PI, POSITIVE_INFINITY, pow, prototype, random, replace, reverse, round, search, setDate, setFullYear, setHours, setMilliseconds, setMinutes, setMonth, setSeconds, setTime, setUTCDate, setUTCFullYear, setUTCHours, setYear, setUTCMilliseconds, setUTCMinutes, setUTCMonth, sin, slice, setUTCSeconds, small, sort, split, sqrt, SQRT1_2, SQRT2, strike, sub, substr, substring, sup, tan, test, toGMTString, toLocaleString, toLowerCase, toString, toUpperCase, toUTCString, unescape, UTC, valueOf, Arguments, Array, Boolean, Date, Function, Math, Number, Object, RegExp, String
JS_CORE_1	apply, charCodeAt, global, ignoreCase, lastIndex, multiline, pop, push, shift, source, splice, unshift
JS_CORE_2	blink
JS_CORE_3	NaN, Infinity
JS_CORE_4	call, decodeURI, decodeURIComponent, encodeURI, hasOwnProperty, encodeURIComponent, isPrototypeOf, localeCompare, message, propertyIsEnumerable, toDateString, toExponential, toFixed, toLocaleDateString, toLocaleLowerCase, toLocaleString, toPrecision, toLocaleTimeString, toLocaleUpperCase, toTimeString, undefined

Table B.4: Built-in Core JavaScript Runability Types by Group

Group	δ	η
JS_CORE_0	β	β
JS_CORE_1	Netscape4, Netscape4.5, Netscape6, MSIE55, MSIE6	Netscape4, Netscape4.5, Netscape6, MSIE55, MSIE6
JS_CORE_2	Netscape4, Netscape4.5, Netscape6	Netscape4, Netscape4.5, Netscape6, MSIE4, MSIE5, MSIE55, MSIE6
JS_CORE_3	Netscape4.5, Netscape6, MSIE4, MSIE5, MSIE55, MSIE6	Netscape4.5, Netscape6, MSIE4, MSIE5, MSIE55, MSIE6
JS_CORE_4	Netscape6, MSIE55, MSIE6	Netscape6, MSIE55, MSIE6

Table B.5: Built-in Client-side JavaScript Identifier Groups

Group	Identifiers
JS_CLIENT_0	action, alert, alinkColor, anchors, appCodeName, applets, appName, appVersion, availHeight, availWidth, bgColor, blur, checked, clear, clearInterval, clearTimeout, click, close, closed, colorDepth, complete, confirm, cookie, defaultChecked, defaultSelected, defaultStatus, defaultValue, document, domain, elements, embeds, encoding, fgColor, focus, form, formName, forms, forward, frames, go, hash, host, host, hostname, href, images, innerHeight, innerWidth, javaEnabled, lastModified, length, linkColor, links, location, lowsrc, method, moveBy, moveTo, name, navigator, onabort, onblur, onchange, onclick, onerror, onfocus, onload, onreset, onsubmit, open, options, outerHeight, outerWidth, parent, pathname, platform, plugins, port, print, prompt, protocol, reload, replace, reset, resizeBy, resizeTo, screen, screenX, screenY, scroll, search, select, selected, selectedIndex, self, setInterval, setTimeout, src, status, submit, target, text, title, top, type, URL, userAgent, value,vlinkColor, width, window, write, writeln
JS_CLIENT_1	referer
JS_CLIENT_2	ABORT, above, background, below, bgColor, BLUR, bottom, CHANGE, CLICK, clip, DBLCLICK, DRAGDROP, ERROR, FOCUS, hidden, KEYDOWN, KEYPRESS, KEYUP, layers, layerX, layerY, left, LOAD, load, modifiers, MOUSEDOWN, MOUSEMOVE, MOUSEOUT, MOUSEOVER, MOUSEUP, MOVE, moveAbove, moveBelow, moveToAbsolute, offset, pageX, pageY, parentLayer, RESET, RESIZE, right, SELECT, siblingAbove, siblingBelow, SUBMIT, UNLOAD, visibility, which, zindex
JS_CLIENT_3	back, description, disableExternalCapture, enabledPlugin, enableExternalCapture, filename, forward, home, length, locationbar, menubar, mimeTypes, pageXOffset, pageYOffset, personalbar, plugins, refresh, scrollbars, statusbar, stop, suffixes, toolbar, visible
JS_CLIENT_4	x, y
JS_CLIENT_5	captureEvents, getSelection, handleEvent, language, releaseEvents, routeEvent
JS_CLIENT_6	border, hspace, vspace
JS_CLIENT_7	availLeft, availRight, pixelDepth
JS_CLIENT_8	height
JS_CLIENT_9	java, netscape, Packages, packages, sun
JS_CLIENT_10	cookieEnabled, scrollBy
JS_CLIENT_11	className, getAttribute, id, innerHTML, lang, onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, removeAttribute, setAttribute, style, tagName, title
JS_CLIENT_12	history, opener, scrollTo
JS_CLIENT_13	activeElement, all, charset, children, defaultCharset, expando, parentWindow, readyState, systemLanguage, userLanguage
JS_CLIENT_14	clientInformation, contains, elementFromPoint, event, innerText, insertAdjacentHTML, insertAdjacentText, navigate, offsetHeight, offsetLeft, offsetParent, offsetTop, offsetWidth, onhelp, outerHTML, outerText, parentElement, scrollIntoView, sourceIndex
JS_CLIENT_15	altKey, button, cancelBubble, clientX, clientY, ctrlKey, fromElement, keyCode, offsetX, offsetY, reason, returnValue, shiftKey, srcElement, srcFilter, toElement
JS_CLIENT_16	hasFeature, doctype, implementation, createComment, createAttribute
JS_CLIENT_17	getElementsByTagName
JS_CLIENT_18	getElementById
JS_CLIENT_19	documentElement, createElement, tagName, getAttribute, setAttribute, removeAttribute, nodeName, parentNode, firstChild, lastChild, previousSibling, nextSibling, insertBefore, replaceChild, removeChild, appendChild, hasChildNodes, item
JS_CLIENT_20	createDocumentFragment, createTextNode

Table B.6: Built-in Client-side JavaScript Runability Types by Group

Group	δ	η
JS_Client_0	β	β
JS_Client_1	Netscape4, Netscape4.5, Netscape6, MSIE55, MSIE6	Netscape4, Netscape4.5, Netscape6, MSIE55, MSIE6
JS_Client_2	Netscape4, Netscape4.5	Netscape4, Netscape4.5
JS_Client_3	Netscape4, Netscape4.5, Netscape6	Netscape4, Netscape4.5, Netscape6
JS_Client_4	Netscape4, Netscape4.5, MSIE4, MSIE5	Netscape4, Netscape4.5, MSIE4, MSIE5, MSIE55, MSIE6
JS_Client_5	Netscape4, Netscape4.5, Netscape6	Netscape4, Netscape4.5, Netscape6, MSIE55, MSIE6
JS_Client_6	Netscape4, Netscape4.5, MSIE4, MSIE5, MSIE55, MSIE6	β
JS_Client_7	Netscape4, Netscape4.5, Netscape6	β
JS_Client_8	Netscape4, Netscape4.5	β
JS_Client_9	Netscape4, Netscape4.5	Netscape4, Netscape4.5, Netscape6
JS_Client_10	Netscape6, MSIE4, MSIE5, MSIE55, MSIE6	β
JS_Client_11	Netscape6, MSIE4, MSIE5, MSIE55, MSIE6	Netscape6, MSIE4, MSIE5, MSIE55, MSIE6
JS_Client_12	Netscape6, MSIE55, MSIE6	Netscape6, MSIE55, MSIE6
JS_Client_13	MSIE4, MSIE5, MSIE55, MSIE6	β
JS_Client_14	MSIE4, MSIE5, MSIE55, MSIE6	MSIE4, MSIE5, MSIE55, MSIE6
JS_Client_15	MSIE4, MSIE5	MSIE4, MSIE5, MSIE55, MSIE6
JS_Client_16	Netscape6, MSIE6	Netscape6, MSIE6
JS_Client_17	Netscape6, MSIE55, MSIE6	β
JS_Client_18	Netscape6, MSIE5, MSIE55, MSIE6	β
JS_Client_19	Netscape6, MSIE5, MSIE55, MSIE6	Netscape6, MSIE5, MSIE55, MSIE6
JS_Client_20	Netscape6, MSIE6	Netscape6, MSIE55, MSIE6