Declarative Specification of Web-based Integrated Development Environments

Sverre Rabbelier
Declarative Specification of Web-based Integrated Development Environments

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sverre Rabbelier
born in Rotterdam, the Netherlands

TUDelft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

www.ewi.tudelft.nl
Abstract

This thesis describes a set of tools and abstractions that facilitate the development of an Integrated Development Environment (IDE) that runs in the browser. We present a plugin for ACE, the Ajax.org Cloud9 Editor, that adds support for languages defined with SDF. The plugin supports vital editor support such as syntax highlighting and validation. Language syntax can be declaratively defined with no concern for the underlying implementation. This approach is realized by compiling a Java library to JavaScript using the Google Web Toolkit. Besides syntactic support, semantic checks are supported by means of the Stratego language. A compiler backend for the Stratego compiler is developed targeting the JavaScript language.
First of all, I would like to thank my supervisor, Eelco Visser, for finding me a project closely related to my interests. I want to thank Danny Groenewegen, Zef Hemel, and Lennart Kats, who explained to me the many ‘hysterical raisins’ behind the tools I have used in the past months. I am indebted many a cup of coffee to my research group. I would like to thank Karl Trygve Kalleberg, for helping me understand his previous work. Finally, many thanks to my friends and family for supporting me throughout my career as a student.

Sverre Rabbelier
Delft, the Netherlands
November 22, 2011
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

1.1 Integrated Development Environments 1
1.2 To the browser 3
1.3 Problem statement 5

2 Syntax 9

2.1 JavaScript 11
2.2 Porting SGLR to JavaScript 14

3 Editor services 19

3.1 Cloud9 21
3.2 The ACE Plugin Architecture 22

4 Semantics 27

4.1 Stratego 28
4.2 Design 30
4.3 Static linking 31
4.4 s2js 32

5 Performance 35

5.1 Parsing parse tables 36
5.2 Reintroducing term sharing to parse tables 38
5.3 Optimizing variable naming 39
5.4 Benchmarks 41
List of Figures

1.1 IDE features ................................................................. 1
1.2 Overall architecture ....................................................... 6

2.1 Parser architecture .......................................................... 11
2.2 require.js dependency management ....................................... 12
2.3 require.js legacy format ..................................................... 12
2.4 JavaScript scoping ............................................................ 13
2.5 JSSGLR architecture ........................................................ 15
2.6 GWT linker ................................................................. 16
2.7 GWT boilerplate code ...................................................... 17
2.8 GWT initialization code ..................................................... 17

3.1 Cloud9 ................................................................. 20
3.2 Editor architecture .......................................................... 22
3.3 Analyzer architecture ....................................................... 22

4.1 Semantic error ............................................................. 27
4.2 Semantics architecture ...................................................... 28
4.3 Compiler design ............................................................. 28
4.4 Labeled break statement ................................................... 33
4.5 Sequence operator .......................................................... 33
4.6 Guarded left choice operator ............................................. 34

5.1 Stripped down parse-table .................................................. 35
5.2 JSSGLR parseArgs function .............................................. 37
5.3 JSSGLR parseArg function ............................................... 37
5.4 Constructing a no-argument 'class' constructor ......................... 38
5.5 Constructing a n-arg 'goto' constructor .................................. 38
5.6 Storing parse-tree terms in variables .................................... 38
5.7 Using short aliases for parse-tree names .................................. 39
5.8 Using short aliases for parse-tree names and constructors ................ 39
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Eliminating whitespace from parse-tree terms</td>
<td>40</td>
</tr>
<tr>
<td>5.10</td>
<td>Parse tree size and compilation time for varying cut off points</td>
<td>40</td>
</tr>
<tr>
<td>5.11</td>
<td>Before and after of a minimal parse tables, minus parse states</td>
<td>42</td>
</tr>
<tr>
<td>5.12</td>
<td>Performance metrics of ACE plugin as measured by time taken to reach various points in the execution</td>
<td>43</td>
</tr>
<tr>
<td>5.13</td>
<td>Metrics of tested languages, showing the parse table size after gzip compression, the same for the JavaScript format and several language properties indicative of language complexity</td>
<td>43</td>
</tr>
<tr>
<td>5.14</td>
<td>WebDSL snippet used to benchmark the plugin</td>
<td>44</td>
</tr>
<tr>
<td>5.15</td>
<td>Runtime performance metrics of JSSGLR parser, all measurements in ms</td>
<td>44</td>
</tr>
<tr>
<td>5.16</td>
<td>Stratego snippet used to benchmark the Stratego compilers</td>
<td>45</td>
</tr>
<tr>
<td>5.17</td>
<td>Raw performance of programs generated by the Stratego compilers, measured with a recursive implementation of $Fibonacci(n)$ for a varying input $n$</td>
<td>46</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The goal of this thesis is to provide a set of tools and abstractions that facilitate the development of an Integrated Development Environment (IDE) that runs in the browser. We will first motivate and describe what an IDE is and why development is moving to the browser. We will then explain the problem statement of this thesis and detail the structure of this thesis.

1.1 Integrated Development Environments

An Integrated Development Environment (IDE) is a program that a software developer can use to more efficiently write code. An IDE integrates many components in a uniform fashion, reducing the amount of mode switching required by the programmer. There are many different IDEs in use, and the feature set is different for each one. Commonly, an IDE consists of an editor in which code is written, augmented by editor services.

The most universally provided feature is syntax highlighting, which assigns different colours to specific aspects of the language used by the programmer, such as can be seen in Figure 1.1(a). This functionality is often accompanied by displaying error and warning
1. Introduction

Markers for syntactic and semantic issues. An outline view, which displays an overview of the syntactic elements present in the current file, usually allows quick navigation to these elements by clicking them, see Figure 1.1(b). A closely related feature is code folding, in which syntactic elements are hidden from the programmer, instead represented in an abstract way, such as a + symbol. Another common feature is autocompletion, where the programmer writes the beginning of a code fragment, and the code (possibly after further input from the programmer), completes the fragment.

Implementing editor services

There are several possible solutions to implementing these editor services. Hand-written tools are written for a specific use cases, all assumptions and design decisions are made with this use case in mind. Any other use case will generally have different requirements, causing little reuse and low abstraction of hand-written tools. Regular expressions (regexp), a pattern written in a formal language understood by a regexp processor, while concise, are limited in the languages they can parse. The concepts of both hand-written tools and regexp can be ported with relative easy to web environment.

Parser that are generated from a grammar definition, are more powerful in the languages they can accept, and provide a higher level of reuse than hand-written tools. However, these generators are far from trivial to use properly and while they provide a higher level of abstraction than hand-written tools, the programmer must still keep in mind many implementation details. To facilitate the development of these services, a high level of abstraction is needed, while retaining performance comparable with existing solutions.

1.1.1 Spoofax

On the desktop, a framework to address this lack of a high-level abstraction already exists in the form of the Spoofax project [23]. Spoofax is a language workbench that integrates with Eclipse, it focusses on providing the developer tools to create new languages. With Spoofax developers can write a language definitions, from which Spoofax generates an Eclipse plugin specially tailored for that language. The generated plugin is initially only capable of providing syntactic editor services, such as syntax highlighting, an outline view, code folding, and syntactic error messages.

Additional services can be added that analyse the abstract syntax tree (AST) and return relevant information to the plugin. For example, semantic analysis can be added by writing checks which produce errors that can be displayed in the editor either as messages for a certain line, token, or file. Another possible feature is that of autocompletion, the plugin is provided with a partial expression for which it must offer completions.

Spoofax provides a superset of the functionality generally present in an IDE, extending the environment with tools aimed specifically at developing new languages.

Tools

Spoofax uses several tools to implement features useful in any IDE, many of which we will use in this thesis.
The Syntax Definition Formalism (SDF) [35] is used to define grammars. SDF can be used to define the languages supported by a regular IDE. In conjunction with SDF, Spoofax uses a scannerless generalised left-to-right parser (SGLR) [22]. There are many benefits to using an SGLR parser, most significantly is that it allows arbitrary embedding of languages. For example, with SGLR editor support for Java code embedding JavaScript code is possible. The output of an SGLR parser can, after conversion to an AST by an *imploder*, be manipulated with Stratego [8], a language designed specifically for the transform and analysis of ASTs. Lastly, the IMP framework [9] is used to abstract from the Eclipse implementation.

**Usage**

Spoofax is a proven platform, several non-trivial languages have been developed using it. The largest consumer of Spoofax is WebDSL [36, 13, 16, 18, 14, 15], a language to create web applications. WebDSL has been used to write several large-scale applications that would have taken hundreds of thousands lines of code to write in Java. Mobl [17], is related to WebDSL, but focusses on mobile web applications. Development for both languages is done using an IDE generated by Spoofax.

1.2 To the browser

For various reasons the browser has recently received a lot of interest as platform for the development of applications. Many applications that were previously written as desktop application, are now implemented as websites. This change begs the question, why do we want to move from the desktop to the web?

1.2.1 Web vs desktop

There are several reasons to target the browser instead of the desktop when developing a new application. There is virtually no perceived startup time for many web application as the time it takes start the browser is not attributed to the application. Desktop programs are one of many applications running on a system, while web servers are often dedicated and can often immediately start serving a cached response.

While most desktop applications are stuck in the old paradigm of menu bars and context menus, user’s expect a much simpler interface from web applications, resulting in an easier to use application. Most innovation in user interface (UI) design is happening on the web due to the relative ease of HTML compared to desktop UI toolkits.

Many desktop applications face the problem of getting their users to upgrade to the latest version. Developers are forced to either automatically download updates, or to nag the user until they download the latest version. The former does not sit well with all users, and the latter is of limited effectiveness. Web applications on the other hand are updated on the server side, no interaction from the user required. If the deployment process is set up properly a web developer can deploy an update multiple times per day, a luxury a desktop application cannot afford.
Native applications usually have far-reaching access to the user’s computer. If the user installs a malicious application generally has at least access to all files. Web applications on the other hand generally do not have any access to the user’s computer. Newer browsers incorporate a sandbox model in which a website can not even access other pages within the browser.

With the rise of the web new business models have been made popular and are commonly accepted. Most websites are free but are paid for by ads displayed on every page. Due to the large potential reach of the web, even a small profit per visitor can be sufficient to sustain a webpage. This larger reach is also attractive to the developer, as their work reaches a large audience.

More and more web applications are working together by means of an application programming interface (API). By exposing an API a web service enables another application to build on the work it performs. These APIs can be used on the client side as well as on the server side. Web applications usually do not have very high performance requirements. Due to the client-server architecture of the web it is easy to move heavy computations to the server, which may have more computing power available than the client.

Finally, web applications run on any operating system (OS). While not every operating system will support the same browsers, users will be able to visit a website regardless of their OS.

1.2.2 Constraints

Even in the face of these reasons to develop applications for the web, not all development has abandoned the desktop. There are several constraints introduced by the web environment which limit its adoption. Desktop applications can require the user to install their application from a physical medium while developers of web applications must take into account that the available bandwidth is limited. Similarly, while desktop applications can present the user with a progress dialog, users of web applications expect fast load times and quick responses from a web page.

The implementation of web applications is limited by features provided by the JavaScript language. Additionally, implementations are limited to a low level of abstraction. JavaScript is entirely single threaded, it depends heavily on events and callbacks to simulate asynchronous behavior. As a result any call to an action that might block must be implemented as callback. The JavaScript runtime and object model limits the scalability of all applications as there is no way to control memory allocation or CPU resources.

While almost every operating system allows users to access the web one way or another, thus solving the problem of cross-platform interoperability, this introduces a new problem, that of cross-browser compatibility. There are many different renderers in use, as a result of which developers must test their site across different browsers. Sometimes the same browser behaves differently on different operating systems, adding to the number of browsers that must be tested against.

Finally, users are dependant on the company that runs the web server for service. If a company ends support for a desktop application, the user may continue to use the last avail-
able version. Should a company decide to end support for a web service and discontinue hosting, users are forced to stop using the service immediately.

1.2.3 Why switch to the web now?

Considering these constraints, why are web applications becoming more popular? The answer lies in several recent developments that allow developers to overcome these constraints.

Web workers have recently been added which allow computation to be performed in a separate thread. Several new libraries, such as require.js which adds a form of modules to JavaScript, as well as new web standards reduce cross-browser compatibility concerns. In addition to the browser, JavaScript is now also usable on the server side by means of the node.js project.

There are several easy-to-use hosting services from reliable providers such as Amazon, Heroku, and Rackspace that make it easier for developers to host their applications online. While download size is still a constraint, steady improvements in the connection speed makes larger downloads more acceptable. Meanwhile faster JavaScript engines (V8, TraceMonkey, Carackan, Chakra) enable writing more CPU-hungry applications. This combined with more powerful browser features such as WebGL, which provides websites access to the GPU, and off-line support bring many of the strengths of the desktop to the web.

Finally, the ease of deployment on the web means instant access to everybody around the world, at virtually no cost to the company hosting the web application. With the continuously increasing access to the web this is a very strong argument indeed.

1.3 Problem statement

In ‘Adinda: a knowledgeable, browser-based IDE’ [12], van Deursen et al. propose a browser-based IDE. This browser-based IDE will facilitates informal communication and collaboration, thereby increasing collaboration and leverages knowledge of other team members. Atwood’s Law [2] meanwhile dictates that: ‘any application that can be written in JavaScript, will be eventually be written in JavaScript’.

The goal of this thesis is to provide a set of tools and abstractions that facilitate the development of an Integrated Development Environment (IDE) that runs in the browser.

Constraints

Running in the browser has several constraints, developing an IDE for the web poses additional constraints on top of those. The parsing required to provide editor services is CPU intensive work, while performance in the browser is more constrained than on the desktop. Due to the limited availability of resources such as shared libraries, development of large programs, such as IDEs, is more complicated. While there are libraries that would aid the implementation of an IDE, these libraries are not implemented in JavaScript. Even the libraries that are implemented in JavaScript cannot be used without taking into account the
cost involved with adding more libraries due to the increased download size. In short, there are very little to no abstractions available to develop IDEs that target JavaScript

Other approaches

Several approaches to further the state of the art on the browser have been tried previously. The Bespin project, later Skywriter, while mostly an editor, set the standard for browser-based editing. Narcissus, a JavaScript that parses JavaScript allowed developers to benefit from syntactic and semantic warnings for JavaScript code. Later, Cloud9, a platform with a plugin architecture, integrated Bespin into its editor, ACE.

While all these approaches improved the availability of tools in the browser, they all share a common problem. These “state of the art” web editors implement services are either by hand or using regexp. All current tools are developed at the wrong level of abstraction. The developer of an IDE for the web must handle too many implementation details that are unrelated to the language being developed for.

Solution

We address the problem of insufficient abstraction by further separating concerns. The Cloud9 project already takes a step in the right direction by separating its editor from the development environment itself. We will further split out the language specific elements from the editor. Specifically, we will reuse abstractions from desktop environment that are known to work and port them to the browser. Additionally, we will develop a compiler-back end for JavaScript that will allow us to reuse the Stratego language.

This solution takes the form of a plugin for ACE, such that many different language components can be plugged in without changing any of the other editor code. With this plugin meta-programmers will have a declarative way of specifying editor services for arbitrary programming languages. These editor services include code completion, code navigation, syntax highlighting, syntax checking, type checking, etc.

Architecture

The architecture of our plugin consists of several components, as can be seen in Figure 1.2. The parser component converts the input into an AST, producing a token stream as byproduct. From the token stream syntactic errors can be extracted, while from the AST semantic errors can be derived. The semantic analysis of the AST is performed by a component written in Stratego. To compile the Stratego component, the Stratego-to-JavaScript compiler is
used. On the editor side, a plugin accepts the errors and warnings and converts them into a
format the editor understands.

In this thesis we have implemented each of these components. The parser is discussed
in Chapter 2, the editor component in Chapter 3, and the semantic analysis is discussed in
Chapter 4. In Chapter 5 we will discuss ways to improve the performance of the plugin,
while Chapter 6 discusses related work. Finally, Chapter 7 concludes this thesis. Not dis-
cussed in this thesis is the communication between semantic analyzer and plugin, which is
left as future work.
There are several approaches to providing editor services. Parsing a document to an AST is very effective, since the AST can be used to implement most services.

Initially, tools to parse documents were hand written, a very low-level approach. Manually written parsers are not descriptive, they require the programmer to write tools, rather than grammars. Instead of focussing on the grammars being designed, the programmer is forced to take into account the capabilities of the tools they are using. In this way, the paradise of pure and declarative syntax definition is lost [21].

With the arrival of regular expressions (regexp), the definition of an expression was independent of the implementation. New regular grammars could be parsed without requiring a new implementation. Additionally, a regexp can describe a regular grammars very concisely. As the name suggests though, regexp can only parse regular grammars. In practice, this means that regexp, without extensions at least, cannot parse expressions that include arbitrarily nested braces, etc. Additionally, to use regexp for anything but keyword based syntax highlighting, workarounds are needed, such as combining a regexp parser with a state machine.

Using parser generators, the same level of abstraction as with regexp could be achieved while not being restricted to regular grammars. From a language definition, a parser could be generated for many grammars that previously required manual implementation. Generally, a parser generated this way will perform an action when certain elements of the grammar are encountered. While these parser can produce ASTs, this must be done manually, by building the AST in the action of each grammar element. Again the level of abstraction is not high enough, code must be written to construct an AST for each created language.

Left to right parsing

In 1965, Donald Knuth published a paper [25] introducing the concept of LR grammars. LR grammars are defined as these grammars that can be recognised by a parser that reads the input from left to right. In addition, this parser may not reconsider a previous decision, this is known as deterministic parsing. However, an LR(k) may look ahead at the next k characters when making a decision. The advantage of such a language is that it can be
implemented by an algorithm whose time complexity is linearly proportional to the length of the input. Knuth demonstrates that such an algorithm can be derived automatically from the definition of the language.

A crucial part of the LR parsing algorithm is the parse table, which is derived from the language definition. The parse table is a table that contains for each situation an instruction on what to do next. The parsing algorithm reads the input, and, using the parse table, decides what to do. There are several actions that the parser may take while reading the input, but most importantly, at a certain point the parser will decide to accept or reject the input.

**Scannerless LR parsing**

After the introduction of LR parsing, much research has been done in extending the usefulness of such parsers. Traditional LR parsers consist of two components: a lexer and a parser. The lexer pre-processes the input before feeding it to the parser in a previously agreed upon format. Salomon and Cormack discuss the disadvantages of this approach in their 1989 paper [29]. Existing solutions to defining LR(\(k\)) grammars require working around the \(k\) look-ahead restriction. Their solution is a language description that describes the syntax of the language at the character level: scannerless LR (SLR) parsing. A disadvantage of this approach is that it often leads to ambiguous grammars.

**Generalized LR parsing**

LR-parsing is restricted to those languages that are deterministic and unambiguous, but SLR parsing introduces grammars that are ambiguous. In order to parse such ambiguous grammars, the concept of LR parsing must be extended. The 1984 paper [34] by Masaru Tomita defines a parallel parser that can handle such ambiguous grammars. The goal of this parser is to handle human language, which is inherently ambiguous. It is a further evolved version of this algorithm that is now known as generalized LR parsing. GLR parsing possesses all traits of LR-parsing, yet it can parse nondeterministic and ambiguous grammars.

**Scannerless Generalized LR parsing**

Combining both the concepts of scannerless and generalized LR parsing, Visser introduces Scannerless Generalized LR (SGLR) parsers in his 1997 thesis [35]. A SGLR parser uses the fact that GLR parsing can handle the ambiguous grammars created by a scannerless parsers. Context-free (CF) grammars are closed under composition, the result of embedding one CF grammar in another CF grammar is a CF grammar. Regular GLR parsers are unable to freely compose context-free grammars due to the fact that the lexers for different grammars are often incompatible. The main advantage of an SGLR parsers is that it enables the free composition of context free languages.

**Syntax Definition Formalism**

The Syntax Definition Formalism (SDF) is a language that can be used to define languages. A SGLR parser can be derived automatically for any language definition written in SDF.
More precisely, a generic SGLR parser exists that can parse any document given a parse table derived from an SDF definition. Using SDF, a document can be parsed given only a declarative language definition. SDF can parse any context-free grammar which is suitable for parsing most languages. The grammar is used to define both the concrete syntax (keywords, etc.) and the abstract syntax (used to perform analysis to provide other services).

Motivation

Because grammars defined in SDF are SGLR grammars, embed other SDF languages is possible. SDF is declarative, implementation details (such as left-recursion) are handled by the parser. To achieve the desired level of abstraction, to regain paradise, a declarative language definition formalism such as SDF, usable in the browser environment, is needed.

2.1 JavaScript

JavaScript is the only language that web applications can use in the browser. If we are to implement an SGLR parser in the browser, it will have to be written in JavaScript. This implies that the implementation of an SGLR parser that is usable in the browser will have to be written in JavaScript. The JavaScript language has many traits that affect the development of such a parser. We will now discuss the properties of JavaScript that are relevant to this thesis.

The problem with JavaScript

Most languages are developed in a fairly central way, their evolution is dictated primarily by changes made to their compiler. JavaScript, on the other hand, has many different compilers, each browser chooses their own. In addition, JavaScript is an interpreted language rather than a compiled language. The code in websites that use JavaScript is interpreted by each browser that renders the page, as opposed to the code being compiled once, and then distributing the result of that compilation. There are then not only different compilers for each browser in use, but also different compilers for the different versions of those browsers. As a result, it is difficult to make any changes to JavaScript, as there are many different engines that implement the language. The JavaScript language has several restric-
2. Syntax

```javascript
//my/shirt.js now has some dependencies, a cart and inventory
//module in the same directory as shirt.js
define(['./cart', './inventory'], function(cart, inventory) {
    //return an object to define the "my/shirt" module.
    return {
        color: "blue",
        size: "large",
        addToCart: function() {
            inventory.decrement(this);
            cart.add(this);
        }
    }
});
```

Figure 2.2: require.js dependency management

```javascript
define(function(require, exports, module) {
/*
   legacy code here
*/
});
```

Figure 2.3: require.js legacy format

...tions that make it a difficult language to develop for, which have not been solved due to the
issues outlined above.

2.1.1 Modules

Code written in JavaScript can be split up into multiple files, which can then be included
by the browser. However, the language does not allow modules to declare dependencies
on other modules. Instead, any dependencies a module has must be included in the main
page before it. Various JavaScript libraries have been written that attempt to address this
problem, require.js is one of them. The require.js library works by relying on the ability
of scripts to dynamically include other files by inserting a new `<script>` element in the
<html> of an html page.

Instead of including all depended upon files in the main page, a page that uses require.js
includes only the require.js file. The page instructs the framework which file to load as the
main module. This main module functions like any other JavaScript file, with the exception
that it defines all its dependencies before it starts execution. This way, the require.js library
can determine which scripts have to be included, and in what order. By the time that the
execution of the main module starts, all dependent scripts have finished loading, and are
passed as argument to the main module. An example from the require.js documentation [3]:

In Figure 2.2 the shirt module depends on the cart and inventory modules that
window.bar = window.bar || {};
window.bar.foo = function() {};

**Figure 2.4: JavaScript scoping**

are files in the same directory. Both dependencies are passed as argument to the module initialization function, which then uses it to define the modules `addToCart` function. The return value of the initialization value is passed to modules that depend on this module.

While this approach works well enough, it places several restrictions on the developer. Firstly, the system works best when all dependencies follow the `require.js` module format, while most commonly used libraries do not adhere to this format. Secondly, some care must be taken not to pollute the global namespace when defining a module. Luckily, `require.js` has a special format for legacy code that is simply meant to be exposed as library, as shown in Figure 2.3, that addresses the first problem.

**require.js module loading**

A `require.js` module is loaded by calling a function defined by the `require.js` library to register a module. Each module passes the list of modules that it depends on to the registration call. When all modules have communicated their dependencies `require.js` uses an algorithm to determine in which order modules should be loaded. As a module is loaded its callback is invoked, passing it a reference to all the modules it depends on. At this time the module can run all its initialization code, optionally making use of the modules it declared to depend on. Once a module is loaded, control is passed back to the `require.js` library, which continues with loading the next module.

### 2.1.2 Namespaces

As the amount of code involved in a project increases, the need for namespaces increases. The C language demonstrates what happens when no namespacing is possible at all: symbols are generally not exported, and when they are, care must be taken to prefix them such as not to pollute the global namespace. While the situation is not quite as dire in JavaScript, it is not much better either. If proper care is taken to follow certain patterns, it is possible to refrain from polluting the global namespace. Sadly, it is very easy to accidentally stray from these patterns. For example, defining a function `foo` as `function foo() {}` results in the creation of a global symbol `foo`. If one intends to scope the `foo` symbol, one must first create a namespace.

In Figure 2.4 the `foo` function is scoped in the `bar` namespace. This creates an object named `bar` in the global namespace (if it does not already exist), and assigns a function to its `foo` attribute. A subtle downside of this approach is that the function now bound to the `foo` property of the `bar` object is unnamed. It is possible to pass a reference to this function around, as in JavaScript functions are first class citizens. If one encounters such a reference during debugging it is impossible to determine what name this function was originally assigned, that is, there is no way of knowing that this is the `foo` function.
2. Syntax

2.1.3 Web workers

The JavaScript language is inherently single threaded. While some features (such as the `setTimeout` function) can be used to simulate threads to some degree, these are useless when faced with a heavy computational task. Due to JavaScript's single threaded nature, any task that involves a lot of heavy computation will block execution of all other functions. As a result, the browser will appear to 'hang' while executing such code, providing a less than optimal user experience.

Web workers are the solution made to solve this problem. The web worker API can be compared with that of threads in languages such as C or Java. When a new worker is created, a new thread is spawned independent of the main thread, any computation that performed on this separate thread does not affect any other thread. This property makes web workers particularly useful for heavy computation that is asynchronous in nature. Web workers do not share memory with other threads, all communication occurs by means of channels.

2.1.4 Google Web Toolkit

The Google Web Toolkit [10] is generally used to write AJAX applications in Java that are then compiled to JavaScript. Applications developed in GWT are written in Java and are compiled to JavaScript by the GWT compiler. The resulting JavaScript applications make heavy use of Ajax to make the application perform well. Instead of manually porting a Java application to JavaScript, GWT can be used to perform the port.

2.2 Porting SGLR to JavaScript

To bring the power of parsing arbitrary text using the SGLR algorithm to JavaScript we need an SGLR parser. Currently, only a C and Java implementation of such a parser exist. A previous effort ¹ has been made to derive a JavaScript SGLR (JSSGLR) parser from the Java SGLR parser by using the Google Web Toolkit (GWT). At that time, the only way to compile an application with GWT was to wrap it in a GWT application. Thus, compiling the JSSGLR parser as a standalone library was not an option, rather application that uses it had to be compiled with GWT.

2.2.1 Wrapping

The main application, the one using the JSSGLR library, would itself make little to no use of the features that GWT offers. Instead, the application would indicate that it depends on JSSGLR, and GWT would take care of integrating it into the application. The early version of JSSGLR that used this approach had a complicated compilation process. While there is support in GWT for the inclusion of raw JavaScript in a GWT application, it is not the expected modus operandi. Developers have to jump through hoops to develop their application just to use the JSSGLR library.

¹https://bitbucket.org/karltk/skywriter-gwt by Karl Trygve Kalleberg
In order for JSSGLR to be generally useful it must be possible to run it without having the main thread controlled by GWT. Not only that, it is highly desirable for JSSGLR to be a library that is compiled with GWT only once. Due to the long compilation time of JSGLR under GWT and the non-trivial set-up, adoption would be hindered by a lack of a standalone library version of JSSGLR. Since traditional GWT applications are expected to have most, if not all, of the application written in Java, the default GWT setup does not allow compiling the application as a standalone library.

**Taking control of the main thread**

To address this we take inspiration from the Speedtracer project, which faced the same problem. Speedtracer is a project that makes use of GWT in order to instrument other GWT applications. Part of Speedtracer is a separate web worker that takes care of some intensive processing. The worker thread does not have access to any of the environment of the main thread, as such it needs to be a separate compile target that includes all the relevant code. By means of a simple extension to the GWT linker, this web worker is compiled as a separate target.

By default GWT uses a special naming scheme that allows many modules to be used together, but complicates deployment when only one module is needed. Figure 2.6 shows a linker which instructs GWT not to use any of its name mangling features (modules have no prefix or suffix). The generated script has a name derived from the contents of the file using a cryptographic hash function. This strong name (75[...8A.cache.js) is only relevant to GWT-enabled programs, where it is used to allow files to be cached forever. In order to provide JSSGLR as a proper library we simply rename the file to something more appropriate (jssglr.js).

**Custom header**

The custom linker depends on the use of a template (DedicatedWorkerTemplate.js) that takes care of initializing the GWT environment. Before this initialization can occur, certain variables must be correctly initialized. In particular, many GWT internal functions rely on the existence of a wnd variable. The wnd variable is used to abstract away the implementation differences of certain core features in many of the browsers. For example,
@LinkerOrder(Order.PRIMARY)
public class DedicatedWorkerLinker extends SelectionScriptLinker {

    @Override
    public String getDescription() {
        return "Dedicated Web Worker Linker";
    }

    @Override
    protected String getCompilationExtension(TreeLogger logger, LinkerContext context) throws UnableToCompleteException {
        return ".cache.js";
    }

    @Override
    protected String getModulePrefix(TreeLogger logger, LinkerContext context, String strongName) throws UnableToCompleteException {
        return "";
    }

    @Override
    protected String getModuleSuffix(TreeLogger logger, LinkerContext context) throws UnableToCompleteException {
        return "";
    }

    @Override
    protected String getSelectionScriptTemplate(TreeLogger logger, LinkerContext context) throws UnableToCompleteException {
        return "com/.../DedicatedWorkerTemplate.js";
    }
}

Figure 2.6: GWT linker
$wnd.XMLHttpRequest = XMLHttpRequest;
$wnd.setTimeout = function() {
    setTimeout.apply(self, arguments)
};
$wnd.clearTimeout = function() {
    clearTimeout.apply(self, arguments)
};
$wnd.clearInterval = function() {
    clearInterval.apply(self, arguments)
};

Figure 2.7: GWT boilerplate code

```javascript
var load = function() {
    gwtOnLoad(undefined, 'jssglrWorker', '', 0);
}
exports.load = load;
```

Figure 2.8: GWT initialization code

the XMLHttpRequest object is not available in Internet Explorer 6, and instead has to be loaded through a call to `new ActiveXObject("MSXML2.XMLHTTP.3.0")`; However, since JSSGLR does not need to support such old browsers, we were able to simplify this initialization.

Figure 2.7 shows the initialization required. The required variable, XMLHttpRequest, and all three required functions, setTimeout, clearTimeout, and clearInterval, are available in all targeted browsers, and can simply be assigned to the wnd variable. As JavaScript does not support function aliasing directly, an anonymous function is needed that simply dispatches to the function that should be aliased.

### 2.2.2 require.js

As mentioned in Section 2.1.2 JavaScript lacks a module system. The `require.js` library seeks to address this problem by defining a JavaScript module format.

### GWT initialization

Before JSSGLR can be the GWT initialization function must be called to allow the GWT setup code to run. Under normal circumstances the GWT compiler would add this call at compile time. However, since we have taken on the responsibility for all initialization the following snippet is required.

JSSGLR does not require any of the GWT strong naming functionality and as such does not need to pass any argument to gwtOnLoad other than the name of the module ('jssglrWorker') as can be seen in Figure 2.8. The assignment to `exports.load` is required to allow the code using the JSSGLR library to call the GWT initialization function.
2. Syntax

Fitting JSSGLR require.js into the require.js module format

In order for JSSGLR to be usable as a standalone library it has to use the require.js format. We can use the special format for legacy code, see Figure 2.3. No other modifications are necessary since the compiled code does not refer to the require, exports or module variables.

Discussion

We now have a fully functional library implementing the SGLR algorithm in JavaScript. This library can be used by any web application to parse a document for which an SDF language definition is available. In Chapter 5 we will address the performance problems caused by the size of the parse table.
Chapter 3

Editor services

Historically, programmers have used text-based editors to write code. These editors gradually adopted more and more features useful to the programmer, and some not (emacs famously has a chess feature). With the introduction of graphical user interfaces (GUI) came graphical editors, much like their non-graphical cousins, but fitted into a GUI. As the popularity of graphical interfaces grew, so did the popularity of graphical editors. Integrated Development Environment (IDE) became popular after that.

Now that interest is shifting to the browser, a similar history can be seen there. While early browser editors did not start out text-based, they operated in a graphical browser interface after all, they did start out very bland. The very first browser based editor were simple input fields in which text could be entered, after which changes were saved with an explicit action by clicking the submit button. Even read-only presentation of code would simply be styled in a monospaced font and perhaps line numbers to the side.

As the presentation of code on the web became more prevalent, programmers were quick to add basic functionality. At first simple regular expressions (regexp) were used to highlight the keywords of the language a document is written in. This same technique worked to some extent to highlight comments and strings, although nested comments and escaped strings complicated things. This regexp based approach was used mainly for the purpose of highlighting static content, there was still little editor support.

The Bespin project [4], announced in July 2009 [26], aimed to improve this situation. An ambitious goal was set for Bespin, it was to be an easy to use editor on the web, with all the features of a desktop code editor. The project used the canvas [28] API to create its user interface. Canvas is an HTML API that provides functionality to browser programmers similar to that of desktop GUIs. The advantage of the canvas API is that it gives developers very precise control of the look and behavior of the program. A significant downside though is that not all applications support canvas to the same degree, most notably Internet Explorer did not support canvas until version 9, which is not available on all Windows versions. The project was renamed to Skywriter [5] in September 2010 [27].

The Cloud9 project [20], see Figure 3.1 which is an independent effort by Ajax.org, was announced in September 2010 [19]. Reaching further than the Skywriter project, Cloud9 aims to be a full IDE running in the browser. The project was initially aimed mainly at JavaScript development, but started out with extensibility in mind. The main functionality
initially consisted of an editor, an outline view, and a plugin system to increase extensibility. The editor, named Ajax.org Cloud9 Editor (ACE), started out with features previously only found on the desktop such as key bindings and theming.

Shortly after, the Skywriter team announced that Skywriter had been merged into the Ace project. Unlike the Skywriter project, ACE does not use canvas, but uses regular HTML and CSS to render the editor. Not only does this make the editor usable in a wider range of browsers, it also improves accessibility. While ACE already had all the editor features Skywriter supported, it did not have the same level of extensibility. These features were ported from Skywriter to the ACE project, and the Skywriter project was abandoned.

Motivation

Currently all plugins developed for ACE that add language support are written in low level JavaScript. This creates a problem similar to that described in Chapter 2, there is too little abstraction. Rather than being able to focus on the language for which support is to be added, the programmer must deal with the details of implementing an ACE plugin. However, now that we have a generic library that can parse any language defined with SDF, we can create a plugin as well. By integrating the JSSGLR library as an ACE plugin we can create an editor for any language for which an SDF definition exists.
3.1 Cloud9

Cloud9 is a web IDE consisting mainly of an editor component, ACE, and several supporting plugins that offer various services. Cloud9 takes care of several things, such as providing a tabbed interface with ACE instances, a file browser, and even a command-line, the usual things one might expect to see in an IDE. All of this functionality is provided by plugins, which can be enabled and disabled at will, including the most basic things, such as the actual editor.

Included is a debugger that integrates both with node.js library and the Google Chrome inspector. Interesting is the command-line feature, which provides access to various tools such as file system access and version control, but is also integrated with the editor itself. The cloud9 codebase is split out into several (reusable) component libraries that can be cloned individually. However, some components depend on being checked out in the layout of the cloud9 repository, reducing their reusability.

3.1.1 Edit modes

ACE, Ajax.org Cloud9 Editor, is a library on its own, and can be used outside the Cloud9 container. It has its own architecture, namely that of modes. In ACE a mode controls the editors behavior, it can be seen as the controller. The presentation layer is handled independently of the current mode, the mode only handles what data should be shown. This is achieved by decorating the document text with tokens that describe how the text should be rendered (e.g., as a comment, keyword). There are also annotations that are displayed in the gutter of the document (left hand margin).

3.1.2 Event-driven

The ACE architecture is event-driven, most features are achieved by registering as a listener to the right service and handling the resulting event. The mode has full access to the current document, and can modify its contents when handling an event. The presentation consists of two distinct components, the gutter and the main body. Both of which contain one or more layers. Layer are diverse, there is a layer just to render the print margin, and there is a layer that is responsible for displaying all of the text with its correct markup.

3.1.3 Theming

Theming is done with simple CSS, rather than manually formatting each text element, the text layer declares the text class (keyword, identifier, etc.), after which the themes CSS takes care of styling it in the right way. This makes theming very straightforward, define what each element should look like in css.

3.1.4 Missing features

Noticeably absent are a way for the mode to create error annotations on tokens directly. Also lacking is infrastructure for the mode to support autocompletion. Neither is there
3. Editor services

The ACE editor makes several assumptions about the architecture of its editor plugins.

- Parsing time is near-instant
- Tokens are always available
- Tokens change only when the document changes
- Each line can be split into a pre-defined set of tokens

3.2 The ACE Plugin Architecture

ACE makes use of web workers, described in Section 2.1.3. The primary use web workers in ACE are to remove the semantic analysis from the main thread. Communication with web worker is achieved by sending messages over a socket. The JavaScript edit mode runs Narcissus, JsHint, and JSLint in a web worker to provide syntactic and semantic warnings and errors for JavaScript code.

When the mode is first loaded it starts by creating the worker, specifying a javascript file to be used for initialization. The worker starts by executing the initialization script, which creates a small object that intercepts calls to what would normally be the require.js library.
(a shim). The require.js shim uses the importScripts functionality available to web workers instead of the technique described in Subsection 2.1.1. Once the shim is set up, the worker creates a listener that waits for the main thread to send it an initialization message.

The main thread sends an initialization message containing the name of the main module and class. When the listener receives the message it creates a Sender class that abstracts from the web worker interface. The Sender will be used by the worker to send and receive messages from the main thread using the techniques, such as EventEmitter, used throughout the ACE codebase. With the Sender set up the require.js shim is used to load the worker class. The worker class is passed the Sender and is now expected to handle further messages received from the main thread.

The worker starts off by loading the parse table it will be using for the current document. While the parsing of the parse table into a require.js module is done by require.js, loading the parse table is an explicit function call. When the parse table is loaded, the resulting AST is used to initialize the JSSGLR parser. After creating the JSSGLR parser the worker sends the loaded signal to the main thread.

**Parsing**

When the main thread receives the loaded signal it retrieves the current contents of the document and sends the worker an update message. The worker receives the update message and checks if the parser has been initialized yet. If the parser has not finished its initialization process, the update is ignored. Within a few seconds after starting the parser finishes initialization. All changes received after initialization finishes result in the worker calling the JSSGLR parser. The worker waits synchronously for the results of the parser and ties each token to the line and code fragment it was created from. A jssglr message is sent to the main thread containing tokens and any errors encountered by the JSSGLR parser.

In the worker thread ACE uses a token-driver to make sure the document is up to date when a user makes any changes. The token-driver requests new tokens from the tokenizer for each changed line.

### 3.2.1 Receiving tokens

ACE is designed around regex-based syntax highlighting engines which complicates the integration of JSSGLR as token provider. The result of this design are several assumptions, see Subsection 3.1.5. JSSGLR runs in a separate worker thread, as such we cannot rely on highlighting tokens being available at all times, which breaks the listed assumptions.

We have worked around this problem by creating our own tokenizer and slightly modifying the way ACE treats tokenizers. The ACE tokenizer framework assumes that a tokenizer has a certain state that must be kept track of. Our custom tokenizer uses this state to instead record which line-number a code fragment belongs to. This is a somewhat fragile construction, since ACE does not guarantee that lines will be parsed sequentially. However, since this is the only way to reliably highlight grammars that require access to the full document to do highlighting we deemed it an acceptable requirement. We have verified that the current implementation does indeed behave in the desired way.
3. Editor services

In addition to having to keeping track of the current line number our tokenizer is different from other tokenizers in that it does not always have a result available. Not only that, but when results are passed to it from the worker thread, ACE is not aware that new tokens have arrived. To solve this problem we do two things, firstly we check that the tokens we have for a certain line belong to the code ACE wants to highlight. If the requested code does not match the tokens we have, either due to the user editing the code, or due to not having any results for that fragment yet, we return the fragment without any highlighting.

When the editor first loads, the file is initially not highlighted, as the JSSGLR parser is still being initialized. We have modified ACE to register as a listener to the tokenizer for the update event. When the worker sends new tokens, these are stored and a update event is fired for all rows of the document. When ACE receives an update event it will ask the token-driver to retokenize all affected lines. The driver in turn will ask our tokenizer for tokens for every row in the document. Our tokenizer, having just received a new batch of tokens, will have tokens for the entire document. When the token-driver has received the last tokens it will notify ACE that the document needs to be redrawn.

Shifting tokens

A downside of this approach is that if a user inserts a newline, i.e., hits enter, all tokens we had for everything below that line is invalidated. The process of retokenizing the entire file and receiving new tokens takes a while, during which the user has no highlighting for the entire document. Adding newlines is a fairly common operation, making it an interesting optimization target. The ACE editor fires specific event when a change is made to the document. By registering to these events and recognizing which signify a newline insertion we can shift the tokens we have down. Similarly, if the user removes a line of text, we can throw away the tokens we have for that line and shift all other tokens up.

The events we need to listen for to recognize an insertion are insertText, and insertLines. The former is fired if only one newline is inserted, the latter when multiple lines are inserted. The situation is similar when lines are removed, we need to listen to the removeText, and removeLines events. By adding this simple improvement the editing experience is made a lot smoother.

Removing listeners

We discovered that after switching modes from and to our plugin once certain plugin code would be executed twice. In particular, any listener would be run twice for each event, consequently deleting too many lines and over-correcting. When switching modes the ACE editor sends each plugin a terminate signal. At first, our plugin merely cleared all document annotations, as is expected when a plugin adds annotations to the document. The problem turned out to be that the document does not automatically clear its listeners when the mode changes. By explicitly de-registering our listener when receiving the terminate signal the problem was addressed.
Discussion

We have created an architecture that facilitates the creation of parser based editors on the web by means of an editor specific plugin and a generic parser. The ACE plugin we have developed is capable of parsing any language defined in SDF and is an example of this architecture. Additionally, we have designed the plugin such that the CPU-intensive components run in a separate worker thread. In Chapter 5, we will address the performance problems caused by the size of the parse table used by the plugin.
Chapter 4

Semantics

In addition to syntactic errors many modern IDEs provide the user with semantic errors. A program containing semantic errors might be syntactically valid, as such a simple parser would not find anything wrong with the program. Semantic errors range from missing variable declarations, as can be seen Figure 4.1, to subtle issues involving multiple inheritance and overridden functions.

In order to provide the user with these errors a semantic analysis must be performed. Semantic analysis is commonly performed by analyzing the output produced by a parser. This process is usually done by the compiler of the language used. For this reason there is generally no distinction between syntactic and semantic errors.

The Syntax Definition Formalism (SDF) used by the Scannerless Generalized Left-to-right (SGLR) parser does not have any facilities to define semantic constraints. As such,
while the JSSGLR library can provide syntactic warnings and errors for any language de-
defined in SDF, but it cannot perform a semantic analysis. A separate formalism is required to
define semantic constraints and detect when they are violated.

**Motivation**

The Stratego [8] language is designed to facilitate transforming abstract syntax trees (AST).
While performing these transformations, Stratego can collect information about the AST.
Due to this analysis Stratego is a particular good fit for performing semantic analysis. How-
ever, there is currently no JavaScript backend that would allow us to use Stratego in the
browser. By extending the Stratego compiler such that it will target JavaScript we can use
Stratego to write the semantic constraints to be used in conjunction with the JSSGLR li-
brane.

4.1 Stratego

These transformations are defined by rules that are applied to all terms in an order defined
by a traversal strategy. Transformation definitions can be written using concrete syntax,
eliminating the need to match AST nodes directly. Stratego is most often used to write
compilers and the compiler tool-chain.

4.1.1 Terms

A core feature of the Stratego language are Terms, which can be seen as trees. A common
usage of Stratego is the transformation of abstract syntax trees which can be efficiently
represented using terms. The *de facto* Stratego term library is the Annotated Term Format
(ATerm).
Annotated Term Format

The ATerm format is a language aimed at the definition of trees. The format consists of the following basic elements:

Integer: An integer number. For example, 31337 is an integer term.

String: A string constant, optionally containing backslash escaped characters. For example "Hello World" is a string term.

Constructor: A constructor (application) consists of an identifier and a number of terms. For example, Plus(Int("4"), Var("x")) consists of the Plus constructor being applied with an Int and Var as argument. Both the Int and the Var argument are itself a constructor application with a string as argument.

List: A list consists of zero or more terms, these are usually of the same type. For example, [Var("x"), Var("y"), Var("z")]} is a list with three Var terms.

Tuple: A tuple is simply a constructor application that has no identifier component. For example, (Var("x"), Var("y")) is a tuple with two Var terms.

Annotation: An annotation is a list of terms that are attached to another term. For example, Var("x"){Type("Int")} is a Var annotated with a Type term.

4.1.2 Strategies and Rules

Most Stratego programs work by defining strategies that traverse a forest of terms. A strategy receives as argument a number of other strategies, and a number of terms as argument. Each strategy operates on the current term, which it can try to match against, or replace with a built term. Strategies can either succeed, or fail, which they do with the fail keyword. The first strategy strategy with the given name that matches is executed, which then operates on the matched term, usually by invoking either another strategy or itself on a subterm of the current term.

4.1.3 Stratego compilers

Generally, a compiler consists of a front end and a back end, see Figure 4.3. The front end takes the user input and transforms it into a format the back end understand, the back end then transforms it into the desired output format. This separation between front end and back end allows these components to be reused. Several front end can be used when a compiler can parse several source languages. Likewise, different back end can be used when a compiler can target multiple output languages.

The two maintained Stratego compilers both target a general-purpose programming language (GPL), specifically, C and Java. Both compilers share a common front end which parses the Stratego input and generates an AST. The two back ends then take this AST and generate output in their target language. In order use Stratego to perform semantic analysis in the browser we must write a JavaScript back end for the Stratego compiler.
4. Semantics

4.2 Design

The design of the strjs compiler is based on the Stratego-to-Java (strj) compiler. It reuses the same front end, which generates and analyses an AST, and implements a custom back end, which converts the AST to JavaScript.

4.2.1 Front end

The first compile stage, the front end, consists of reading the main input file and parsing it to an AST. Parsing of the Stratego input is handled by the pack-stratego-trm strategy, which is part of the strc library. After parsing the options it received, it dispatches to pack-stratego. In pack-stratego all imports are extracted from the input AST. Starting with the input AST, a dependency graph is constructed from the imports of all (imported) files. As part of resolving each import, the pack-stratego-parse-stratego strategy is called. The pack-stratego-parse-stratego strategy resolves an path and a set of search locations to an AST. By default, it will try to read the imported file from one of five ways:

1. An absolute file path (only if the import is specified as absolute path)
2. From a source file in one of the directories in the include path
3. A parsed file (in .rtree format) from a directory in the include path
4. Using the transformation tool composition (XTC) system (only for the C compiler)
5. A cached file found using the XTC system (in .rtree format, only for the C compiler)

The .rtree format used for cached files is a stripped, serialized form of the AST generated by parsing a Stratego source file. Libraries commonly use the .rtree format as a way to define which symbols they include, since only the declaration of a symbol is included in an .rtree file, not the definition. As we are using the Java compiler, option 4 and option 5 are not relevant. When all imports have been resolved in this way, a complete AST, with all imports resolved, is the result. Using the tfcl(|"Stratego-Sugar-Cong", "pack") construct, the correctness of the AST is verified.

4.2.2 Front end (analysis)

The next stage in the compiler is the analysis, which takes an AST and analyses it. The frontend strategy handles the analysis aspect, recording which strategies, constructors are used. Using this knowledge, the compiler can generate code only for those parts of the input that are used. During this part of the front end stage the frontend strategy also performs desugaring on the input AST. A distinction is made between strategies that are defined locally, as part of the input file, and strategies that are defined in one of the imported libraries. If a strategy is defined locally, the joindefs strategy is used
to: ‘Join multiple definitions for the same strategy operator into one definition by unifying the list of formal strategy parameters’ ¹. This is important because, as described in Subsection 4.1.2 all strategies with the same name are evaluated in turn. Again, the \texttt{tfc1("Stratego-Core", "fe-cong")} construct is used to verify that the AST has been completely desugared to the Stratego Core language.

The compiler can run in two different modes depending on whether the output will be a library or not. In library mode, all local symbols are exported, even if they are not used in the library itself. This allows a library to define symbols intended solely for the consumer of the library. When not in library mode, unused definitions are removed to reduce the size of the generated output. In either mode, unused external definitions are removed, as they are not useful even to the consumer of the library.

After desugaring, the \texttt{optimizer} strategy is called, which applies several optimizations depending on the optimization level, such as inlining and dead code removal. Optionally, the compiler will cache the current AST in an .rtree file to speed up future compilation using the \texttt{strc-export-external-defs} strategy. Instead of continuing, the compiler can be instructed to at this point emit the AST and exit. At this point the front end processing is done and the AST is ready to be passed on to the back end.

### 4.2.3 Back end

During normal execution, the AST is passed through the \texttt{backend-simplify} strategy to the \texttt{js-compile-spec} strategy, which is responsible for converting a fully analysed AST into JavaScript. The \texttt{backend-simplify} strategy does several things to prepare the final AST for output. For example, since Stratego is case sensitive, and some output targets use case-insensitive filenames to identify symbols, a dollar sign is inserted before capital letters: \texttt{Desugar} becomes \texttt{$Desugar$}. Finally, the complete AST is passed to the \texttt{js-compile-spec} strategy, which generates the actual output.

### 4.3 Static linking

Section 2.1.1 describes the lack of modules in JavaScript. Stratego programs generally make use of several libraries, so a solution to the lack of modules is needed. Since the library code is required for the program to run, it must in one way or another be included in the JavaScript program. The simplest solution then is to compile an entire program from Stratego to JavaScript and use \texttt{require.js} to load the libraries a program needs. This approach is similar to that of Java \texttt{jar} files.

To examine the feasibility of this approach we examined which libraries a basic program would need at the bare minimum. We concluded that at the \texttt{libstratego-lib} library would be the smallest requirement, as it contains much functionality that is core to any Stratego program. When we compile the entire \texttt{libstratego-lib} library to JavaScript, the resulting file is 12MB JavaScript file. Considering that each program would require this

---

¹\texttt{strc} library, \texttt{strc/front/needed-defs.str
library to be available, and the reluctance of users to use web based programs that take long to load, we deemed this approach to be infeasible.

Upon examining the libstratego-lib we encountered many symbols that are unused by most simple programs. As discussed in Subsection 4.2.2 when compiling libraries the front end also includes all unused symbols to make them available to consumers of the library, this is also known as dynamic linking. The opposite of dynamic linking is static linking, in which the compiler includes all the required code in the final executable, including all libraries. The downside of this approach is that compilation takes longer, since the library code must be evaluated as well. However, when compiling statically, the compiler can remove all unused symbols from the imported libraries as well as from the main program. By including only the relevant library code a small program like ![1,2]; map(inc) will include only the code for the map and inc strategies.

Due to this added advantage, we decided to use static linking. While the compilation time is increased, it is a one-time cost that the developer must pay that results in a many-time saving for each user of the application. During development, static compilation can be turned off to avoid the longer compile time.

4.4 s2js

The final step in compiling Stratego to JavaScript is to transform the Stratego AST to JavaScript. Commonly this step is done by a model-to-model transformation, transforming the Stratego AST to a JavaScript AST. However, doing so requires a more or less complete grammar of the target language, in this case JavaScript, from which a language model can be derived. Due to JavaScripts quirks it is difficult to write an SDF grammar for it, no complete SDF definition currently exists. The Stratego AST has been thoroughly desugared and simplified, resulting in a relatively small core language. Using string interpolation to generate code for such a small core language is very feasible. Additionally, creating an SDF definition for JavaScript would be a significant amount of work. Given these constraints we decided to use string interpolation to create JavaScript code directly from the Stratego AST.

String interpolation is a technique in which placeholders contained in a string, a template, are replaced with values from the environment in which the string is defined. In Stratego string interpolation can embed any Stratego expression and has been enhanced with an indentation preserving algorithm. Strings created in this way are readable, although not as good as with model-to-model transformation. A downside of using string interpolation is that no error checking on the generated output is performed. If output is generated by means of a model-to-model transformation both syntactic and semantic errors in the generated output can be detected. When using string interpolation such errors are only found when the generated output is used.

In s2js, the interpolation process consists of two phases, first all declarations are collected, second all output is generated. In the collection phase the AST is traversed and a dynamic rule [7] is created to store all constructors. At the start of the output generation phase the constructors that were collected are declared and initialized. These constructors must be collected, declared and initialized before the strategies, as the strategies use these
function(ctx, term) {
  Fail_0: {
    term = x_0_0(ctx, term);
    if(term === null) {
      break Fail_0;
    }
    return term;
  }
  return null;
}

Figure 4.4: Labeled break statement

function(ctx, term) {
  Fail_0: {
    term = x_0_0(ctx, term);
    if(term === null) {
      break Fail_0;
    }
    term = y_0_0(ctx, term);
    if(term === null) {
      break Fail_0;
    }
    return term;
  }
  return null;
}

Figure 4.5: Sequence operator

constructors. Declaring and initializing a constructor consists of creating a new variable with the appropriate name, and calling the `factory.makeCtor` method. The factory constructs a new Stratego-style constructor from the specified parameters.

The defined strategies are already available, as they are an explicit term in the AST. The next step does the heavy lifting, translating all strategies to JavaScript. All strategy names are first mangled; their arity is appended for both term and strategy arguments. Thus, a strategy `foo` with no term or strategy argument is mangled to `foo_0_0`. For each strategy a variable is created in which the strategy is stored as a function object. Stratego strategies can fail at any point during execution, which means that any cleanup code required would potentially be duplicated a lot. A simple solution to this problem is to use a labeled break statement, as in Figure 4.4. If an error is encountered, the `break Fail_0` statement will break out of the block labeled `Fail_0` and proceed at its end. At that point the function will perform any cleanup required, and finally return. A strategy that fails is implemented as a function that returns `null`, otherwise the (possibly modified) current term is returned.
4. SEMANTICS

```javascript
function(ctx, term) {
  Fail_2: {
    Success_0: {
      Fail_3: {
        term = x_0_0(ctx, term);
        if(term === null) {
          break Fail_3;
        }
        break Success_0;
      }
      term = y_0_0(ctx, term);
      if(term === null) {
        break Fail_2;
      }
    }
    return term;
  }
  return null;
}
```

Figure 4.6: Guarded left choice operator

Strategies can be combined into a sequence with the sequence operator: ;. When two strategies are combined in this way, such as in x; y, the result of applying strategy x is fed to y. See Figure 4.5.

In Stratego there are several constructs that deal specifically with strategies that fail. The <+ operator, also known as guarded left choice is one of these. For example, in a <+ b, the a strategy is executed, and only if it fails, the b strategy is tried instead. To implement the guarded left choice operator, we must deal with a failure in strategy a in a different way, namely to instead execute strategy b. Not only that, but we must also deal with failures in any of the strategies that are part of a when a is a composite strategy, such as in the case of x; y <+ z. In order to solve this problem the compiler remembers which label it should break to when a fail is encountered. In the case of x <+ y, the generation of x and y results in multiple fail labels. As can be seen Figure 4.6 the first fail block encompasses the entire expression as before. If the strategy before the <+ succeeds the second strategy must not be executed, a success label is needed. Failures in the b strategy, such as in x <+ y; z, are handled with another fail label.

Discussion

We have ported Stratego, a language specialized in semantic analysis, to JavaScript so that it can be used in the browser. Using this language, developers will be able to extend the available editor services with semantic analysis. The code generated by the Stratego-to-JavaScript compiler can be executed by any JavaScript interpreter.
Chapter 5

Performance

The performance of the Java Scannerless Generalized Left-to-right (JSGLR) parser has been fine-tuned during its development, resulting in an extremely fast and efficient parser. Even so, the performance of JSGLR corresponds with the size of the language as well as the size of the file being parsed. A JSGLR parser can take several seconds just reading in the parse table during start up. Responsiveness is vital for the usefulness of an IDE, users might be willing to wait a certain time for the IDE to get started, but once it is running it must provide near-instant feedback to their input. Very large languages will result in a parser that takes several seconds to analyze a file that is a mere 500 lines of code in size.

Motivation

Because the JavaScript SGLR (JSSGLR) parser is compiled with GWT from JSGLR, it inherits all these problems from JSGLR and suffers an additional performance penalty due to its emulation of Java techniques in JavaScript. While some of the performance improvements made to JSGLR can be directly applied to JSSGLR, many of them are specific to Java, or are simply impossible to implement in JavaScript. Initially, the JSSGLR parser would take at least 30 seconds just to start up, and would sometimes fall into an infinite loop, consuming CPU without accomplishing anything.

To make the JSSGLR generally useful its performance must be improved significantly. In order to measure the effectiveness of our improvements we will benchmark the performance of the JSSGLR parser. Looking forward, we also benchmark the Stratego-to-JavaScript (strjs) compiler to estimate how integrating semantic analysis will affect performance.

```
parse-table(6, 0, [  
    label(prod(cf(layout())), attrs([assoc(left())])), 260
], label(prod(attrs([term(cons("Start"))])), 258)
, label(prod([char-class([97])], lit("a"), no-attrs()), 257)
])
```

Figure 5.1: Stripped down parse-table
5. PERFORMANCE

5.1 Parsing parse tables

The JSGLR parser uses parse tables generated from an SDF definition, see Chapter 2. This parse table, and ATerms in general, come in several formats [6]:

**Textual:** The plain text format is intended primarily for human consumption, it is very inefficient.

**Binary:** The binary format (BAF) is very efficient, it makes use of term sharing where multiple references to the same Term take up a constant amount of space. BAF uses many optimizations to reduce the size of the stored term.

**Textual shared:** The plain text format adds simple sharing to the textual format, reducing the size significantly, but remaining textual.

The size difference between the textual and the binary format is significant, typically a 5x increase. Most of this is caused by the lack of term sharing, which can be seen from the difference between textual shared and binary, which is typically only a 1.6x increase.

While sub-optimal, the JSSGLR parser initially used textual formatted parse tables, due to the inability of the derived parser to correctly read in binary parse tables. Not only are textual parse tables larger by a factor of 5, they are a lot more time-consuming to read in, as the parser table parser has to parse 5 times more text. Porting the textual shared format to the JavaScript parse table environment would improve performance.

5.1.1 Efficiently parsing parse tables

The first step to improving this situation is to stop using the naive parser used by JSGLR. This parser is translated by GWT from Java to very inefficient JavaScript. While the textual parse table format is not very complex, it is not trivial to parse correctly either.

Instead of writing a parser manually, we could rely on the browser to do the parsing for us. Several options are available, including JSON and XML, but one stands out in particular. All modern browsers have invested a lot of time in optimizing the code that parses JavaScript; what if we could trick the browser into parsing the parse table definition as if it were JavaScript?

Indeed we can. As can be seen in Figure 5.1, the parse table format is very similar to JavaScript. The parse table format is a valid JavaScript program from a syntactic perspective if we must rename all keywords. For example, we must replace `class` with `_class` in all constructor names. If we sanitize the constructor names further, by replacing `’-’` with `’\’`, the parse table is also valid semantically, assuming we first define a function for each of the used constructors. Defining a function for each constructor is easy, but what should this function do, and how should it treat its arguments?

Let us first analyse what a JavaScript parser will do when parsing the parse-table-massaged-into-javascript shown in Figure 5.1. The parse table is effectively a single function call, albeit with arguments that must first be evaluated, to the `parse_table` function. The first two arguments are simply numbers, and will be passed to `parse_table` as such.
Parsing parse tables

```
var parseArgs = function(args) {
    var length = args.length;
    var result = [];
    for(var i = 0; i < length; i++) {
        var arg = args[i];
        var parsed = parseArg(arg);
        result.push(parsed);
    }
    return result;
}
```

Figure 5.2: JSSGLR parseArgs function

```
var parseArg = function(arg) {
    var type = Object.prototype.toString.call(arg);
    if(type === '[object Array]') {
        return _f.makeList(parseArgs(arg), null);
    }
    if(type === '[object Number]') {
        return _f.makeInt(arg);
    }
    if(type === '[object String]') {
        return _f.makeString(arg);
    }
    return arg;
}
```

Figure 5.3: JSSGLR parseArg function

Conveniently, the syntax in JavaScript and Stratego for lists is identical, so the final argument, a list, is parsed correctly. The contents of the list are a little more complicated, as they are themselves function calls. In order to evaluate the parse_table function call, the JavaScript parser will first have to evaluate the calls to label. When all arguments of a function call are basic types, such as with lit("a"), evaluation is simple.

Implementing the constructors

The implementation of the constructors uses two functions, parseArgs and parseArg. When the lit constructor is called like lit("a"), the lit function passes its argument through the parseArgs function. The parseArgs function, see Figure 5.2, simply takes each argument, and hands it to parseArg. The _f object referenced in the parseArg function, shown in Figure 5.3, is a factory that constructs Stratego objects. When parseArg encounters a list, it dispatches back to parseArgs to convert its arguments to Stratego objects. Using the parseArg and parseArgs functions and the _f factory we can create the constructors used in a parse table.

There are two cases we need to take into account, that of a constructor with, or without arguments. In Figure 5.4, the construction of a no-argument constructor as a JavaScript object is shown. The makeConstructor call creates a new constructor, with no arguments,
5. Performance

```javascript
var _class = _f.makeAppl(
    f.makeConstructor('class', 0),
    parseArgs([]),
    null
);
```

Figure 5.4: Constructing a no-argument 'class' constructor

```javascript
var _goto = function() {
    return _f.makeAppl(
        f.makeConstructor('goto', arguments.length),
        parseArgs(arguments),
        null
    );
};
```

Figure 5.5: Constructing a n-arg 'goto' constructor

```javascript
var lit_a = lit("a");
var list_97 = [97];
var char_class_97 = char_class(list_97);
return prod([char_class_97], lit_a, no_attrs);
```

Figure 5.6: Storing parse-tree terms in variables

named class. Parse tables never refer directly to constructors, they only ever create constructor applications, as such we pass the newly created constructor to makeAppl. The makeAppl call receives the newly created class constructor, an empty list (as it has no arguments), and no annotations. Resulting is a constructor application of a no-arg constructor named class.

Constructors that do receive arguments require a different approach, since they must be evaluated as function calls and pass their arguments to the makeConstructor call, as shown in Figure 5.5 Again, the makeConstructor call creates a constructor, named goto. This goto constructor takes as many arguments as the _goto call received. Resulting is a constructor application of an n-arg constructor named goto.

5.2 Reintroducing term sharing to parse tables

Since the textual shared format is so much more efficient than the regular textual format, it makes sense to attempt to replicate its success. In order to do so, we must reuse terms in the same way the shared format does. Doing so involves storing a reference to a terms somehow, and then using that reference later on instead of redefining the term. In JavaScript this can be achieved by storing the term as a variable, similarly to how we stored to class constructor application.
5.3 Optimizing variable naming

In addition to the slowdown caused by the larger file size, the download size in itself is problematic. If a large parse table is downloaded in textual format, a user on a slow connection would have to waste a lot of time waiting. Now that we have eliminated the spurious memory usage of large tables, it makes sense to attempt to reduce the size. Instead of giving each term a recognisable name, we can reduce space requirements by using short names instead.

Notice that in Figure 5.7, the now pre-defined constructors take up most of the space in the parse table. Since these pre-defined constructors are called most often, we can create shorter aliases for them and use those instead.

While Figure 5.8, in which we have implemented this, might seem longer, it actually saves a lot of space due to how often the pre-defined constructors are used and the fact that the alias is only defined once. There is only a limited number of 1-character variables that are valid in JavaScript: both lower case and upper case letters of the alphabet, the underscore, and the dollar sign. After we exhaust these 54 possible combinations, we can simply start with 2-character variable names. In JavaScript a variable may contain a numerical digit, as long as it is not the first character, thus increasing the total number of possible characters to 64. This results in variable names going from _ to aa, etc.
Currently each alias takes up 8 characters in addition to the length of the variable name and the actual term being stored. In JavaScript we can define multiple variables in one var declaration by separating them with commas. If we leave out the spaces around the equals sign, as in Figure 5.9 the overhead can be reduced to two characters.

This approach works well, but there’s one last optimization we can do to reduce the size of the parse table. We currently assign term names based on the order in which they occur in the parse table. For large parse tables, this results in the terms that occur early to have short names, and the terms that occur later to have increasingly long names. When observing a parse table, we notice that some of the most repeated terms do not appear until halfway through the table! The earlier terms that are assigned the shortest names are often used exactly once, thus taking up valuable space.

To verify this observation we instrumented the compiler to record how often a term is used, and add that count as a comment behind the alias declaration. Doing this confirmed two things, firstly, most terms occur exactly once, creating an alias for actually takes up additional space, rather than saving any. Secondly, the location of the first occurrence of a term in the parse table is not a very good indicator of how often a term occurs in total. The solution is obvious given the above two observations, we should assign variable names based on frequency rather than location and exclude terms that are used only once.

Following the line of reasoning that led us to exclude terms that are used only once, we investigate if excluding terms that are used less than \( n \) times results in a size reduction. We expect that if this is the case, it will only hold for low \( n \), since the overhead for creating a new alias is fairly low, 2 characters plus the length of the variable name.

Figure 5.10 shows the relation between the size of the parse table and the cut off value \( n \). Surprisingly, there is a slight dip at \( n = 3 \), which means that if we alias only those terms that are used at least three times the size is actually smaller than if we alias the terms that are used twice as well. This is probably a characteristic to the parse table used to investigate this optimization. A JavaScript parse table with \( n = 1 \) has to sort many more terms by their
usage count than a table with $n = 3$ due to distribution of term frequency. The time it takes to create a JavaScript version of a parse table for $n = 1$ is three times as long than with $n = 2$ or $n = 3$, for only minor size gains. Since compilation time decreases only very little after $n = 3$, we decided to use $n = 3$.

A before and after comparison is shown in Figure 5.11 (with additional newlines added to 5.11(b) for legibility). The difference for the full parse table is about half the size of the original. For larger parse tables the result is even greater, reducing the size to almost a third of the original size. Notice that as we chose $n = 3$, the 'reduce' constructor is not aliased.

When we use the variable naming described above for large parse tables, we quickly run into problems. For example, at slightly over 2000 terms, the variable name `do` is generated, which is a keyword in JavaScript. The same thing happens again for the variable name `if`, and `in`. We decided to go with the most simple solution possible, introduce a blacklist against which all variable names are checked. When a variables is generated that is on the blacklist, we instead skip it and generate another one. We can use the same mechanism to prevent variable names from starting with a digit. Since the blacklist is limited in size, this process of retrying until a non-blacklisted variable is found works well.

5.4 Benchmarks

We have performed two sets of benchmarks to measure the performance of our solution. All benchmarks were performed on a dual-core 2.8Ghz Lenovo W500 laptop with 4GB RAM running the Debian operating system. At the time of the benchmark no other programs were running on the laptop.

5.4.1 Plugin performance

To compare the effectiveness of the solutions proposed in this chapter we instrumented the plugin code such that it records a timestamp at the start of the loading process. At certain specific points we recorded the current timestamp, we then logged the difference in milliseconds between both points. We have named these points so that we can refer to them later. In the ‘before’ situation, measurements were performed at the following points:

**Worker:** When control passes to the worker initialization code

**Hold time:** In worker main loop, called at the end of the hold time

**Parser:** When the loaded event fires

**Result:** When the first result arrives from the parser
parse-table(6, 0,
  [ label(  
    prod(
      [cf(layout()), cf(layout())],  
      cf(layout()),  
      attrs([assoc(left())]),  
      260)
  , label(  
    prod(
      [lit("a")],  
      sort("Start"),  
      attrs([term(cons("Start"))]),  
      258)
  , label(  
    [char-class([97])],  
    lit("a"),  
    no-attrs(),  
    257)
  ],
  priorities([arg-gtr-prio(260, 1, 260)]))
)

(a) Parse table in textual format

```javascript
var E=sort,F=prod,G=label,C=_goto,H=action,A=state_rec;
var D=[256],B=[];

return parse_table(
  6
  , 0
  , [ G(F([E("<START>"), char_class(D)]), E("<Start>"), no_attrs), 261)
   , G(F([cf(layout), cf(layout)]), cf(layout), attrs([assoc(left())])), 260)
   , G(F([E("Start")], E("<START>"), no_attrs), 259)
   , G(F([lit("a")], E("Start"),  
     attrs([term(cons("Start"))]),  
     lit("a"), no_attrs), 257)
  ],
  priorities([arg_gtr_prio(260, 1, 260)])
);
```

(b) Parse table in new JavaScript format

Figure 5.11: Before and after of a minimal parse tables, minus parse states
### Benchmarks

<table>
<thead>
<tr>
<th>Table</th>
<th>size (kb)</th>
<th>worker (ms)</th>
<th>hold time (ms)</th>
<th>parser (ms)</th>
<th>result (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini</td>
<td>&lt; 4</td>
<td>127</td>
<td>501</td>
<td>137</td>
<td>1014</td>
</tr>
<tr>
<td>Go</td>
<td>748</td>
<td>187</td>
<td>501</td>
<td>1976</td>
<td>2791</td>
</tr>
<tr>
<td>Stratego</td>
<td>1188</td>
<td>116</td>
<td>502</td>
<td>2821</td>
<td>3707</td>
</tr>
<tr>
<td>WebDSL</td>
<td>6556</td>
<td>122</td>
<td>502</td>
<td>14656</td>
<td>15537</td>
</tr>
</tbody>
</table>

(a) Loading table files directly

<table>
<thead>
<tr>
<th>Table</th>
<th>size (kb)</th>
<th>worker (ms)</th>
<th>init table (ms)</th>
<th>parser (ms)</th>
<th>result (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini</td>
<td>5</td>
<td>157</td>
<td>2</td>
<td>0</td>
<td>162</td>
</tr>
<tr>
<td>Go</td>
<td>340</td>
<td>251</td>
<td>227</td>
<td>45</td>
<td>752</td>
</tr>
<tr>
<td>Stratego</td>
<td>512</td>
<td>314</td>
<td>353</td>
<td>104</td>
<td>1124</td>
</tr>
<tr>
<td>WebDSL</td>
<td>2244</td>
<td>840</td>
<td>1418</td>
<td>1351</td>
<td>5028</td>
</tr>
</tbody>
</table>

(b) Loading JavaScript table format

Figure 5.12: Performance metrics of ACE plugin as measured by time taken to reach various points in the execution

<table>
<thead>
<tr>
<th>Table</th>
<th>table gzipped</th>
<th>JavaScript gzipped</th>
<th>productions</th>
<th>gotos</th>
<th>states</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Go</td>
<td>52</td>
<td>48</td>
<td>676</td>
<td>21556</td>
<td>1094</td>
</tr>
<tr>
<td>Stratego</td>
<td>224</td>
<td>76</td>
<td>659</td>
<td>38056</td>
<td>1977</td>
</tr>
<tr>
<td>WebDSL</td>
<td>324</td>
<td>292</td>
<td>2238</td>
<td>141867</td>
<td>8764</td>
</tr>
</tbody>
</table>

Figure 5.13: Metrics of tested languages, showing the parse table size after gzip compression, the same for the JavaScript format and several language properties indicative of language complexity

In the 'after' situation, measurements were performed at the following points:

**Worker:** When control passes to the worker initialization code

**Init table:** When the parse table init function has been called

**Parser:** When the loaded event fires

**Result:** When the first result arrives from the parser

We recorded these events for each of four languages. Firstly, Mini, the smallest possible language, recognising only a single ‘a’ character. Secondly, Go, a small C-like language developed by Google. Thirdly, the Stratego language, and finally the WebDSL language.

Figure 5.12 shows a comparison of the plugin performance before and after the optimizations presented in this chapter. As can be seen in Figure 5.12(a) in the old situation the worker is loaded in more or less constant time. The 'hold time' is to give the JSSGLR library time to load. Most time is spent loading the parser, since first the parse table must be downloaded, and then parsed.
5. Performance

```javascript
application helloworld

imports hello

init {
    var hello := Hello {
        name := "Hello World"
    };
}
```

Figure 5.14: WebDSL snippet used to benchmark the plugin

<table>
<thead>
<tr>
<th>Document size (lines)</th>
<th>fastest</th>
<th>slowest</th>
<th>fastest with errors</th>
<th>slowest with errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>19</td>
<td>32</td>
<td>23</td>
<td>35</td>
</tr>
<tr>
<td>20</td>
<td>35</td>
<td>46</td>
<td>94</td>
<td>112</td>
</tr>
<tr>
<td>30</td>
<td>51</td>
<td>68</td>
<td>143</td>
<td>161</td>
</tr>
<tr>
<td>40</td>
<td>72</td>
<td>88</td>
<td>140</td>
<td>201</td>
</tr>
<tr>
<td>50</td>
<td>93</td>
<td>106</td>
<td>153</td>
<td>201</td>
</tr>
<tr>
<td>100</td>
<td>194</td>
<td>264</td>
<td>262</td>
<td>349</td>
</tr>
<tr>
<td>150</td>
<td>194</td>
<td>370</td>
<td>350</td>
<td>471</td>
</tr>
<tr>
<td>200</td>
<td>396</td>
<td>503</td>
<td>396</td>
<td>526</td>
</tr>
</tbody>
</table>

Figure 5.15: Runtime performance metrics of JSSGLR parser, all measurements in ms

In the new design, when the worker is created, the JavaScript file containing the parse table is loaded with the other worker code. This can be seen in Figure 5.12(b) as the time to load the worker increases with the size of the parse table. With the new approach, the parse table must be initialized, the function that create the parse table AST must be called, before it can be used. As expected, the time required to initialize the parser increases as the size of the parse table increases.

To put the different languages in perspective, Figure 5.13 shows several properties of these languages. When documents are transferred from a server to a browser they are usually compressed with the gzip algorithm. For this reason we have included the sizes of both the parse table and JavaScript file after gzipping. We have also listed the number of productions, state transitions (goto), and states for each language. These features are representative of the size and complexity of each language.

If the server uses gzip compression to transfer the JavaScript files, even a large language like WebDSL will load in five seconds. Generally though, languages are not quite that large, in that respect the Go or Stratego language are a better representative.

### Runtime performance of JSSGLR

To evaluate the runtime performance of the JSSGLR parser we instrumented the plugin in a similar manner as before. We recorded the current timestamp when the change is sent to the worker, and compute the difference when the results come back from the worker thread.
Even for very small files, results were returned in a little over 500 milliseconds. This 500 millisecond limit is imposed by a timeout in the mirroring code. Rather than running the parser each time a change is made, all changes made within 500 milliseconds are grouped together. This is advantageous since generally many small updates are made in success as the user types in new code. To improve the reliability of our measurements we reduced this timeout to 5 milliseconds. Additionally, in our testing we made changes to the document by pasting in text to ensure that all changes are introduced as one batch. We used the snippet from Figure 5.14.

Figure 5.15 illustrates the relation between document size and response time. To perform this benchmark we used the WebDSL parse table and repeated the var block. We decided to create a separate column for measurements performed on a document containing errors, as we noticed that errors cause the response time to increases significantly. This increase in parse time is caused by the error recovery [24] algorithm, an important aspect of an interactive editor as it facilitates syntactic and semantic analysis when the user is typing. The introduced error was the removal of semicolon after the last var block. From these numbers we conclude that performance is acceptable as long as the document size remains small. While performance degrades as the document size increases, it is still in sub-second even for larger documents. A solution to this problem would be to use incremental parsing.

5.4.2 Stratego-to-JavaScript performance

We have measured the performance of binary generated by the three different Stratego compilers, the Stratego-to-C compiler (strc), the Stratego-to-Java compiler (strj), and our Stratego-to-JavaScript compiler (strjs). Since JavaScript does not support file input, our benchmark could not use any files as input vectors. Additionally, the JavaScript compiler does not have an efficient mechanism to embed input vectors. We decided to use a simple recursive implementation of the Fibonacci function for our benchmark. The Stratego snippet used with all three compilers can be seen in Figure 5.16.
5. Performance

<table>
<thead>
<tr>
<th></th>
<th>time (s)</th>
<th></th>
<th>time (s)</th>
<th></th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.023</td>
<td>25</td>
<td>0.468</td>
<td>25</td>
<td>0.338</td>
</tr>
<tr>
<td>26</td>
<td>0.036</td>
<td>26</td>
<td>0.500</td>
<td>26</td>
<td>0.505</td>
</tr>
<tr>
<td>27</td>
<td>0.053</td>
<td>27</td>
<td>0.508</td>
<td>27</td>
<td>0.767</td>
</tr>
<tr>
<td>28</td>
<td>0.083</td>
<td>28</td>
<td>0.576</td>
<td>28</td>
<td>1.124</td>
</tr>
<tr>
<td>29</td>
<td>0.132</td>
<td>29</td>
<td>0.604</td>
<td>29</td>
<td>1.794</td>
</tr>
<tr>
<td>30</td>
<td>0.213</td>
<td>30</td>
<td>0.694</td>
<td>30</td>
<td>2.767</td>
</tr>
<tr>
<td>31</td>
<td>0.344</td>
<td>31</td>
<td>0.870</td>
<td>31</td>
<td>4.533</td>
</tr>
<tr>
<td>32</td>
<td>0.577</td>
<td>32</td>
<td>1.133</td>
<td>32</td>
<td>7.363</td>
</tr>
<tr>
<td>33</td>
<td>0.921</td>
<td>33</td>
<td>1.539</td>
<td>33</td>
<td>11.590</td>
</tr>
<tr>
<td>34</td>
<td>1.397</td>
<td>34</td>
<td>2.072</td>
<td>34</td>
<td>18.808</td>
</tr>
<tr>
<td>35</td>
<td>2.310</td>
<td>35</td>
<td>3.092</td>
<td>35</td>
<td>30.115</td>
</tr>
</tbody>
</table>

(a) Compiled with strc compiler, ran as native program
(b) Compiled with strj compiler, ran under java 5
(c) Compiled with strjs compiler, ran under node

Figure 5.17: Raw performance of programs generated by the Stratego compilers, measured with a recursive implementation of $Fibonacci(n)$ for a varying input $n$

Note that $n$ was replaced with values 25 through 35 for each compiler. We chose the 25 through 35 range because values of $n < 25$ resulted in very short runs, which make for unreliable performance measurements, while values of $n > 35$ resulted in very long runs. When referring to the performance of a particular compiler, we mean the performance of the binary it generated. The binary generated by the strj compiler is a Java program and the binary generated by the strjs compiler is a JavaScript program. Generally Java and JavaScript code are not considered binaries, however, since they are the output of their respective compilers, they are binaries in this context.

Figure 5.17 shows the timing data for each compiler. We measured only the run-time of the resulting binary, compilation time was not included. As expected the performance of the strc compiler (Figure 5.17(a)) is best of all three. The strc compiler has been heavily optimized, over the course of many years, targeting a language very suitable to optimizing. The strj compiler on the other hand (Figure 5.17(b)) must deal with the overhead of the Java Virtual Machine (JVM). While it is not quite as mature as the strc compiler, much effort has gone into optimizing the strj compiler.

While both the strc and strj compilers have been heavily optimized, the strjs compiler is not optimized at all. Most of the code of the strjs compiler is similar to that of the strj compiler. As such, for much of the development of strjs, the strj code was usable as example. The optimizer for the strj compiler on the other hand, is directly targets the Java language. Because of this, the strjs compiler currently lacks an optimizer other than those provided by the front end.

For this benchmark we turned off the generation of stack traces and set the optimization level of the GWT compiler to the highest setting. As can be seen in Figure 5.17(c) though, the strjs compiler does not keep up. While the overhead of the strjs compiler is not immediately obvious for low $n$, it becomes more obvious as $n$ increases. With each increase in $n$
the overhead of the strjs compiler is compounded, as a result it is quickly outpaced by both the strc and the strj compiler.

**Discussion**

With the performance improvements we made we greatly reduced the time the user must wait before a useful result is produced. We achieved this mainly by using the native JavaScript parser of the browser to load the parse table, rather than parsing a different format in JavaScript. This technique is applicable to other data formats that can be represented by native language elements. For example, in Java the parse table could be shipped as a `.class` file which is then loaded at runtime.

We reduced the start-up time of the JSSGLR parser by moving initialization to a web worker. For example, in a very large language such as WebDSL, we reduced the load time down from 15 seconds to 5 seconds. This technique, offloading processor-intensive work to a separate thread is applicable in many fields.

Additionally, by sharing common data structures and minifying variable names, in addition to using gzip, we reduced the size of the largest parse table to a mere 292kb. Given the reduced size of the parse table, we conclude that performance of the plugin is acceptable for regular use. We recommend the use of incremental parsing to reduce the parse time for large documents.
Chapter 6

Related Work

The browser environment, while relatively new, is a well explored domain in computer science. This chapter discusses technologies related to compilation targeting JavaScript. It is divided into four sections. First we discuss HTML Templates that Fly [33], a paper discussing a technique to improve server performance. Second we discuss Links [11], a language that generates JavaScript code. Third we discuss Hop [30, 31], a language for writing interactive web applications. Last, we discuss Hilda [1] a language for developing data-driven web applications.

6.1 HTML Templates that Fly

In HTML Templates that Fly, Michiaki Tatsubori and Toyotaro Suzumura [33] present a new technique to improve the performance of web servers, providing a speed up in between 1.6x and 2.0x. Their technique involves the separation of page rendering and data retrieval. Since most web applications already separate the definition of template and data by means of the model-view-controller pattern, their solution is easy for developers to implement. Easy, because most model-view-controller patterns for web applications involve some sort of template which is then used to render a HTML page.

The proposed solution skips the rendering step, and instead returns a skeleton page to the client. This skeleton page contains the data that should be rendered, and a reference to the template that should be used. The client fetches the templating library and template, and renders the page on the client side using the data provided in the skeleton. There are very few changes on the server side, the programmer simply uses a different templating library, e.g., the FlyingTemplate library proposed in the paper. The benefits to this approach are twofold, the templates and templating engine can be cached by the client and as such need to be transferred only once, and the server is now only responsible for serving static files and data, reducing the server load.

A direct consequence of this approach is that template data is available to all users, more or less regardless of their level of access to the system. Other papers, such as Swift, go to great lengths to offload only that work to the client that the client is authorized to. The FlyingTemplates engine takes a different approach, and instead requires that template data
6. RELATED WORK

to be public. In practice this is not much of a problem, most templates are already public and only the data that is passed to the client has to be verified.

In order to get optimal caching the authors investigate the caching behavior of the most popular web browsers currently in use. Their conclusion is somewhat saddening, while it is possible to get most browsers to cache the templates through the combination of various techniques, there is no single straightforward approach that works reliably on all browsers. In order to get the best results, multiple, somewhat obscure techniques need to be combined.

The problem of caching is complicated further by the need for cache invalidation, when the templates change on the server side, the client should stop using the old templates. While there there is a mechanism to do just this, by means of the If-Modified-Since request header, this is not very efficient. Instead, the authors propose to instead turn off this cache invalidation, and manually invalidate templates by means of a client side script. If the skeleton template includes the version number of each used script, the client can validate that they have the correct version, or fetch a new version if they have a stale template.

Using the SPECweb2005 benchmark, the authors have verified that their approach is indeed an improvement on the existing situation. They find that through the use of their system server load decreases drastically, showing a performance improvement between 59% and 104%, depending on the number of clients. Additionally, they show that even if the client fails to cache the templates the throughput is at least equal to that of the old system. The only caveat is that throughput suffers significantly if the templating engine is not reliably cached. The paper concludes with a comparison with related in the field of security, function shipping, and client-side templates.

In ‘Adinda: a knowledgeable, browser-based IDE’ [12], van Deursen et al. propose an Integrated Development Environment (IDE) that runs on the server, but is rendered on the client. In this thesis we have described an IDE that is run entirely on the client, the server consists of a static file server. A hybrid approach that applies the principles set forth in ‘HTML Templates that Fly’ might consist of a server component that performs the tasks executed in separate thread in our approach. Since all communication between the main thread and the worker occurs over a socket in our design, this approach would be easy to apply to our design. By setting up a socket between a state-full server and the client, the proposed approach would remove the computation burden from the client and move it to the server, while letting the client perform all rendering operations.

6.2 Links: Web Programming Without Tiers

In Links: Web Programming Without Tiers, Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop [11] present a language that generates code for all parts of the web serving process: JavaScript code for in the browse, O’Caml for the server, and SQL for the database. This language, Links, aims to combine the best practices of functional programming as proven by other languages such as Kleisli, PLT Scheme, and Erlang.

In order to ensure that programs written in Links scale to many users, all session state is stored in the client using an extension of the continuation passing style found in PLT Scheme. An advantage of this technique is that the server need not store any session state
on the server, sidestepping problems such as when to expire session state. Additionally, this solution facilitates the emulation of the server evoking a function on the client by passing a continuation to the client which is then called with the result of the function call. An obvious problem with this approach is that server state between requests is stored in the clear on the client side. This research topic is currently being investigated by researchers at Microsoft and the University of Maryland.

Links has supports for concurrent programming, and uses the communication asynchronous style of exchanging messages, borrowed from Erlang. The authors provide a translation of their message passing style to the extension of the lambda calculus by Niehren et al. This illustrates that the typed message passing technique in Link can be simulated on any concurrent language with support for typed channels. In addition, they demonstrate that synchronous message passing can be simulated on top of the asynchronous in Links by means of blocking the sender on a Reply message from the receiver. In contrast with Erlang, messages in Links are typed and as such the message passing concept does not apply cleanly. The type of each message sent by a process needs to be included in the type signature of the process.

To facilitate the usage of Links in writing web pages, XML fragments can be directly included in the source code. In order to include dynamic elements in the generated fragments, curly braces can be used to escape to Links expressions. This does mean that the less than comparison operator must always be followed by a space, but such is considered good style and is therefore not a significant problem. Additionally, Links has a special syntax to allow embedding XML fragments that are not contained in one XML tag. All generated XML maps directly to the Document Object Model in the client browser as specified by the W3C.

Handling of form input is specified at form declaration time by means of special attributes added to input and form tags. Normally, expressions surrounded by curly braces are evaluated immediately, but if those expressions are included in certain relevant attributes such as the onkeyup, the code is instead executed when the associated event is fired. In order to refer to the value of input fields in the code that handles the fired events, input elements are annotated with a name attribute. The Links compiler statically checks that such variables are in scope when they are used. This approach does pose a significant problem, as each field is named statically by the programmer at compile time, it is impossible to include a form snippet twice while at the same time referring to the elements it contains from the enclosing scope. This precludes the creation of libraries with form templates.

Database queries are written in Links own syntax and are compiled to SQL at compile time. As such, database queries are type checked statically and SQL injection is not a problem. Links exposes several SQL specific constructs such as ORDER BY and LIKE directly in its query syntax and tries to optimize queries by using LIMIT and OFFSET statements.

The Links language supports embedding XML fragments in the language itself. Had Links been developed using the Syntax Definition Formalism (SDF), this embedding would have been trivial to implement. Given two languages, SDF can arbitrarily compose the two by embedding a language element of one in the other. In the case of Links, classifying XML Element as a value would enable the use of XML as a value without any further work.
6.3 Hop: a Language for Programming the Web 2.0

In Hop: a Language for Programming the Web 2.0, Manuel Serrano, Erick Gallesio, and Florian Loitsch [30] present a language for writing interactive web applications. Their language, Hop, enables both the creation of web applications that communicate with a server, as well as applications that run solely on one computer. It creates a strict separation between the application logic and the graphical interface, enforced by a different syntax and API for each component.

The two different layers in Hop can communicate and exchange data, but are defined in the same file, easing deployment. The logic layer is in control when the program starts and can call into the GUI layer transparently through the use of an escaping syntax. The GUI layer can call functions in the logic layer through a similar escaping mechanism.

The language has a syntax Scheme-like syntax, but the standard widget library includes functions for HTML tags, lending it a more HTML-like feel. The logic layer evaluates these function calls and generates the appropriate HTML to be displayed to the user. The logic layer has access to server-specific facilities such as the file system, conversely the GUI layer has access to client-specific features. The logic layer can send the client signals due to the event-driven nature of Hop. The client, rather, communicates with the server by making remote procedure calls.

In Hop, the response is treated like a tree that is being constructed on the server by means of function calls. As such, it is possible to abstract over implementation details, such as HTML rows, by defining a function that constructs a new part of the tree, such as a new row in an HTML table. When the logic layer escapes to the GUI layer the relevant snippets are sent to the client, which then evaluates them and renders the resulting document accordingly. If in the escaped expression an escape back to the logic layer occurs, it is evaluated at the same time as the original expression, and sent along to the client.

Services are defined and bound to a particular url in one method call, making it obvious to the programmer to what url a service corresponds. These definitions also serve as variables to which can then be referred in other services to create internal links. Alternatively, services can be defined inline to allow the action of a form to be defined inline in the form declaration. Both these services and the client GUI layer can exchange compound values with one another in addition to regular HTML constructs. Additionally, the GUI layer can call to other websites that are not powered by Hop whose response they can then handle as a regular XML document.

Two different event loops are implemented in Hop, one using simple timeouts, and one based on the logic layer sending signals to the GUI layer. When the GUI layer registers as a listening to events from the server, it long-polls a specific url on the server that returns a result only when one of the registered signals is fired. By implementing signals this way, the GUI thread does not enter a busy-wait, but only does work when a signal is fired.

Using a Hop server the design of an IDE would be similar to the architecture of the Spoofax language workbench. By explicitly defining services the programmer creates a reusable design in which the various can be used separately, or together, depending on what the service needed. The same advantages presented by van Deursen et al. are applicable with this approach.
6.4 Hilda: A High-Level Language for Data-Driven Web Applications

In Hilda: A High-Level Language for Data-Driven Web Applications, Fan Yang, Jayavel Shannugisundaram, Mirek Riedewald, and Johannes Gehrke [1] present a high-level language for developing data-driven web applications. Their language, Hilda, uses a unified data-model for all layers of the application, is declarative, and supports structured programming. Its modelling language can be used for both querying and updating, and enabled conflict detection for concurrent updates.

An important feature in Hilda is are AUnits, which can be likened to UML classes. AUnits are created and manipulated declaratively and facilitate conflict detection. AUnits can use inheritance to derive website AUnits from application AUnits.

The Hilda control flow resembles that of structured programming, with a hierarchical model. Each AUnit has zero or more Activator, and zero or more child AUnit instances. The activation and deactivation logic of child AUnits is handled by an Activator. Using an activation query, the Activator can create child instances. Then, with an input query, the activator prepares input for the child. When a child AUnit returns, its activator handler processes its output. Each child recursively activates its own child instances, resulting in an activation tree. Optionally, an Activator may declare an activation condition, which it can use to validate its state in the face of concurrent updates made by other AUnits.

The Hilda data model, specified declaratively using SQL statements, is used to represent all application state, from database to application logic to client. An AUnit can define an input and/or output schema. In addition to AUnits, which are used solely for application logic, Hilda features PUnits, which use embedded HTML for presentation. Hilda features both a set of built-in AUnits, called Basic AUnits, and external AUnits, called External AUnits. External AUnits can be written in an imperative language, such as Java.

Hilda maintains a conceptual separation between client and server. The compiler is however allowed to make decisions based on optimization criteria to push logic from server to client.

The separation of presentation and logic presented in this paper is similar to our work. This separation is vital in assuring a satisfying user experience, in which the user interface does not become unresponsive. Ensuring that the presentation layer and the logic layer are separate simplifies a design in which the logic layer runs in a separate thread. Since the compiler is in control of the communicating between the presentation layer and the logic layer, the programmer does not have to make any effort to facilitate such a design.
Chapter 7

Conclusions and Future Work

This chapter gives an overview of the project’s contributions. After this overview, some ideas for future work will be discussed. Finally, we will reflect on the results and draw some conclusions.

7.1 Contributions

We developed a fully functional library implementing the SGLR algorithm in JavaScript. This library can be used by any web application to parse a document for which an SDF language definition is available.

We have created an architecture that facilitates the creation of parser based editors on the web by means of an editor specific plugin and a generic parser. The ACE plugin we have developed is capable of parsing any language defined in SDF and is an example of this architecture. Additionally, we have designed the plugin such that the CPU-intensive components run in a separate worker thread.

We have ported Stratego, a language specialized in semantic analysis, to JavaScript so that it can be used in the browser. Using this language, developers will be able to extend the available editor services with semantic analysis. The code generated by the Stratego-to-JavaScript compiler can be executed by any JavaScript interpreter.

With the performance improvements we made we greatly reduced the time the user must wait before a useful result is produced. The main technique used, to natively parse the parse table, can be applied to other languages. For example, in Java the parse table could be shipped as a .class file which is then loaded at runtime. We greatly reduced the start-up time of the JSSGLR parser. For example, in a very large language such as WebDSL, bringing it down from 30 seconds to 3 seconds.

7.2 Future work

Although we have created a set of tools and abstractions that facilitate the development of an Integrated Development Environment (IDE) that runs in the browser, several improvements are still possible. The Stratego-to-JavaScript compiler that facilitates semantic analysis is
7. CONCLUSIONS AND FUTURE WORK

not yet integrated with the IDE. In order to achieve this a shared framework between the parser and compiler needs to be developed that will allow the compiled Stratego code to use the output of the parser.

We have mentioned that the Spoofax language workbench served as inspiration for our design. The main feature of Spoofax though, facilitating the development of new languages, is not yet possible on the web. To realize this functionality much of the work that is currently performed on the client side would have to be ported to the browser. For example, the creation of the parse tables must currently be done with the Stratego-to-Java compiler.

We have performed several benchmarks to measure the performance of the work proposed in this thesis. These benchmarks show that while our approach is already viable, more work is needed to make it generally useful. In particular, the use of incremental parsing would increase the usefulness of our plugin for larger documents. Additionally, our benchmarks have shown that there is a lot of room to optimize the Stratego-to-JavaScript compiler.

7.3 Conclusions

The goal of this thesis is to provide a set of tools and abstractions that facilitate the development of an Integrated Development Environment (IDE) that runs in the browser. We have addressed the problem of insufficient abstraction by further separating concerns. We have split out the language specific elements from the editor into a language independent architecture. The abstractions from desktop environment that are known to work have served as an inspiration for this design.

We have developed a compiler-back end for JavaScript that allows us to reuse the Stratego language. This solution takes the form of a plugin for ACE, such that many different language components can be plugged in without changing any of the other editor code. With this plugin meta-programmers now have a declarative way of specifying editor services for arbitrary programming languages. These editor services include syntax highlighting, and syntax checking, with a clear path to adding other such services as type checking and code completion.

The development of a plugin for ACE, provides us with insight in the difficulties involved with development targeting the browser. Firstly, compiling a Java library to JavaScript with GWT is feasible, but it is clearly not the use-case for which GWT was designed. Secondly, bandwidth and processing power constraints require significant optimization of SGLR parse tables in the form of a native JavaScript format. Finally, JavaScript as a target for a Stratego compiler runs into many of the same problems, such as bandwidth and processing power require the implementation of static linking to reduce output size.
Bibliography


[34] Masaru Tomita. “LR Parsers For Natural Languages”. In: *COLING*. 1984, pp. 354–357. DOI: [http://acl.ldc.upenn.edu/P/P84/P84-1073.pdf](http://acl.ldc.upenn.edu/P/P84/P84-1073.pdf).


Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

**ACE**: Ajax.org Cloud9 Editor

**API**: Application Programming Interface

**AST**: Abstract Syntax Tree

**ATerm**: Annotated Term Format

**BAF**: Binary Aterm Format

**GLR parser**: Generalized left-to-right parser

**GPL**: General-purpose programming language

**GUI**: Graphical User Interface

**GWT**: Google Web Toolkit

**IDE**: Integrated Development Environment

**JSGLR parser**: Java Scannerless Generalized Left-to-right parser

**JSSGLR parser**: JavaScript Scannerless Generalized Left-to-right parser

**LR parser**: Left-to-right parser

**OS**: Operating system

**regexp**: Regular expression

**SDF**: Syntax Definition Formalism

**SGLR parser**: Scannerless Generalized Left-to-right parser

**SLR parser**: Scannerless Left-to-right parser

**XTC system**: Transformation tool composition system