Generating Web-based Semantically Aware Source Code Editors

Master's Thesis

Richard G. Vogelij

Generating Web-based Semantically Aware Source Code Editors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Richard G. Vogelij
born in Vlaardingen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Generating Web-based Semantically Aware Source Code Editors

Author:	Richard G. Vogelij
Student id:	4052676
Email:	richard@vogelij.nl

Abstract

This thesis describes spoofax2ace, the tool to generate source code editors which can run in the browser. The features which are common in state of the art desktop-computer based source code editors are investigated after which the difficulties in getting these features running on the Web are discussed. We present, implement, and compare multiple approaches in constructing a fully semantically aware source code editor which runs in the browser. The most useful aspects in these approaches are combined in the proposition of our "editor-generator" which produces browser-based source code editors with as sole input a language declaration in the form of a Spoofax project.

Thesis Committee:

Chair:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. L.C.L. Kats, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Pinzger. Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Iosup. Faculty EEMCS, TU Delft

Preface

I would like to thank my supervisor, Eelco Visser, for “bringing me in” and providing me with this project. Also, I would like to thank Lennart Kats and Karl Trygve Kalleberg for always being available to answer any questions I had along the way. I have had some very interesting conversations which broadened my horizon greatly. Also, having worked closely with Eelco, Lennart and Karl on a paper: “Software Development Environments on the Web: A Research Agenda” [38] which got accepted for the Onward! Conference 2012 has been a great experience.

Finally I would like to thank my girlfriend, parents, and family for their ongoing interest and support in the road to completing this thesis.

Richard G. Vogelij
Delft, the Netherlands
December 4, 2012

Contents

List of Figures

Chapter 1

Introduction

Ever since the the World Wide Web (The Web) was introduced it has become an important source of information and interaction. Most of its content is added through web-applications which allow increasingly easy access to the various assets the Web has to offer. Regretfully, the tools which are used to *create* applications with are not yet universally available on this platform. Even though there are a number of online code editors, most of them support the programming languages they are built for very generically. A lot of feedback programmers have grown accustomed to in desktop environments such as type checking and reference resolving, explained in Section 1.3, can not be taken for granted in typical web based source code editors. This lack in availability of programming tools is mainly because on the Web the technologies to create client-side programs are very limited and server-side resources can become costly.

1.1 The Web

The Web is in its core built on three main technologies:

- URI (Uniform Resource Identifier) [6]
- HTTP (HyperText Transport Protocol) [20]
- HTML (HyperText Markup Language) [50]

These mostly static content enabling technologies still provide the building blocks on which the Web is built. Since the introduction of richer client-side technologies such as *JavaScript*, *Cascading Style Sheets* (CSS), the *Extensible Markup Language* (XML) and *Asynchronous JavaScript And XML* (Ajax) [46] web pages became greatly more dynamic and interactive. This added interactiveness introduced the Web as it is implemented today, the user-generated Web, or Web 2.0. Practically all websites which currently exist make use of these Web 2.0 technologies and allow a degree of interaction with a web page visitor. Such interactions can vary from executing a search query [4] at the Google website to logging in and posting messages on a forum.

Web Applications

Many of the current websites can rightfully be called applications as they can in principle provide nearly the same, and sometimes much more, functionality a traditional desktop application could. Take for instance Google Documents or Google spreadsheet in Figure 1.1 which allows a user to create and edit spreadsheets fully on the Web. It features the same functionalities Desktop spreadsheet applications such as Microsoft Excel provides.

Figure 1.1: A Google Spreadsheet instance running entirely in a web browser

1.2 Developing Web applications

There are some fundamental differences when designing and implementing an application which will run on the web compared to classical desktop applications. In typical web applications the actual work is done remotely on a web-server or the cloud where the user is presented with a user interface built in HTML. Through the use of GET/POST requests or AJAX communication is handled from the client to the server. This communication layer with the back-end is arguably where most differences between desktop and web applications lie because of its inherent asynchronous nature. At the server side a programmer has virtually unlimited options in which he implements the web application back end. However at the client side the web application has to be presented in a web browser. Currently this means the implementation is bound to only use flavours of (X)HTML, CSS and JavaScript.

Even though many Web applications have been created by software developers, there currently are few web applications which provide the necessary tools to actually create applications with. The small amount of tools which do exist, such as CoRED[41, 2] and

Cloud9 [1], are fundamentally limited in the sense that they only support a select set of languages. Even though Cloud9 supports language plug-ins, these plug-ins still have to be implemented specifically for that platform (in JavaScript) and are mainly implemented using regular expressions which make sophisticated editor feedback impossible. In this thesis will focus on *generating desktop quality code editors which can run in the browser*, based on the higher level definition of the syntax and semantic constraints of a programming language.

1.3 Regular Integrated Development Environments

IDE stands for Integrated Development Environment and typically consists of a set of tools which work together to provide a means to positively influence the productivity of an application developer. This set of tools is integrated into a single application, an IDE. Even though many IDEs frameworks currently exist such as Eclipse [21], Netbeans [7] and Visual Studio [48], most IDE implementations are mainly targeted at a small fixed set of programming languages. Modern IDEs do however usually provide a way to add language specific functionality in the form of plug-ins. Implementing such a plug-in for a previously unsupported language is however a difficult task [11, 21].

We can split the functionalities of any given IDE into two groups, namely language specific and non-language-specific. The latter consists of features such as version management support, searching/replacing in files, keyboard shortcuts and so on. In this thesis we will focus on the *language specific* aspects of an IDE.

1.3.1 Language specific aspects

In this subsection we will identify the components in a typical IDE which are, or are based on, language specific tooling.

A modern IDE provides a number of common features which are seen in most desktop based IDEs. In its core functionality obviously lies the ability to write or modify source code (textual of visual). Usually the source code editor in an IDE differs from a normal text editor in the amount of feedback a programmer gets while adapting or writing source code. This feedback is possible thanks to a back-end which "understands" the source code to some degree. Such understanding is achieved through the use of a parser which checks the syntax, described in Section 2.2, and possibly even a semantic analysis, described in Section 2.3.

In most allround text editors, including the ones intended for editing source code such as Notepad++ [2] and UltraEdit [3], a basic form of language specific feedback is achieved

[1] `http://c9.io/site/features/`
[2] `http://notepad-plus-plus.org/`
[3] `http://www.ultraedit.com/`

through the use of regular expressions. These editors use a regular expression based approach to highlight keywords and for instance make sure an opening bracket is followed by a closing bracket.
While this is acceptable for an allround code editing tool, the editor an IDE provides should provide more meaningful feedback. The following features can in principle only be implemented when the editor has access to a deeper "understanding" of the language in the form of semantic knowledge:

- **Outline**
  A code outline gives a brief summary of a piece of code which is being edited. It can for instance show all method names in a class and usually also provides a way to quickly navigate to a portion of code.

- **Folding**
  Code folding is used to hide and show a region of source code. The editor has a syntactic knowledge of the program which is being edited which allows a block of code to be collapsed with a single click. This can for instance be used to hide the body of methods in a class to improve the readability of the code.

- **Hover help**
  Hover help is shown when a programmer hovers his mouse over a piece of source code. Useful information can be shown such as the (possibly inferenced) type of a variable or an explanation regarding a syntactic keyword.

- **Error marking**
  When an error occurs, the IDE can provide useful feedback regarding the origin of an error. This can be any type of error, ranging from a syntactic problem to uncompilable code due to the use of an erroneous type. In advanced IDEs, error marking is also often used to provide hints to a programmer when a code smell (common bad practise) is detected.

- **Reference resolving**
  Reference resolving is used to determine the origin of a piece of code. This can be the interface a class is implementing, the definition of the type of a variable and so on. Usually this is implemented using control+click, where the text beneath the mouse pointer becomes a clickable link. When this link is clicked, the editor jumps to the relevant piece of code.

The editor services shown in Figure 1.2 are made possible due to language specific knowledge. We split up the "understanding" an editor can gain about a program into two concepts. Syntax, explained in Section 2.1 and semantics, explained in Section 2.3.

1.4 Research Questions

We have formulated the following research questions which will be answered in this document.

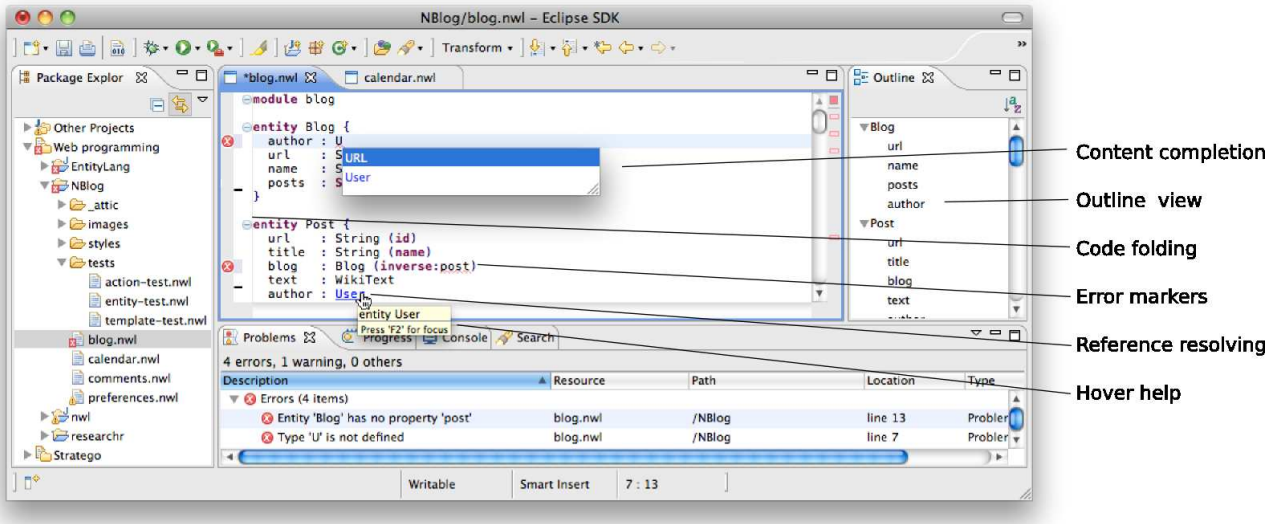

Figure 1.2: Editor services in a desktop-based IDE (from [37]).

1. Can a web-based, Desktop IDE quality, source code editor be created for any context-free programming language?

2. Is generating a semantically aware Web based source code editor feasible?

1.5 Previous work

Previous work has been done at Delft University which lays the groundwork for some of the work we will present in this thesis [49, 59]. Especially the JSSGLR parser implementation described in Section 4.4 is leaning heavily on this work. This work is highly specific and in order to keep a chronological order in which we will present this thesis we explain the previously done work in the sections where this is relevant instead of in an introduction or related work section.

1.6 Outline

We will begin to identify the various components a general IDE consists of. These components will be briefly discussed and we will attempt to identify the components which change between multiple programming languages, the language/specific components in Chapter 2. We will further divide these language specific components into their respective groups and elaborate on the theory behind these components in Section 2.1. Finally we will introduce the language-workbench Spoofax in Section 2.4.1 and explain the steps which are required to generate an IDE using definitions in SDF and Stratego. Next, we will enumerate the difficulties which rise when designing an application specifically for the Web and mirror

these difficulties to the implementation of the previously discussed IDE components in a Web environment in Section 3.1. We will continue to enumerate a number of deliverables we will produce in the form of benchmark results, proof of concept implementations and tooling in Section 3.2.

We will present our work in designing, and implementing, a *client-side* based approach to perform the static analysis for our semantically aware Web-editor. In Chapter 4 we will introduce NodeJS, the architecture in which we will execute our JavaScript based benchmarks. We use GWT (The Google Web Toolkit) to transform the Java implementation of the SGLR parser, JSGLR into JavaScript and compare the performance of the Java and JavaScript implementations in Section 4.4. In Section 4.5 we will introduce our first editor back-end which runs fully on JavaScript. Next we will explain s2js, the Stratego to JavaScript compiler in Section 4.6 and provide a comparison of performance between the GWT and s2js result implementations. Our second major approach is a *client-server* based approach in Chapter 5. In this approach the actual syntactic and semantic calculations are performed server side. We will identify the problems and propose ways to overcome these, while presenting performance statistics comparing the various paths we have tried.

In Chapter 6 we introduce *spoofax2ace*. Using this tool an implementation of a semantically aware Web editor can be generated based on one of the earlier discussed approaches. The input for this tool is a Spoofax project. The final addition we have made is a balancing algorithm which aims at automatically switching between the various approaches we have defined. In Chapter 7 we will go into a number of possible areas of future research and work and we will present some related work which is currently being done in Chapter 8. Finally, in Chapter 9 we will briefly summarize our work, and discuss our conclusions and contributions.

Chapter 2

Language specific IDE components

2.1 Syntax

The syntax of a programming language specifies the requirements source code must meet in order to be part of the set of all possible programs for that programming language. For programming languages, or formal languages, the syntax is defined as a set of rules which pose constraints on which symbols can be used together in some specific order. Such symbols can be non-terminal and terminal. Analogous to natural languages, terminal symbols are the building blocks which are used to define words, so they can be in a sense thought of as letters. Non-terminal symbols are used to represent variable terminal or other non-terminal symbols. Combining these symbols is done through production rules. A production rule consist of a left-side, an arrow and a right side. On the left-side there has to be a non-terminal symbol which can be replaced into an at the right-side defined order of terminal and non-terminal symbols [10].

A formal grammar can be used to define the syntax of a formal language using a set of *terminal symbols*, *nonterminal symbols* and *production rules* [10]. In [35] a solid case is made in favor of using SDF in [63, 26] to represent these three elements which are required to define a formal language. Such a formal grammar definition can in turn be used to *parse* source code in order to for instance determine whether the source code is consistent with the syntax of the language or to produce an abstract syntax tree (AST).

2.2 Parsing

In order to provide useful syntax based feedback, the target source code has to be parsed. Much research has been done in order to design optimal parser implementations. Most parsers are implemented using a lexer which forms a layer between the actual input and the parser implementation. There are two main strategies in implementing a parser, namely LL based, discussed in Section 2.2.1, and LR based, discussed in Section 2.2.2.

Either of these parsing strategies use a parse table in some form. Possibly encoded in

an (automatically) derived parser implementation or an explicitly defined table. In essence a parse table contains information a parser needs in order to make decisions based on a current symbol. LL and LR parsers can be defined in a so called LL(k) or LR(k) manner where k indicates the number of lookaheads the parser implementation can use. [1, 47, 16]

2.2.1 LL Parsing

An LL parser parses input from **L**eft to right in a topdown fashion, constructing a **L**eftmost derivation. LL parsers can fundamentally not recognise the full set of regular grammars. The grammars an LL parser can recognise is however a subset of the context free grammar space and is called the *LL grammar space*. Because of the topdown approach, starting at a special start symbol is a somewhat intuitive approach which makes implementations of LL parsers quite popular due to its relative simplicity. Another major upside of a topdown approach is easy error recovery; It is known what is expected next, if an expected token is not there an error can be marked and the parsing can continue in a straightforward manner by acting as if the missing token was found. Also, multiple LL grammars can not be composed to form a new language. [40, 51]

2.2.2 LR Parsing

An LR parser uses an opposite approach compared to LL parsers. LR parsers read input text from **L**eft to right, but utilize a **R**ightmost derivation, meaning the grammar rules are applied bottom up. The parsing ends at a start symbol rather than starting at one. Because of this the parser can handle ambiguities more easily than LL parsers, allowing them to recognise the *LL grammar space*, but also some languages LL parsers cannot recognise. Even though LR parsers can recognise more languages than LL parsers they are much more complicated and counter intuitive to implement. We will call the grammars an LR parser can recognise the *LR grammar space*. In Figure 2.1 we have used Chomsky's representation of grammars to show which set is a subset of another set. Compared to LL parsers, it is also difficult to implement general error recovery because of the bottom up nature which makes the naive solution of simply continuing more complicated as there might be multiple "paths" up. As is the case with LL parsers, a typical LR parser cannot recognise ambiguous grammars. Multiple LR grammars can however be *composed* to form new languages, enabling sub languages in a main language. In LR definitions it is for example possible to define SQL syntax rules within the Java syntax enabling the possibility of for instance a type safe Java+SQL hybrid language. [39, 1]

SGLR Parsing

The GLR parser [56, 43] is an extension of the LR parser algorithm to cope with ambiguous and non-deterministic grammars. This is done through a notion of parallelism by traversing the parse table in all possible manners when an ambiguity is found. This is done in a breadth-first manner.

In [60] and [57] the GLR parser and a number of other parsing techniques such as SLR [53] (The Scannerless LR parser, which eliminates the need for a lexing step) are combined

into SGLR introducing a parser which can recognise all context free grammars [22]. In Figure 2.1 we have emphasized the languages an SGLR parser can recognise with a gray color.

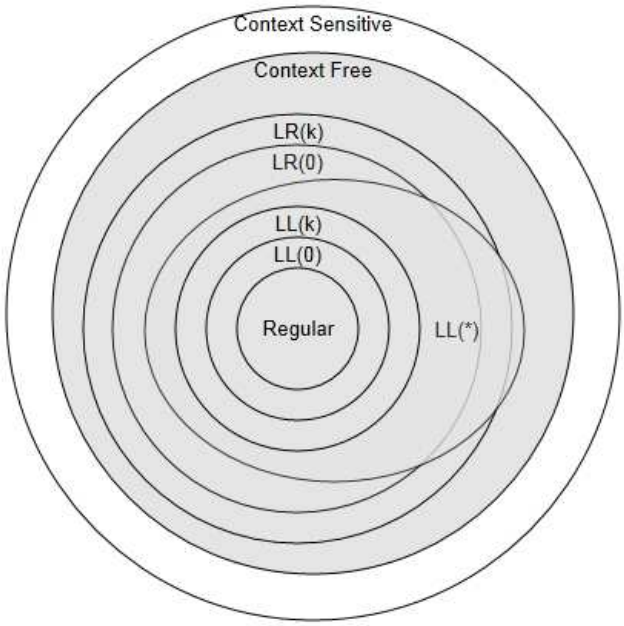

Figure 2.1: The various relevant grammar spaces for programming languages. Based on [35, 10, 56, 43, 57]

2.2.3 Parser implementation strategies

The general way of implementing a parser is by using a *parser generator.* A parser generator takes a definition of a target language and compiles it into an implementation of a parser. It is of course also possible to manually implement a parser [1]. This is in fact what is being taught at most compiler construction classes to familiarize students with the concept of parsing source code. Obviously, a major downside of such a low level approach lies in the fact a language designer is focusing more on the implementation of the parser instead of the syntax rules themselves. Also when extensions or alterations need to be made in the syntax definition of a language, changing the parser implementation might be close to impossible.

2.2.4 Parser generators

Parser generators are also known as compiler compilers. With respect to current compilers this name is not sufficient as a parser generator merely handles the syntactic aspect of a

compiler. There exist a number of parser generators such as ANTLR [1], Bison [2] and Yacc [31]. In this thesis we will focus on JSGLR [58, 32, 34] which is a Java based implementation of an SGLR parser, discussed in Section 2.2.2. JSGLR is not strictly a parser generator because the language specific dynamic aspects of JSGLR are handled through the use of an external parse table.

JSGLR

JSGLR is an SGLR parser implementation written in Java. It uses a parse table which contains all syntactic information for a target grammar. Such a table can be derived from an SDF [63, 26] (The Syntax Definition Formalism) based definition of a grammar. Since SDF has abstracted away from practically all implementation issues it is very easy to take a generative approach to design the syntax for a (new) language. Recently JSGLR has been extended with sophisticated error recovery functionalities allowing the parser to mark an error, and continue the parsing to still provide a valid AST. [13, 36] This makes JSGLR a very suitable back-end for an IDE because as a programmer is typing there will be many moments where the program under construction is not strictly syntactically correct. Due to this error recovery a semantic analysis which relies on the existence of an up to date AST can still be executed.

Furthermore, SDF and JSGLR lie at the core of the Spoofax Language workbench which allows for the generative development of a fully featured integrated development environment for programming languages. In Section 2.4.1 we will explain Spoofax and its relevant components in more detail.

2.3 Semantics

The semantics of a language go further than a language's syntax. While a syntax checker merely tests whether a program is well formed with relation to the syntax definition, there is no guarantee the program is actually runnable or even compilable.

A straightforward example would be the assignment of a variable identifier in Java of a variable that was not declared in the scope in which it was used. Syntactically the program would be perfectly valid, however a semantic analysis would pick this up and provide the programmer with a suitable error message. See Figure 2.2.

A semantic analysis also allows an editor to gain knowledge about context and be able to provide suggestions to the programmer about what keywords are expected next as he types (code completion). Of course, depending on the complexity of the language, implementing the semantic analysis is usually a non-trivial task. Take for instance the Java language in which inheritance is a major feature which greatly impacts the way a type compatibility check would be implemented. Refactoring functionality, for instance renaming all occurrences of some identifier in its scope is also in most cases a semantically aware operation.

[1] `http:/antlr.org`
[2] `http://www.gnu.org/software/bison/`

```
class Test
{
  public static void main(string[] args)
  {
    x = 5; //semantic error since x is not defined yet
    int x;
  }
}
```

Figure 2.2: Syntactically valid but semantically invalid code snippet

2.4 Language workbenches

The most widely used set of tools to implement domain specific languages (DSLs) is Xtext [18, 17, 5]. In Xtext the semantic aspects of languages are implemented in a set of goal specific domain specific languages such as a code completion language. Grammar rules are also entered in a specific DSL. A major downside of Xtext is that the parser (ANTLR [47]) used to transform source code into an AST is limited to the *LL grammar space*, discussed in Section 2.2.1. Because of this limitation, languages such as C++ can fundamentally not be implemented using Xtext, indicating Xtext's popularity by simplicity comes at a high price.

The alternative we have investigated, Spoofax, has arguably a steeper learning curve, but is also a lot more powerful. In the next section we will explain Spoofax in relevant depth and provide some examples of the various components Spoofax is based on.

2.4.1 Spoofax

The Spoofax language workbench is a toolset to build textual source code editors with. In essence, it is an Eclipse plug in to create Eclipse plug ins with. Eclipse [3] is an IDE written in Java aimed at being extensible by being very plug-in friendly. Using SDF definitions the full set of *context free languages* can be described. By declaring term rewrite rules in the Stratego language an AST can be traversed and manipulated. In the following paragraphs we will elaborate SDF and Stratego.

SDF

As mentioned in Section 2.2.4, an SDF definition can be converted into an SGLR parse table. The JSGLR parser implementation uses this parse table to build an abstract syntax tree based on ATerms. An ATerm is basically a tree node which can consist of ATerms itself. ATerms can carry additional arbitrary information as attachment. In JSGLR such ATerms contain for instance information about where in the source code that tree node originated from.

[3] http://www.eclipse.org/

SDF is used in Spoofax to define the syntax of the designed language. Spoofax takes care of the conversion of the SDF definitions to an SGLR parse table using an external tool which resides in the "SDF2 Bundle" [4]. When an SDF definition is updated and stored, the equivalent SGLR parse table is automatically derived in the background allowing the language designer to instantly see the effects his changes have made on a derived AST. This makes debugging and tweaking the syntax definition of the designed language a straightforward task.

In Figure 2.3 we have included the syntax definition for an example language, the Entity language, in SDF. In Figure 2.4 we have included a program in this example language

```
%% Grammar for the Entity language

module Test

imports Common

exports
  context-free start-symbols
    Start

  context-free syntax

    "module" ID Definition*      -> Start {cons("Module")}
    "entity" ID "{" Property* "}" -> Definition {cons("Entity")}
    ID ":" Type                   -> Property {cons("Property")}
    ID                            -> Type {cons("Type")}
```

Figure 2.3: The SDF grammar rules for an example language (the Entity Language)

and finally Figure 2.5 illustrates the AST an SGLR parser produces based on this example program and syntax definition.

[4] http://www.program-transformation.org/Sdf/SdfBundle

```
module example

entity User {
  name      : String
  password  : String
  homepage  : URL
  homepage2 : URL2
}

entity BlogPosting {
  poster : User
  body   : String
}

entity URL {
  location : String
}
```

Figure 2.4: An example program based on the language defined in Figure 2.3

```
Module(
  "example"
, [ Entity(
      "User"
    , [ Property("name", Type("String"))
      , Property("password", Type("String"))
      , Property("homepage", Type("URL"))
      , Property("homepage2", Type("URL2"))
      ]
    )
  , Entity(
      "BlogPosting"
    , [Property("poster", Type("User")), Property("body", Type("String"
    ))]
    )
  , Entity("URL", [Property("location", Type("String"))])
  ]
)
```

Figure 2.5: The resulting AST for the example program in Figure 2.4

Stratego

The Stratego language is a small domain-specific language for program transformation. Stratego is, next to SDF, the core language in Spoofax. Using rewrite rules and so called strategies an AST can be traversed and altered. Such rewrite rules are based on pattern matching. Using a number of strategies a tree can be traversed in a number of ways. There is for instance the option of using a bottom up or top down strategy which allows the application of a rewrite rule to all nodes in a tree. The third major concept in Stratego is the notion of dynamic rules. A dynamic rule can be defined while traversing the tree. This can for instance be used when in a topdown traversal a scope needs to be defined. Such rewrite rules are globally accessible and can therefore also be used to store information which needs to be accessed from some other part in the traversal steps. Dynamic rules are particularly useful to store declarations of variables in their scope, allowing a language designer to be able to retrieve type information when such a variable is used further on in the program.

In Figure 2.6 we have included an example semantic check for the example Entity language which determines whether a referred type is valid or not.

```
strategies

  analyze = topdown(try(record-entity))

rules

  /*Records the definition of an entity in
     a dynamic rule called 'GetEntity'*/

  record-entity:
    Entity(x, body) -> Entity(x, body)
    with
      rules(
        GetEntity :+ x -> x
      )
rules

  /* Reports an error if a property type is undefined.
      This error is reported on the type name 'type'*/

  constraint-error:
    Property(x, Type(type)) -> (type, $[Type [type] is not defined])
    where
      not(!type => "String");
      not(!type => "Int");
      not(<GetEntity> type) //$ no entity for this type
```

Figure 2.6: Stratego code to perform a semantic check on the Entity language

Spoofax executes strategies through an entry-point rule of the form shown in Figure 2.7. Spoofax also allows for the creation of custom *builders* which are also a hooking point to directly call a Stratego strategy. There could for instance be a builder to call a strategy

which transforms an AST into an equivalent implementation in a different language. If that different language would be bytecode, or an intermediate language, it is effectively a *compiler* for the source language if the resulting AST is pretty-printed into an executable file. Pretty printing is essentially the conversion of an AST back into source code.

```
rules

  editor-analyze:
    (ast, path, project-path) -> (ast, errors, warnings, notes)
    with
      editor-init;
      analyze;
      errors   := <collect-all(constraint-error, conc)> ast;
      warnings := [];
      notes    := []
```

Figure 2.7: Form of the stratego entrypoint for editor feedback in Spoofax

The rule shown in Figure 2.7 matches on a tuple of the form (a,b,c) where a represents the AST which was returned by the JSGLR parsing step. This tuple is rewritten (or returns) a tuple of the form $(ast, errors, warnings, notes)$ where *errors*, *warnings* and *notes* are lists of tuples of the form $(node, string)$. The *node* part is the node in the tree where the error, warning or note originated and the accompanying *string* is the actual message which was defined in the semantic check which should be shown to the programmer.

In our example in Figure 2.4, the type $URL2$ is not defined for property $homepage2$. The semantic check we have shown in Figure 2.6 and Figure 2.7 returns the list of errors shown in Figure 2.8.

```
[("URL2", "Type URL2 is not defined")]
```

Figure 2.8: List of errors returned by the example semantic check

Note that in Figure 2.8 $URL2$ is actually the tree node which, in turn, is actually an ATerm which (hiddenly) contains attached information about where in the source code this node originated. Spoofax extracts this location from the tree nodes and shows the reported error in the editor accordingly.

Moving on

As should be clear, Spoofax is a very powerful workbench to design languages with. The SDF definitions and accompanying Stratego programs provide a general way to implement

advanced language analysis for existing and new languages.

In the following sections of this document we will use Spoofax as a base line to research the possibilities and difficulties of building a semantically aware Web based source code editor.

Chapter 3

Requirements

Since applications built for the Web are constrained to the use of technologies which are available in the browser (JavaScript), implementation of a Desktop IDE quality analysis is not a trivial task. In this section we will define the requirements the back-end components of our source code analysis tooling should meet.

3.1 Background

In order to capture the essence of what would enable a Web based IDE, we will enumerate the minimal functionalities our back end will need on which a fully featured Web IDE could be built.

In Section 2.4.1 we have enumerated editor services typical current IDEs support. All of these editor services can be implemented using rewrite rules as explained in Section 2.4.1. In our proof of concept design we will not actually implement all mentioned flavours of editor services. Instead we will solely focus on the quality checking of source code. By this we intend to be able to return Desktop IDE quality feedback regarding errors, warnings and notes. Since we will be investigating multiple angles of approach we need some way to compare these angles. We add the requirement of being able to benchmark our proof of concept implementations.

3.1.1 Benchmarks

For Web IDE based editor services to be feasible, the rate of feedback should be comparable to a desktop based IDE. Based on our experience with IDEs the time it takes for language-specific feedback to be visible is roughly around 300 milliseconds. This would allow for up to around three full update cycles per second. We have set this goal at 300 milliseconds as a subjective answer to the question whether an editor “feels” smooth while modifying code. In real world situations Desktop editors provide feedback in a non-constant time as the size of the program or complexity of an edit fluctuates. Obviously the complexity of a target language will also have great impact on the amount of time it will take to (re)analyse a program in that language.

Since there are multiple steps our back-end will have to perform, at least parsing and analyzing, it would likely be of help to increase the resolution of our data by obtaining statistics about each back-end step. Such an increase of resolution could help in identifying bottlenecks and provide clues where to introduce optimizations.

3.1.2 Quality

In order to guard the quality of the feedback our back-end would provide to a front-end editor we add the requirement of being able to run tests batch wise. This would allow us to compare the feedback our back-end provides to the results obtained from a proven to be correct set of feedback based on a set of input programs.

3.1.3 Maintenance

Because our work will be based on state of the art technology there is a likelihood that dependencies evolve. In order to make sure our proof of concept implementation keeps working while ongoing maintenance on dependencies is done, it would be useful to automatically rebuild our tools as soon as a dependencies get updated in order to identify issues as early as possible.

Another advantage of building automatically means the potential complicated build steps are programmatically defined. Anyone who wants to extend or use our tools could gain knowledge about the build complexities by investigating the build files. The Nix [14] package manager has support for all of the above. Through functional definitions, packages, their dependencies and build setups can be defined. Such build setups can be automatically built as soon as one of the dependencies gets altered. Another advantage of using Nix is the fact most of the dependencies Spoofax is based on have nix release scripts.

3.1.4 Responsiveness

Since JavaScript was introduced it has mainly been used to provide a way to execute some simple client side scripts in order to handle for instance input validation or visual effects. Because of this, JavaScript code is inherently implemented to run in the same program thread as the presentation layer of web pages. Even though recent leaps in optimization and performance of JavaScript engines, a “heavy” JavaScript function will still freeze the user interface of a web browser until the function finishes.

In order to provide a way around this, W3C has proposed WebWorkers in HTML5 [1][42]. WebWorkers basically introduce threads to JavaScript and are particularly useful when needing to execute longer running code on the background without having the execution time interfering with the user interface.

Because our proof of concept Web Editor should not freeze during the times an analysis runs, interrupting the programmer, we add the requirement of a non blocking user interface.

[1] `http://www.w3.org/TR/2012/CR-workers-20120501/`

3.1.5 Generation

As added requirement in order to answer the "*Is generating a semantically aware Web based source code editor feasible?*" research question, we propose a tool which can take any typical Spoofax project and generate an equivalent back-end which can run in the browser. This would also introduce the possibility for future integration of a Web editor builder in the Spoofax language workbench itself and provides a solid abstraction on which to base the experiments for this thesis.

3.1.6 Use cases

In this section we will list some languages for which we want to provide a Web editor. Since we will base our work on the input of a Spoofax project, in principle any typical Spoofax project should be convertible to a Web-based source code editor using the products of this thesis.

Entity Language

When a new project is created with Spoofax (Section 2.4.1), a minimalistic example language template is added called the Entity language. This language implementation includes a simple but "door opening" definition for some simple errors such as a check on duplicate names. It would be a good starting point to start with this simple language and introduce more sophisticated languages from there.

Tiger Language

Since there already exist some Spoofax projects which implement editor services for languages it would be useful to extend our use cases with some of these real world scenarios. The second use case next to the entity language we propose is the more complicated Tiger language [3]. The Spoofax based editor for the Tiger language features semantic analysis for typed variables, functions, etc.[2] Even though a powerful language, its specifications[3] are concise and therefore make Tiger a very suitable language to be used to illustrate the various components a language consists of. In fact, the Tiger language is currently used in the Compiler Construction course at Delft University to help explain how modern compilers can be implemented.

Mobl Language

The last use case language we will explicitly create a Web editor for is the Mobl language. [29, 28, 30, 27] [4] The Mobl language is a statically typed domain specific language (DSL) which can be used to implement web applications. These Web applications are aimed at mobile devices, optimizing the resulting UI implementation for low resolution displays. Mobl

[2] `http://strategoxt.org/Tiger/TigerLanguage`
[3] `http://www.lrde.epita.fr/~akim/ccmp/tiger.html`
[4] `http://www.mobl-lang.org/`

also exposes most recent HTML5 features such as offline cached applications, interaction with GPS hardware and gyroscope libraries. Mobl is closely related to WebDSL [62]. We have chosen to use Mobl instead of WebDSL because WebDSL makes use of much custom, non-Spoofax native, functionality in the semantic aspect of the analysis implementation.

The Entity, Tiger and Mobl language all have an open-source implementation in Spoofax.

3.2 Deliverables

Based on the previously discussed requirements we propose the following deliverables our research should produce:

- Being able to benchmark the various components
- Being able to run quality tests
- Provide Desktop IDE Editor comparable feedback
  - Syntactic
  - Semantic
  - Location of the point in the program where the feedback originated from
  - Acceptable (300 ms) delay between an edit and feedback
  - Non-blocking user interface while an analysis runs

We will present the following deliverables:

- A Web IDE back-end generator
- Proof of concept implementations
  - A Web editor back-end for Mobl
  - A Web editor back-end for Tiger
  - A Web editor back-end for the Entity language
- Benchmark results
- Automatic builds (Nix [14])

Chapter 4

Client based editor services

With "*fully client side* based editor services" we mean having the complete back-end which calculates the editor feedback to be executed locally in the browser. No connectivity to a web server is required and no server side load or bandwidth is used. In this chapter we will discuss, implement and compare two approaches in obtaining a fully browser-based back-end for a semantically aware source code editor. The quality of the back-ends we will present will meet the requirements discussed in Chapter 3.

Because we are mainly interested in the feasibility of automatically deriving a semantically aware source code editor based on JavaScript, the browser programming language, we will not actually implement a front-end. Instead, we will mainly focus on comparing the various aspects of our implementations such as size, performance and memory usage. In Chapter 6 we will present a front-end capable of actually using and displaying the output of the back-ends we will discuss in this chapter.

4.1 Architecture

NodeJS [55] is a platform which can execute JavaScript programs from the command line. NodeJS is based on V8 [1] which is used as JavaScript engine in the Google Chrome browser and is one of the most optimized JavaScript engines currently available.

Because NodeJS and Chrome both use V8 as JavaScript engine, the performance between the two should be very similar if not equal when tested on the same machine. In order to confirm this similarity we have devised an experiment where we run the same JavaScript program in both NodeJS and the Google Chrome browser. We have implemented a deliberately inefficient iterative prime number enumerator and a recursive Fibonacci sequence determination function for JavaScript in (Figure 4.1) and timed the execution times on NodeJS, Chrome and FireFox.

All benchmarks were performed on a *AMD quad core 3.2GHz 8 GB Ram* computer running Windows 7. The version of NodeJS was *0.7.2* and the version of Chrome was *18.0.1025*.

[1] `http://code.google.com/p/v8/`

Each test was executed five times straight, after which we took the average run time. In Figure 4.4 and Figure 4.3 we present the benchmark results of executing these algorithms on both NodeJS and Chrome.

```
function fib($n)
{
        if ($n < 2)
                return $n;
        return fib($n-1)+fib($n-2);
}
```

Figure 4.1: Recursive Fibonacci function in JavaScript for performance comparison

```
function PrimeCount($max)
{
        var $c = 0;
        for ($i = 2; $i < $max; $i++)
        {
                if (isPrime($i))
                        $c++;
        }
        return $c;
}

function isPrime($num)
{
        $isPrime = true;
        for ($i = 2; $i < $num; $i++)
        {
                if (Math.round($num / $i) === ($num / $i))
                        $isPrime = false;
        }
        return $isPrime;
}
```

Figure 4.2: Inefficient iterative JavaScript based function to count the amount of prime numbers smaller than $max

Clearly NodeJS and Chrome show a very similar performance. We expected the V8 implementation running in Chrome to always have a slightly worse performance due to overhead caused by the browser itself. This was however only the case in our heavily recursive test whereas the iterative test was practically equal. As is also obvious from these numbers, FireFox performs orders of magnitude worse for both our tests.

Since the in this document described work revolves around a proof of concept we have deemed it acceptable to mainly target Chrome. Also, a large number of the benchmarks we have performed and will present in the next chapters were done on NodeJS for convenience

Platform	40	38	34	30
NodeJS	1931.4	738.2	108.0	16.4
Chrome	2128.0	814.6	119.4	17.2
FireFox	crashes	crashes	5366.8	785.4

Figure 4.3: Performance in milliseconds of running the recursive JavaScript $fib(x)$ test from Figure 4.1 on multiple platforms.

Platform	20K	15K	10K	5K
NodeJS	3333.2	1865.0	819.4	205.4
Chrome	3316.4	1868.2	830.6	210.4
FireFox	8311.2	4534.6	2019.0	523.4

Figure 4.4: Performance in milliseconds of running the inefficient iterative $PrimeCount(\$max)$ JavaScript from Figure 4.2 on multiple platforms.

of being able to create scripts for a command line tool whereas automating performance tests in a UI based application such as Chrome would be cumbersome. Because of our presented benchmark results between Chrome and NodeJS in Figure 4.4 and Figure 4.3, we are of opinion using NodeJS to make claims about performance we could achieve in the Chrome browser is justified in the following sections.

4.2 GWT

Spoofax produces a Java based back-end for editor services which are created for a target language. The Google Web Toolkit (GWT) contains a transformation tool to convert Java to JavaScript. Since our goal is to investigate the various aspects of Web based *Desktop quality* editor services, GWT is an obvious first attempt at obtaining a proof of concept JavaScript based implementation of the Java based back-end.

4.2.1 Difficulties

Regretfully GWT cannot transform the full set of Java languages into JavaScript. There is for instance no support for file I/O and threading. Even though the Spoofax generated back-end currently does not actually rely on threading, thread safe data types were used to keep threading an option in the future. The back-end Spoofax produces for the Eclipse editor was likely designed without the requirement of it ever being portable to JavaScript in mind. Therefore when attempting to use GWT to directly port a Java based back-end to JavaScript a lot of problems are reported by the transformation tool.

We have spent a fair amount of time manually fixing these compilation problems in the generated Java code by for instance implementing dummy I/O classes. These implemen-

tations were made using JSNI (JavaScript Native Interface). Using JSNI it is possible to write native JavaScript code inside Java methods. These native JavaScript methods can be called from Java to JavaScript converted classes. There is also the possibility to have the GWT compiler *replace* Java classes with a manually implemented JavaScript version. In Figure 4.5 is an example of JSNI. We have in addition to our own fixes used the totsp-emu library[2] which is a set of JSNI Java classes which enabled us to add a great amount of missing Java functionality in JavaScript and "repair" GWT build errors.

```
public static native int alert(String msg) /*-{
   $wnd.alert(msg);
 }-*/;

 public void test () {
       alert(''Native JS alert popup message'');
 }
```

Figure 4.5: A JSNI native JavaScript method body called from Java

4.2.2 Limitations

Because JavaScript has a browser based nature some features in typical current programming languages such as persistent storage access can not be taken for granted. As previously mentioned, GWT does not support converting Java to JavaScript if it makes use of file I/O classes. This poses a problem in particular for our IDE back end because any arbitrary programming language is very likely to support the inclusion of external files. Take for instance the possibility to reference public classes in Java or the inclusion of library functions which are defined in another set of files.

These limitations are however in theory solvable by wrapping all Java I/O calls, and forwarding them to some sort of client/server JavaScript API where a required file can be downloaded on request through HTTP or by embedding all possible required external file contents in the analysis engine itself. However since we want to restrict ourselves to a *fully* (and potentially offline) browser based analysis of a program we have decided not to include external file support in our initial proof of concept design.

4.3 Measurements

Measuring and capturing performance statistics on Desktop applications is usually a rather straightforward task. There is however a complication when attempting to gather uniform performance statistics on client and/or server side Web applications. Since we want to keep the possibility open to measure performance between both command-line run JavaScript

[2] http://code.google.com/p/totsp-emu/

applications and JavaScript programs which run in the browser we have decided to take a logging-server based approach. The main advantage is that given the Web based nature of our research, (HTTP) requests can easily be made. This in great contrast with easy access to persistent storage. Also having a single entity gather the data in a uniform manner allows us to interpret the information more efficiently than having to manually gather measurement results from various locations.

We have implemented a data gathering service which listens for incoming results using a simple custom TCP protocol which allows for a (remotely) measuring entity to post statistics. In order for the data to be meaningful we have added the possibility of clients to also provide a metric concerning the amount of work and which task they were performing. Obviously in our use of this server we have taken steps to assure the posting of the results, or the measuring itself, does not influence what is being measured. This method of measurement was used throughout the graphs and tables we provide in this document.

4.3.1 Metrics

The obvious first metric one thinks of when trying to compare programs together is to count the lines of code (LOC). Even though we shall mostly use LOC as metric we have attempted to define an *easy to obtain* more in-depth metric which should provide insight into the complexity of a program. Because we will be working with programs in multiple programming languages, this metric should be applicable to any arbitrary programming language.

The metric we will now present has been named “MDS”, short for Max Depths Summed. Since we will base most of our measurements on SGLR parsable languages we can easily obtain an AST for any given program in a language for which an SGLR parse table exists. MDS is calculated using the Stratego program in Figure 4.6. In essence, a leaf node has the weight 1, and each parent gains a point, where it takes the maximum value of their children if it has multiple. The result is a list of numbers which is finally summed together forming a weight for the source program.

```
max-depths-summed = <sum><max-depths>
max-depths = collect-all(max-depth, conc)

max-depth = max-depth(|1)
max-depth(|d):
  ast -> max-depth
  with
    params := <?_#(<id>)> ast;
    max-depth := <list-max> [d|<map(max-depth(|<inc> d))> params]
```

Figure 4.6: The Stratego definition of the MDS metric

This metric is especially useful when comparing sets of programs which are written with very different programmer signatures, where for instance one programmer tends to make one-liners a lot and another prefers to add comments before each statement. MDS provides

an easy to use method to obtain a comparable complexity-based weight to programs written in any language. In this document we will use either MDS or LOC as horizontal metric in our graphs depending on the uniformness of the input-program set.

4.4 Syntax

Because a native JavaScript implementation for the SGLR parsing algorithm does not exist, we have prioritized getting JSGLR to run reliably on JavaScript. It would have also been possible to fully implement JSGLR in pure JavaScript but since the Java implementation consists of around 11,000 lines of code and relies on a number of extra libraries such as the Stratego Terms library it would have taken too much time to implement SGLR in pure JavaScript in the time frame of this thesis.

4.4.1 Previous work

A major step in previously conducted work has been the creation of the JSSGLR project in [49]. GWT was used to port a wrapped version of the java based JSGLR parser implementation to JavaScript. Most of the work done revolved around optimizing the loading times for a JSSGLR parser. The time to initialize the JSGLR parser is heavily dependent on the parse table it is using.

In [49] an approach to transform a parse table from a file/string representation into JavaScript functions is described. The transformation of a regular parse table into this "JavaScript Functions" based representation of the input parse table is implemented as a Stratego program. The resulting representation of the parse table is smaller than its original because of a number of optimizations such as the sharing of recurring terms and optimally renaming of internal tokens. Using this approach they have improved the initialization times for JSSGLR.

The final relevant contribution in this previously conducted work has been the introduction of a wrapping JavaScript WebWorker, also mentioned in Section 3.1.4. A JavaScript WebWorker is in essence a thread which can execute a long-running task in the background without slowing down the UI.

In order for this previously done work to be viable in our research we have decided to rewrite much of JSSGLR so we could make use of the newest version of the JSGLR parser so we could benefit from state of the art work such as [13, 36]. Also in order to be able to conduct performance tests in a straightforward manner we wanted JSSGLR to produce a stand alone build rather than the product presented in the previous work which is highly aimed at, and depends on, its target architecture. Finally, we required the parser to be able to produce custom AST nodes which would be more JavaScript friendly, discussed in Section 4.6.

In the previously conducted work in [49], most measurements and effort went into speeding up the loading times of JSSGLR. Because we are mainly interested in the run time performance of JSSGLR we have conducted a number of experiments involving randomly picked and generated source files for different languages. In the following section we will present the relevant statistics regarding the performance JSSGLR offers.

4.4.2 Comparison

In order to be able to compare JSSGLR to JSGLR we have added identical measurements regarding performance of speed and memory usage inside both implementations.

The Java Virtual Machine and a large number of CPUs become slightly faster when they have been busy with a workload directly prior to measuring performance. This difference in performance is due to in-memory optimizations the Java Virtual machine makes, and possible power saving functionalities which put an idle CPU at a lower clock frequency. Therefore, all the measurements we have performed are preceded by a preheating of running the parser for ten iterations.

The results we present in the following sections were obtained by running all benchmarks on the same Windows 7 *AMD quad core 3.2GHz 8 GB Ram* machine, Java version *Java(TM) SE Runtime Environment (build 1.7.0 04-b22)* and NodeJS version *0.7.2*.

Size

The size of the to JavaScript transformed implementation of the SGLR parser was (at max GWT optimization and obfuscation setting) 162*KB*. The combined size of the Java implementation jars is 265*KB*. We will present performance graphs for both the Java and Mobl language. The parse table for Java is 363*KB* and the parse table for Mobl is 624*KB*.

Speed

We will present a comparison between the native Java JSGLR and the JavaScript based JSSGLR implementation. We have used the Java syntax definition and a random selection of 50 Java source files from the open source TomCat [3] project. We ran the tests three times per input program and took the average numbers in our data set to base our analysis on.

[3] `http://tomcat.apache.org/`

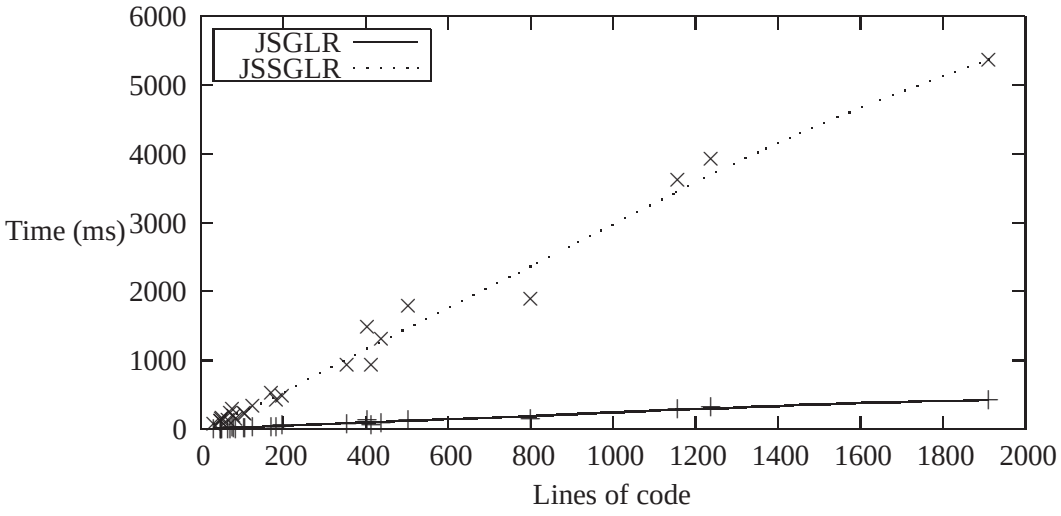

Figure 4.7: Performance of executing the Java based JSGLR versus the JavaScript based JSSGLR parser on a *random* set of Java source code files

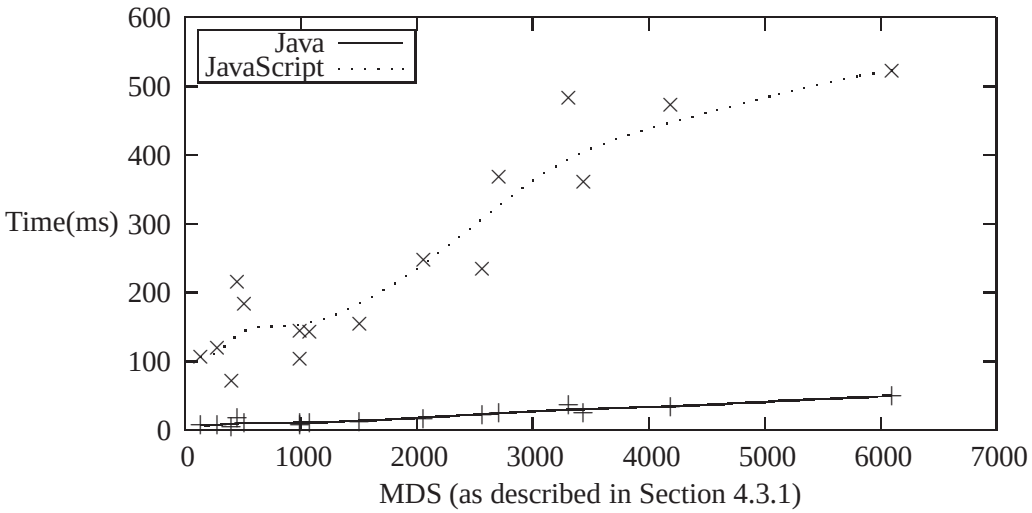

Figure 4.8: Performance of executing the Java based JSGLR versus the JavaScript based JSSGLR parser on a *random* set of Java source code files (Removed results for source files with more than 200 LOC)

As can be seen in Figure 4.7 the native Java implementation is clearly a lot faster than the JavaScript version. This graph however does show a linear growth in direct relation to

the amount of lines of code indicating JSSGLR scales linearly, as JSGLR does. In Figure 4.8 we have graphed the same data set, without the (> 200) lines of code containing Java source files. In this graph the curve is less straight which is most likely caused by fluctuating complexity per source file. We have plotted this graph using the MDS, discussed in Section 4.3.1, metric because simply looking at lines of code does not take the syntactical complexity, especially deep nesting, into account. The changes in syntactical complexity are in this case especially significant because this is a random test set in which also Java interfaces with large comment blocks reside. We can conclude the JavaScript implementation is slightly less stable performance wise compared to the Java implementation but does follow the general curve which was also visible in the large test set from Figure 4.7. Since our previous comparison in Figure 4.7 was based on a *random* set of source files we will now present a test based on *generated* source files. Each test steadily increases in size which guarantees a reliable comparison. The following graph in Figure 4.9 shows the performance of JSGLR vs JSSGLR when parsing a generated set of programs created in the Mobl [29, 28, 30] language and parsed using the Mobl SGLR syntax parse table.

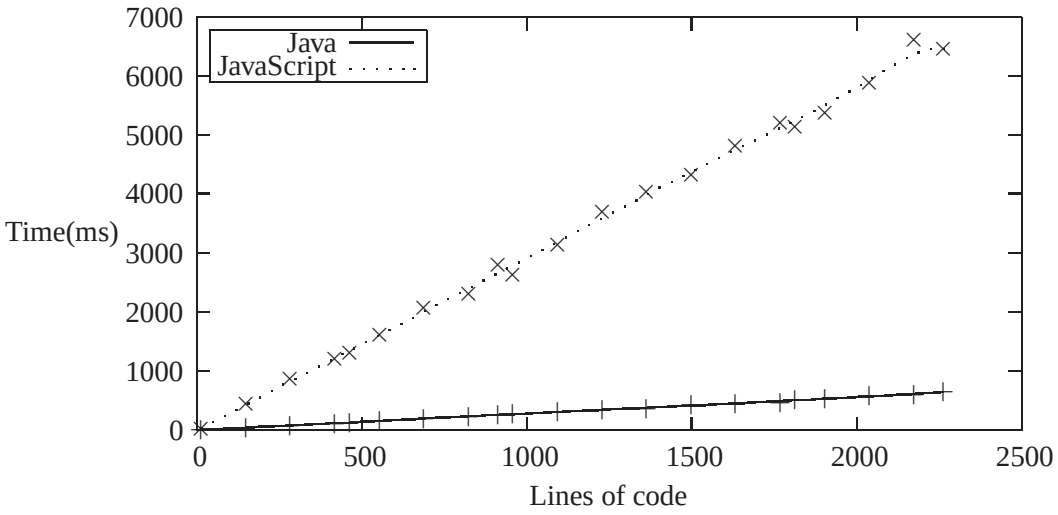

Figure 4.9: Performance of executing the Java based JSGLR versus the JavaScript based JSSGLR parser on a *generated* set of mobl source files

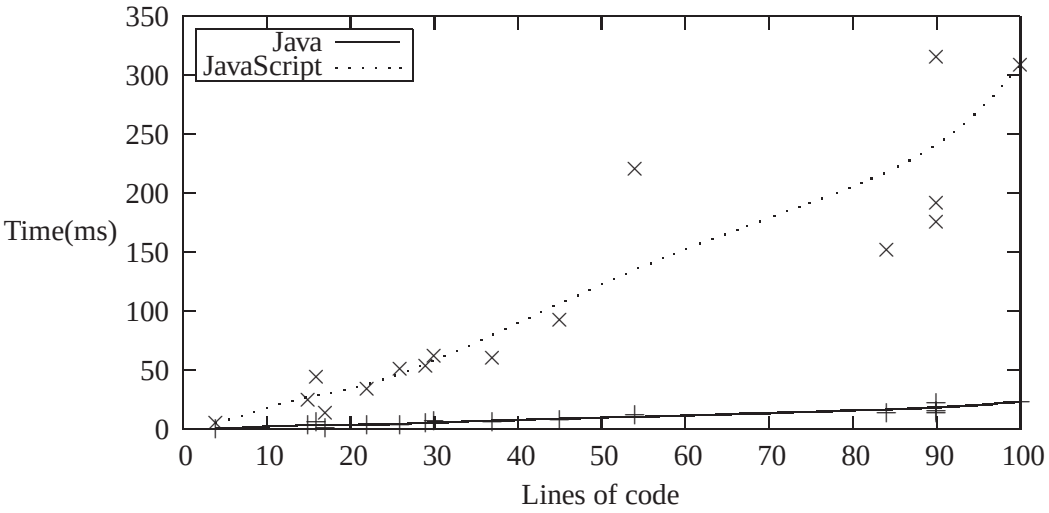

Figure 4.10: Performance of executing the Java based JSGLR versus the JavaScript based JSSGLR parser on a *random* set of mobl source files

Figure 4.9 shows a similar straight linear curve as the graph in Figure 4.7 which was to be expected because we used a generated set of programs. In our final comparison of the Java vs the generated JavaScript based implementation of SGLR we present the performance of a set of *random* Mobl source files taken from the Mobl back-end library and example files in Figure 4.10. Interpreting these graphs we can conclude that the JavaScript based implementation of the SGLR parser performs in a stable manner and can be scaled up to parse very large input files.

Memory

Finally we will present our obtained statistics regarding memory usage. The memory usage is of particular interest because this is currently a significant factor on portable devices. Current portable devices have about 256 MB RAM. In Figure 4.11 we present our results for the generated Mobl programs set. As can be seen, the memory required for JSSGLR begins at roughly 30 MB and increases around 1 MB per thirty lines of code. Memory wise, these are promising results when looking at the possibility of running JSSGLR on a portable device.

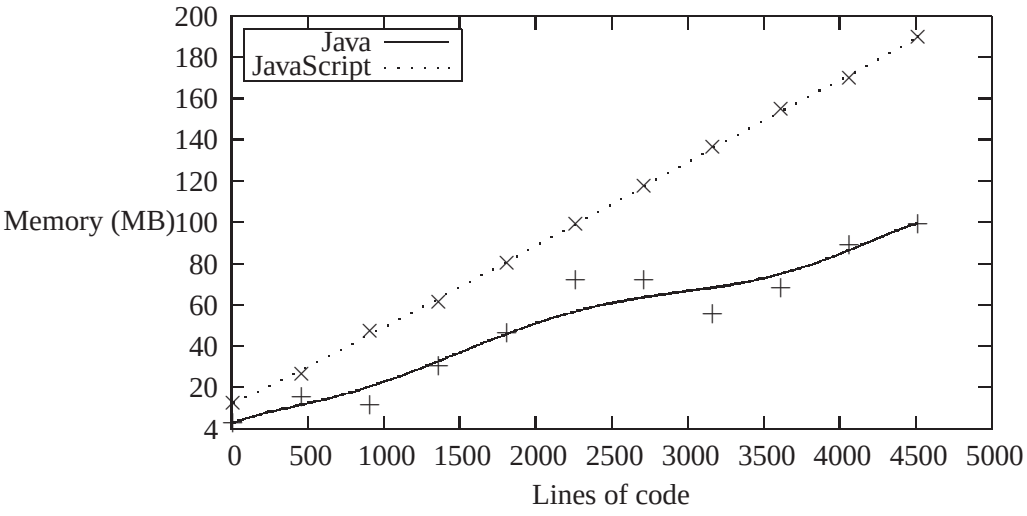

Figure 4.11: The memory usage of the Java based JSGLR and the JavaScript based JSSGLR parser, parsing a set of *generated* Mobl source files

Discussion

The resulting size of the JavaScript implementation and the required parse table are quite large. These sizes are however not prohibitive for usage on the Web. The parser implementation and parse tables are also static and can therefore easily be cached on the client after a single transfer.

Comparing the curves of both SGLR and JSSGLR, it appears that JSGLR runs eight times as fast as JSSGLR. This very noticeable difference in speed between JSGLR and JSSGLR in the previous graphs was to be expected for multiple reasons. For one, JSGLR is optimized for Java in particular. Also, even though there is no fundamental reason for JavaScript to run slower than Java [44], in reality the state of the art JavaScript engines run much slower than current Java Virtual Machines (JVMs). This difference is mainly because of the historical difference of priority in which both technologies were developed. There simply has been a lot more research and effort into speeding up Java compared to JavaScript. Given that the performance gap steadily decreases as JavaScript gains more public interest in the ongoing movement toward Web applications, JSSGLR appears to have a future.

Looking at the performance of JSSGLR, and knowing this is the performance we can get when actually executing this JavaScript program in an actual browser, we qualify the results as actually promising. For Mobl, every ten lines of code account for roughly 15ms increase in parsing time. Also, the average number of lines of code for our Mobl test set lies around 70 lines of code which accounts for an average of 150ms parse time. These

results indicate that producing a syntactical feedback in a web editor is possible in around 150 milliseconds, which would "feel" quite responsive to an end user.

Maintenance

We have created a Nix script, mentioned in Section 3.1.3, which can automatically build the JSSGLR project [4]. This build [5] is executed automatically as soon as the project itself or a dependency is changed. The great advantage is a fully up to date build can always be downloaded. This makes it unnecessary to manually compile JSSGLR which requires checking out a number of dependency projects while JSSGLR is stand alone after it has been compiled. This makes it easy for other projects to depend on JSSGLR and also provides future maintainers of the software a thorough insight into the requirements to compile this tool.

4.5 Stratego to Java to JavaScript

In this section we will provide the results of our experiment to automatically convert a generated Java based semantic analysis to JavaScript using GWT (Section 4.2). The Spoofax language workbench compiles an in Stratego defined language analysis into a Java based implementation. This Java based implementation is normally run natively (in the JVM) on the Desktop pc, providing the back-end for an Eclipse [21] language plug-in.

Our first approach to building a Web editor back-end with the same semantic functionality would be to automatically port the Java implementation to JavaScript. Since we require to have no dependency on Eclipse, because Eclipse does not run on JavaScript, we have created a custom Stratego entry point like Figure 2.7 which takes a source code string as parameter and returns a list of tuples with semantic errors. This would allow us to call the derived Stratego program from command line and enable us to compare the "modified Java" and "through GWT to JavaScript converted" based implementations of the original semantic analysis for the Mobl language.

Given the fact that this transformation would be on an already generated Java implementation of the analysis there was no need to implement this experiment neatly so we basically took a trial and error, "duct tape"-facilitated, approach in order to get it to work. First we obtained all required dependency source files such as the Stratego Term library, JSGLR and the Stratego libraries and placed everything together in a single project. Next, through use of the in Section 4.2.2 mentioned totsp-emu library, eliminations of unused library classes and manually implementing necessary JSNI methods of our own we were able to get a first version of a fully JavaScript based syntactic and semantic analysis for the Mobl language[6].

[4] `https://svn.strategoxt.org/repos/StrategoXT/spoofax/trunk/spoofax/org.spoofax.jssglr/`
[5] `http://hydra.nixos.org/job/spoofax/spoofax-jssglr/build`
[6] `https://svn.strategoxt.org/repos/StrategoXT/experimental/mobl-gwt/`

Performance

In order to obtain performance data between the Java and javaScript based implementation of a full semantic analysis we have modified our earlier mentioned Stratego entry-point to perform the various parts of an analysis in measurable steps. We also added ten warm-up runs to make sure the underlying platforms (JVM, NodeJS) and CPU are fully optimized before taking the actual measurements. We have gathered the results of our measurements using the earlier mentioned data collecting server from Section 4.3.

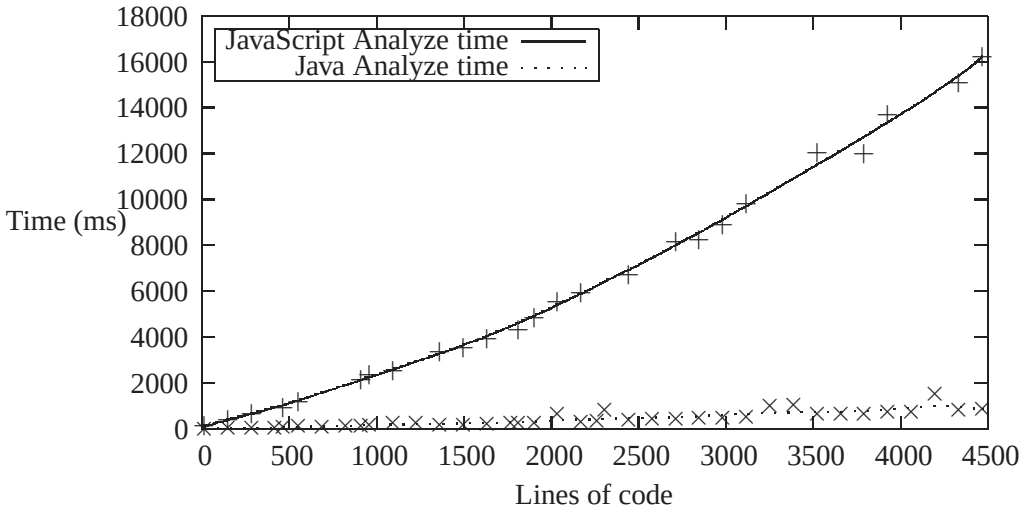

Figure 4.12: Performance of the Strj **(Stratego → Java)** vs the GWTd **(Stratego → Java → JavaScript)** based implementations for the semantic analysis of a *generated* set of programs in the Mobl language

In Figure 4.12 we present a graph which shows the performance of executing the semantic, in Stratego defined, analysis for both the Java and JavaScript implementations. The test base we have used for this graph is a number of *generated* Mobl source files. We have measured the time it takes to obtain semantic errors for code of up to 4500 LOC. Note that the presented graph *does not* include the time it takes to parse the source code, in other words, to obtain the corresponding AST using (J/JS)SGLR, as discussed in Section 4.4. We have presented the performance of obtaining an AST using JSSGLR in Section 4.4.2.

In our generated test set we have introduced a number of trivial semantic flaws which both the Java and JavaScript implementation equivalently reported. As with our results for JSSGLR in Section 4.4, the JavaScript port of the Java implementation runs much slower. Do note however that it is very uncommon to have such large source files, and this graph is mainly used to emphasize the behavior of both implementations when their workloads

increase. In Figure 4.13 we present a similar result set but for more typical real-world scenarios with a range of up to around 250 lines of code. As can be seen, the JavaScript analysis takes roughly 100*ms* for very simple, < 50 LOC, programs and increases by around 25*ms* for every 10 lines of code.

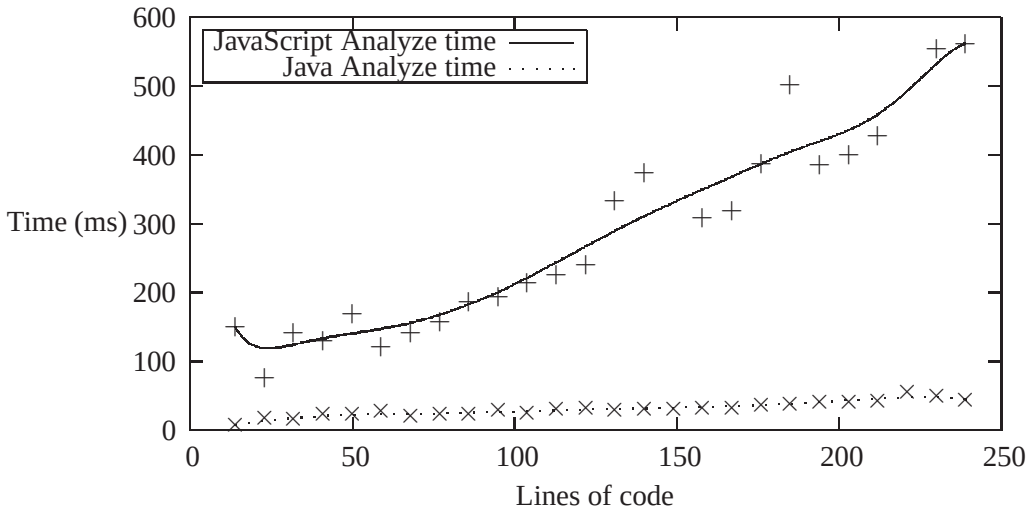

Figure 4.13: Performance of the Strj **(Stratego → Java)** vs the GWTd **(Stratego → Java → JavaScript)** based implementations of a semantic analysis for Mobl source code up to 250 LOC

4.5.1 Limitations

The main limitation in this proof of concept implementation is that it cannot run in the browser in its current form. This is because of a number of, non-performance influencing, simplifications we have made such as minor usage of NodeJS API functions which are not available in the browser. We have for example used the NodeJS file I/O API to open and parse the Mobl SGLR parse table. Even though this limitation exist in this crude attempt, as we will show in Section 4.6, it is possible to embed the parse table in the JavaScript implementation itself.

4.5.2 Discussion

We have spent a substantial amount of time and effort into getting the generated Java code to be transformable to JavaScript using GWT which would indicate this approach to be very unfeasible as it destroys the generative nature of developing a language using Spoofax. However, most of our effort went into getting the non-generated dependencies which the

generated Java code relies on to be transformable by GWT. In fact we have achieved to programmatically alter the language-specific generated Java implementation to be GWT-able through the use of a series of regular expressions on the set of generated Java classes.

Even though we have achieved a decent performance in the Java to JavaScript converted implementation, a major down side of GWT is that most of the conversion process seems like a magic black box. For example we would like to be able to experiment with JavaScript specific optimizations in the compilation step of Stratego code. All current optimizations are Java specific and some parts such as the "Stratego Term" library implement non-required functionality. We are confident that when focusing on a direct Stratego to JavaScript compiler similar, and better, results as the Java to JavaScript implementation should be possible. In the next section we will discuss STR2JS, the Stratego to JavaScript compiler.

4.6 Stratego to JavaScript

The Stratego language currently has two compilers. *Strc* and *strj*. The first compiler generates a C implementation of the Stratego program and the latter generates a Java based version [9, 33, 15]. The *strj* Stratego to Java compiler is embedded in the Spoofax Language Workbench.

Looking at the results we presented in our previous approach in Section 4.5 in which we attempted to port Java to JavaScript using GWT we could not find much obvious optimization points due to the complexity of dealing with generated code. Some optimization angles, proceeding on our previous GWT based approach, are of course possible but we decided a more maintainable approach would be to design a Stratego to JavaScript compiler. We should be able to get at least similar results by simply mimicking the optimizations GWT makes. In addition we would be able to manually implement and specifically optimize the Stratego back-end functions for JavaScript which could provide significant performance increase.

4.6.1 Previous work

In December 2010, work was started at Delft University to implement a Stratego to JavaScript compiler [7], called "s2js". This implementation consists of two parts. On one side the actual compiler which transforms a Stratego program into JavaScript, and on the other a JavaScript implementation of required primitive types and functions. For instance the "Stratego Terms" type system and low-level implementations of strategies that handle core functionality such as dynamic rules and the basic traversal strategies of Terms. This Stratego to JavaScript compiler was created using a small subset of tests from the Stratego to Java and C compilers in order to compare the JavaScript based binaries run time output to the correct output.

In mid 2011 this work was continued [49] in which a number of optimizations and alterations were made. The contributions to s2js in this work particularly involved streamlining

[7] `https://svn.strategoxt.org/repos/StrategoXT/strategoxt-javascript-backend`

the transformation of Stratego to JavaScript. Regretfully, this work was never finished and was not very usable.

4.6.2 S2JS

We have continued the work done on the Stratego to JavaScript compiler. In this section we discuss the challenges we had to overcome in order to obtain an actually usable Stratego to JavaScript compiler for *any* Stratego program which is created in Spoofax. Since the initial set of non-failing tests was rather small compared to the available tests in the Stratego to C and Java compilers we have added virtually most of these tests to the s2js project. With exception of File I/O based tests. Next we implemented an easy way to run all these tests and gather a list of failures using the in Section 4.1 described NodeJS environment.

Obviously, in our initial attempts we came across a lot of problems. We attempted to group the problems into similar Stratego functionality in order to efficiently modify the Stratego JavaScript back end to produce correct output. We have also made our additions to the Stratego JavaScript keeping a close eye on the original Java implementations from the Stratego Terms library. The JavaScript binaries the s2js compiler produces depend on the following key components:

1. SRTS - Stratego RunTime System
   The Stratego run time System is in essence the core dependency for any s2js'd Stratego program. The three low-level implementations of the core traversal strategies in Stratego, namely "one", "some" and "all", are implemented here. For example $all(s)$ is defined to apply the strategy s on all sub terms of the current term. The low-level implementation is platform-specific, so in s2js JavaScript based. The most commonly used traversal strategies, bottomup and topdown, are defined in the Stratego library and directly rely on the $all(s)$ strategy. [61, 9]

   ```
   bottomup(s) = all(bottomup(s)); s
   topdown(s) = s; all(topdown(s))
   ```

2. JS ATerms - JavaScript ATerms (Annotated Terms)
   The JS ATerms component in s2js contains the implementations of the JS Terms. This is essentially a stripped down version of the Java based Stratego Terms. Each term has a certain type such as (*int*, *real*, *string*, *list*, *tuple*, etc). The JS Term system exposes functions to retrieve the type of, compare, construct and modify terms.

3. JS Primitives - JavaScript Primitive functions
   The JS Primitives library consists of all core functionality which can not be (efficiently) defined in Stratego itself. There are for instance primitives to add and subtract integers. In the next subsection we will explain Stratego primitives in more depth.

4.6.3 Primitives

The Stratego language is layered on top of a set of primitive strategies. These strategies call native code which is platform specific and has to be implemented in the underlying language. This is in our case of course JavaScript. In Stratego these primitive strategies are mostly invisible from normal users because they lie at the heart of the various Stratego libraries. The Stratego libraries provide a large set of commonly used functionalities such as multiplication of numbers, concatenation of lists or testing for (in)equalities.

Eventually, most Stratego statements reach one or more of these primitive functions. In Figure 4.14 is an example of an invocation of a primitive function.

```
call-primitive-func : x -> y
  where y := <prim(''Primitive_Function'')>x
```

Figure 4.14: A Stratego call to a primitive function

Note that the Stratego to Java compiler translates this to an invocation of a method in a class which implements a special interface. In s2js this translates to a simple JavaScript function call. In this example *x* is the first parameter for this function call.

4.6.4 Obtaining an AST

At this point we are able to compile Stratego programs to JavaScript. This JavaScript implementation is based on a custom implemented ATerm system. In order to be able to apply our JavaScript based Stratego program to an AST we need to obtain such ASTs in a compatible format. The Stratego language provides access to an SGLR parser through a specific Stratego SGLR library. In the Stratego to Java compiler this library depends on JSGLR. We could not simply use the output AST which JSSGLR provides directly in s2js because JSSGLR provides Java based Stratego Terms. Granted, these are actually JavaScript objects, they are still based on the GWTd Java implementation which is not compatible with the earlier mentioned JavaScript ATerms in Section 4.6.2.

Thankfully the architecture of JSGLR allows for the implementation of a custom TermFactory which provides an abstraction from whatever term system is used. Using JSNI, discussed in Section 4.2.1, we implemented a specific s2js-TermFactory. We made sure the in native JavaScript implemented Term factory is called during the construction of the AST by our improved version of JSSGLR.

We have mentioned a JavaScript aimed optimization for JSSGLR in Section 4.4.1 which transforms an SGLR parse table into a set of JavaScript functions rather than a string which has to be parsed in order to initialize an instance of JSSGLR. This optimization was implemented in Stratego. We have essentially embedded this “table to functions” transformation inside the s2js compiler by replacing the Stratego statement which imports a table file in the

to be to compiled AST representation of the target Stratego program.

At this point we were able to write a Stratego program which accepts a string as parameter, parse this string into an AST by calling a custom Stratego Primitive which calls JSSGLR, and pass the resulting AST to a strategy. This Stratego program can be compiled by s2js and run on NodeJS.

From Spoofax

Because all language-specific aspects which are required for a semantically aware editor is inside a Spoofax project in the form of SDF definitions and a Stratego program we have created a general Stratego program which in essence glues everything together. An existing Stratego program which retains inside a Spoofax project is wrapped by a custom Stratego program which exposes a command line interface. This wrapper transforms a string into an AST and feeds this AST into the original Stratego program which detects and reports the possible semantic errors. In Chapter 6 we present a tool which automates the generation of a Web IDE based on a Spoofax project.

4.6.5 Results

In this section we will present the benchmark results we have obtained from running a set of Stratego programs which were compiled using s2js. We will mirror these results with the previous Java and GWTd Java implementation performance. In Figure 4.15 we have included a graph representing the performance of the s2js implementation of Mobl. In this image the initial comparison is between the native Java and native JavaScript implementation. Even though the directly to JavaScript transformed Stratego program is much slower than the native Java implementation, these results are promising because we now have a solid foundation to implement JavaScript specific optimizations on.

When looking at our second comparison, where we compare the GWTd implementation from Section 4.5 with the implementation we have obtained using the Stratego to JavaScript compiler we see a very similar performance. We suspect this to be a confirmation that JavaScript interpreters are currently simply just less optimized than Java Virtual Machines (JVMs). When looking closely it seems the GWTd implementation is always a fraction quicker than the s2js implementation. We are however confident that future optimizations in the s2js compiler can be made. One could specifically look at the Stratego language and introduce optimizations which are specifically beneficial for JavaScript in order to further improve the s2js compiler and obtain faster running semantic analysis implementations. Both Figure 4.15 and Figure 4.16 are based on the performance of a set of random files from the Mobl library.

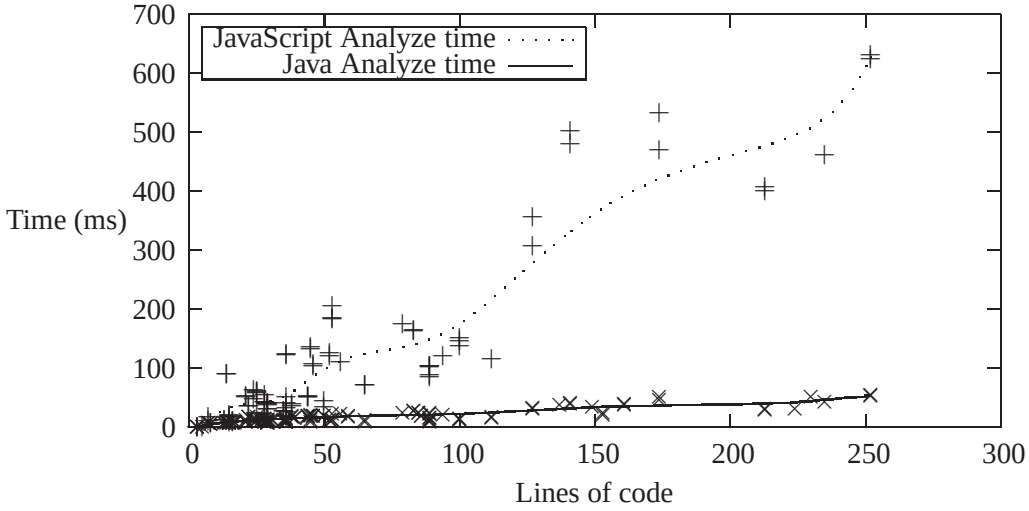

Figure 4.15: Performance of the Strj **(Stratego** $\rightarrow$ **Java)** vs the S2JS **(Stratego** $\rightarrow$ **JavaScript)** implementations of a semantically aware analysis for the Mobl language on a random set of Mobl source code files $<=$ 250 LOC

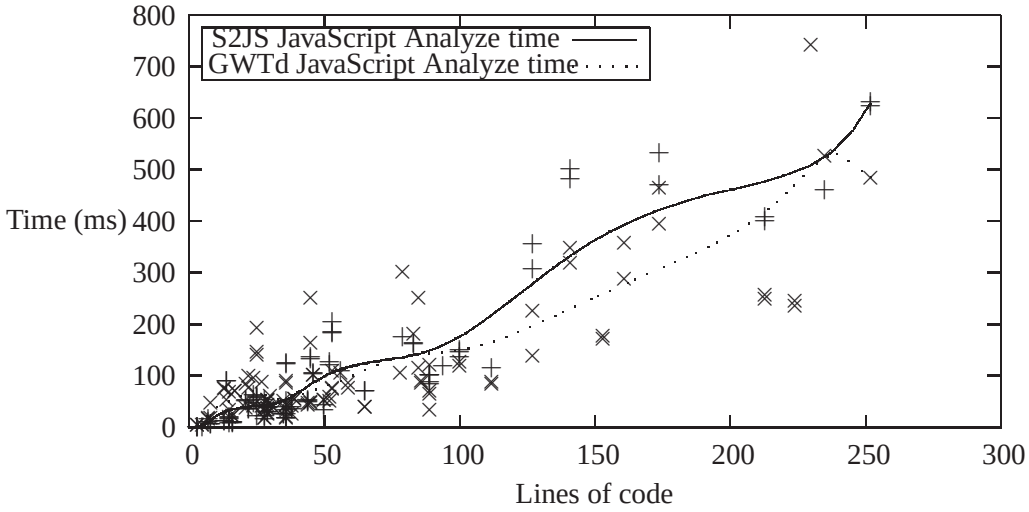

Figure 4.16: Performance of the GWTd **(Stratego** $\rightarrow$ **Java** $\rightarrow$ **JavaScript)** vs the S2JS **(Stratego** $\rightarrow$ **JavaScript)** implementations of a semantically aware analysis for the Mobl language on a random set of Mobl source code files $<=$ 250 LOC

Chapter 5

Server based editor services

Our second major approach to implementing a semantically aware Web based source code editor is to rely on a server back end to perform the actual calculations required for the analysis. We have implemented a proof of concept service which provides semantic feedback to remote clients. This new approach obviously poses a wide range of new problems such as how to handle connectivity problems. In this section we will discuss the solutions we have used to overcome these problems.

5.1 Disadvantages

It is clear that the radically different approach in where the calculations are performed introduce a number of problems and aspects to take into consideration. The obvious first requirement would be that the users who program in our Web editor shall always need to be connected to the internet in order to receive the syntactic and semantic feedback. Also, the availability and latency of an internet connection could greatly influence the performance and the quality of the editor feedback.

5.1.1 Overhead

Whenever a key press occurs which alters the program being edited there could potentially be new editor feedback which is why ideally the analysis should be executed. In comparison to our previous approach when everything is calculated inside the browser there now are a number of extra steps which have to be performed. In the most naive approach in which all data is always sent over the line the following steps are at least required (in order):

1. Client side:
    a) Packaging the program in a form which can be sent over the internet
    b) Posting/uploading this package to the server back end
2. Server side:
    a) Receive the package

b) Decode the package into the program which must be analyzed

c) Running the Stratego program which performs the analysis

d) Package the resulting editor-feedback in a form which can be sent over the internet

e) Returning/uploading this package back to the client

3. Client side:

a) Decode the received editor-feedback package

b) Apply the editor-feedback to the running editor-instance

In contrast, in the method explained in Section 4.6 only steps 2c and 3b have to be performed.

5.1.2 Loss of scalability

The fully client-based method described in Section 4.6 has in essence no limit to the amount of concurrent users. The analysis is run on the browser at the programmer which only puts a small amount of stress in a hosting entity in the form of serving the required files to the users. This also needs to happen only once due to client-side caching.

Obviously, the client-server based approach we propose in this section will have a maximum amount of concurrent users because each user would put stress on the server back-end. Even though with the recent developments in Cloud based architectures in which we can scale seemingly indefinitely, the cost in doing so could still be a limiting factor.

5.2 Advantages

A major advantage is the fact we are no longer bound to the use of JavaScript to perform the actual calculations in. As we have shown in both of our previous proof of concept implementations in Figure 4.12 and Figure 4.15, a native Java implementation *currently* executes an analysis much faster than a JavaScript implementation. Also, because of this freedom in choice of platform, problems regarding File I/O are much easier to cope with because such functionality is natively available in most general purpose programming languages.

5.3 Justification

Even though there are many disadvantages in the client-server based approach, our previous attempts in Section 4.5 and Section 4.6 have clearly shown a native Java approach to currently be much faster. In the proof of concept editor we will present in this section we have made a number of abstractions in the form of a couple of assumptions. These assumptions are:

- A non-failing connectivity between client and server

- A constant latency

Based on the graphs mentioned in Section 5.2 we suspect that for larger analysis jobs a server-based (Java) approach will “win” from the fully browser based implementations even with the added complications of the overhead we have discussed in Section 5.1.1. Obviously the loss of “cheap” scalability as mentioned in Section 5.1.2 can not easily be solved.

5.4 Proof of concept

We have implemented a proof of concept client-server based semantically aware Web editor. The enumerated steps mentioned in Section 5.1.1 have been implemented in JavaScript and Java. We have created a bandwidth-heavy version which basically sends all data back and forth between the client and server omitting the necessity of managing a state at either the client or server side. At the server side we receive the source code which must be analyzed, analyze it and return the semantic feedback back to the client.

5.4.1 Transport layer

The classical approach in developing a web-application which communicates with a server back end is to use requests. These requests originate from the client side. There are a number of protocols a web developer can use to achieve this extra level of communication such as HTTP Get/Post and Ajax [46]. A common workaround to *simulate* server-initiated communication (push) is by implementing long-polling or streaming as is done in Comet[52, 8].

Currently, W3C is in the process of standardizing the WebSockets[19] protocol. The WebSockets protocol exposes the possibility for a full-duplex connection between client and server and, more interestingly, allows *either* side of the connection to initiate the sending of a message. Also, because of the nature of a socket there are always two endpoints which removes complexity in the managing of multiple sessions server side. At the server-side, an open socket could simply be coupled to a session with a certain state whereas the alternative would mean extra checks and overhead. In [25] a comparison is made between the classic approach and WebSockets in which WebSockets clearly perform with less latency and bandwidth overhead.

WebSockets

We have decided to use WebSockets as main transportation layer in our proof of concept. We did come across a number of difficulties which were mainly due to the fact that at the time of this writing the WebSocket API definition was not completely finished by W3C[1] and therefore not uniformly implemented in all browsers. Our server-side implementation

[1] `http://www.w3.org/TR/websockets/`

was implemented using Jetty [2], a lightweight Java based HTTP server which supports WebSockets. We identified a rather prohibitive problem in our first tries in which a WebSocket message which originates from a client could be no longer than 16385, or $2^{14} + 1$, bytes. The opposite way, from server to client there appeared to be no limitation at all indicating it was a Chrome browser-specific limitation. We have tested this by transmitting up to $50MB$ of data to the client. In order to overcome this client to server limitation we have implemented a simple fragmentation protocol on top of the WebSocket API which slices a message into multiple parts of maximally 16385 bytes and transmits the sub messages in order, prefixed with information about the current slice and how many slices exist in total. At the server-side the original message is reconstructed.

5.4.2 Analyzing

We now arrive at the moment where we have the original source code which resides in the editor at the server in a buffer. The next step is of course to actually perform the analysis in order to obtain the editor-feedback. We have implemented our Java based Jetty WebSocket servlet as a project which references the Java implementation of a Stratego program.

We obtain this Java implementation by compiling a wrapped target Stratego program with a custom entry point like we did in Section 4.6 and Section 4.5. We compiled this wrapped Stratego program using the Stratego to Java compiler, *strj*. Next we call the custom entry-point for this compiled Stratego program and pass the source code we obtained from the WebSocket as parameter. The resulting *StrategoTerm* which represents the AST, errors, warnings and notes is then converted into a JSON[12] string representation which can be used by the client side. This JSON string is then transmitted back over the WebSocket.

5.4.3 Receiving an analysis result

Once the client receives the response from the server back-end in the form of a JSON string, the result is is parsed into a JavaScript object. At this stage we have all the syntactic and semantic feedback a Spoofax derived editor would have used at our disposal. In Section 6.1.3 we will present an editor front-end in which we actually visualize these analysis results in the browser.

The resulting feedback can in its current form be more precise due to the availability of referenced files at the server side. In for instance the Mobl case, there is a large library which defines a number of common types such as a “label” or a “table”. In the s2js approach each reference or usage of such a type would result in a semantic error. In this current server-side proof of concept it was however trivial to make the library available to the Stratego program since the Java implementation could make easy use of Java’s file I/O functionalities.

[2] `http://jetty.codehaus.org/jetty/`

Performance

In Figure 5.3 we present the performance of this approach for a *full* cycle between client and server. We have used a subset of the generated set of Mobl programs from Section 4.5. We have manually performed these measurements in a Chrome based editor. We hosted the server back-end on our main test machine and initiated the editor on a physically distant computer. The (internet) latency between these machines was at all times we performed the tests between 25 and 30ms. Regarding bandwidth throughput between client and server, we have obtained a stable 400kB/s to and from both ends. In Figure 5.2 and Figure 5.1 we present the amount of incoming and outgoing bytes in these tests.

Since in practice an analysis will run after each key press (or after a sequence of key presses) the amount of bandwidth required starts to become a major bottleneck. In an average 100 lines of code Mobl program, each edit of the program would initiate the "upload" of 2KB and the "download" of 80KB. This difference in size is because the server sends tokens instead of only the source code itself. These tokens contain a significant amount of annotated information such as the type of the token, its contents, its origin and a possible error marker. Note that this implementation is not optimized at all and consists of raw, unzipped, JSON data.

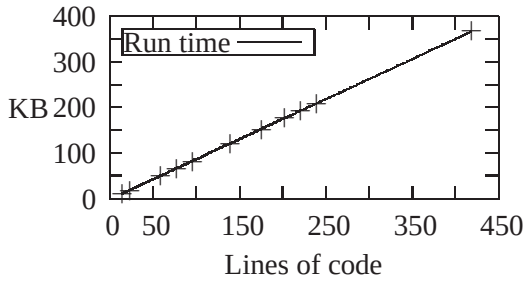

Figure 5.1: The amount of incoming bytes for our client-server based analysis for the Mobl language

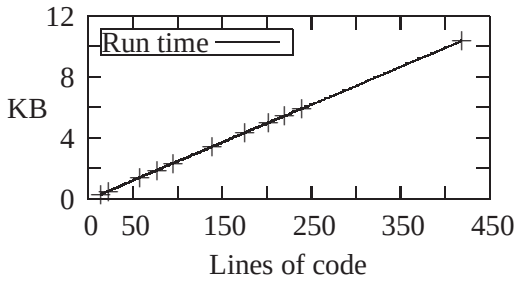

Figure 5.2: The amount of outgoing bytes for our client-server based analysis for the Mobl language

5.4.4 Reducing overhead

Our previous attempt has revealed an obvious point of optimization because of the high usage of bandwidth. With exception of the optimizations described in Section 6.1.2 regarding insertion of new lines, practically each keystroke could trigger a full update cycle. We have implemented bandwidth-saving functionality in our earlier proof of concept. Since most of the source code which is being edited does not change, and any typical alteration of the source can be viewed as a patch to the previous "version", we have decided to incorporate a

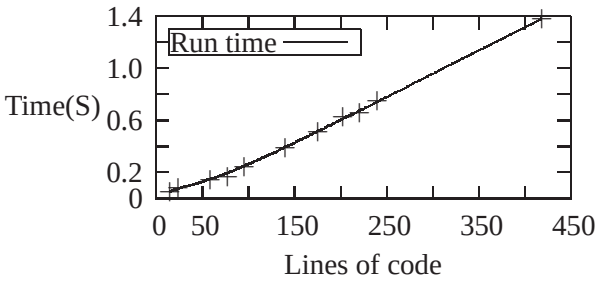

Figure 5.3: Total time working for a full cycle between client-server (210 KM physically apart, with an average latency of 28 ms and a 400 KB/s up and downstream) for our client-server based Mobl analysis

diff/match/patch algorithm [45]. What such an algorithm basically does is take two versions of a string, for instance a string *A* and *B*, compares them, and outputs a patch to obtain *B* from *A*. We have used the JavaScript and Java implementation of Google's diff/match/patch algorithm[3].

In order to only require to send such a patch, the client has to maintain two versions of the source code. The first version is the *last sent* version, which effectively must be the version currently known at the server-side. Of course the second version is the one currently being edited. As soon as a patch is derived by "diffing" the old and current versions, the patch is sent to the server and the *last sent* version is updated.

At the server side, the patch is applied to the currently known version of the source code after which the normal analysis is executed. The resulting JSON tokens are then "diffed" in the same manner after which a patch of that JSON string is replied back to the client. The client applies the patch to its previously most-recent server reply and updates the editor using the new information.

Extra calculations

In essence, our bandwidth overhead reducing algorithm actually introduces extra calculations on both the client and server side. After all, the *diff/match/patch* algorithm is being run a number of extra times. It is in fact possible that adding this layer actually slows down the overall cycle from a modification in the editor to the actual display of updated feedback. We have measured the added overhead in *ms* to actually perform the *diff/match/patch* algorithm. Because the first execution of the algorithm, in which an empty string is compared to whatever needs to be sent over, will produce a patch consisting of exactly the entire second string we have taken this very first execution separate. Following this first execution we ran five successive executions for each test case and took the average extra time it takes to

[3] `http://code.google.com/p/google-diff-match-patch/`

execute the *diff/match/patch* algorithm on the client and server side.

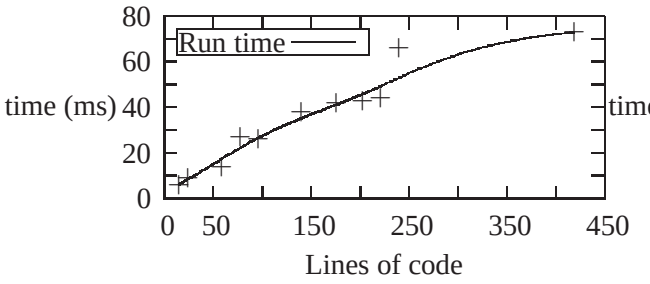

Figure 5.4: Overhead of executing diff/-match/patch (first run) for our client-server based analysis for Mobl

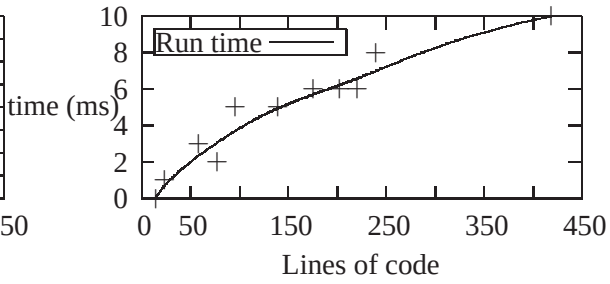

Figure 5.5: Overhead of executing diff/-match/patch (successive runs) for our client-server based analysis for Mobl

In Figure 5.4 and Figure 5.5 we present our results. We can conclude that for a first execution the algorithm is introducing most overhead. For successive runs however, the amount of overhead is negligible compared to the full-cycle execution times we have presented in Figure 5.3. At this point we are of course interested in the actual amount of data which would be transmitted over the network.

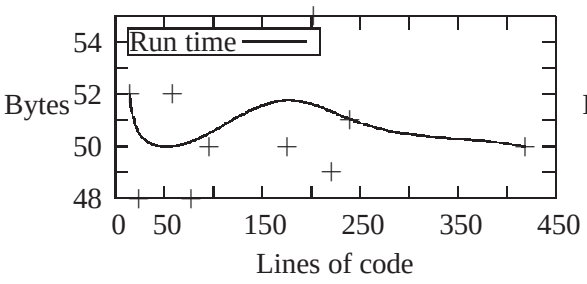

Figure 5.6: The amount of outgoing bytes in the, with diff/match/patch, extended version of our client-server based analysis for Mobl

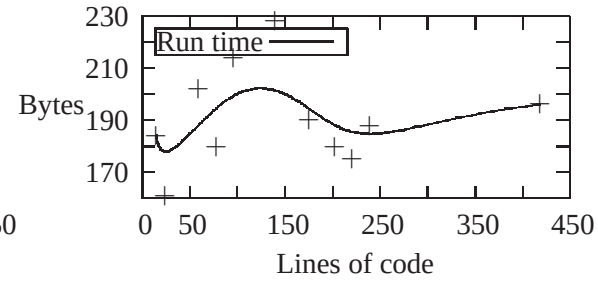

Figure 5.7: The amount of incoming bytes in the, with diff/match/patch, extended version of our client-server based analysis for Mobl

In Figure 5.6 we present the graph regarding the amounts of outgoing data when making small modifications to the program in the editor. In Figure 5.7 we show the size of the resulting patch message which is being received from the server. Depending on the type

of modification that was made, the size of the patch message fluctuates. When for instance pasting in a number of lines, the patch message obviously contains the information from these lines. In these measurements we have simply added arbitrary comments to the code. These are the average sizes of the payloads sent over the network for five edits per test case. Note that regardless of the size of the program which is being analyzed only the type of modification to the program has effect on the size of the patch message. In essence, for comparable acts of modification to the program in the editor, the patch message size is *constant*.

Finally, in Figure 5.8 we present the combined measurements from Figure 5.3 and our with the *diff/match/patch* algorithm extended implementation. Clearly much of the overhead was caused by the large amount of data which was being sent over the internet. Looking at the amount of data which needed to be sent, with a maximum of $368KB$ in the original attempt, one might wonder why this introduces such a significant delay. After all we were able to get a stable $400KB/s$ between both the client and the server in both ways. In Figure 5.3 the maximum amount of time to complete an analysis lies at $1417ms$. Looking at the performance we have obtained in Figure 5.8 there is roughly $400ms$ "too much" time in that original approach. This seemingly extra overhead can however be explained because when initiating a (large) transfer over the internet, there is a small "warm-up" time before the maximum throughput is reached.

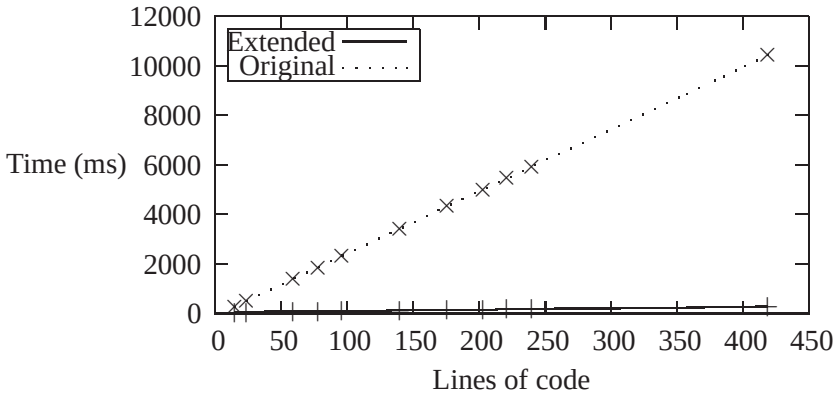

Figure 5.8: Our original vs. the, with diff/match/patch, extended implementation of our client-server based analysis for Mobl

The performance we achieved in our final version of the client-server based proof of concept is close to the native-java performance. After all, the amount of data which is actually sent over the internet is mostly constant because of the diff/match/patch approach and the actual calculations are performed in a virtually equal environment.

5.5 Discussion

We have set out to provide an alternative to the s2js implementation. In Figure 5.9 and Figure 5.10 we present a comparison between our client-server approach versus the performance we achieved in our s2js proof of concept.

Lines Of Code	Time s2js (ms)	Time clnt/srvr (ms)
14	45	53
23	79	56
32	110	79
77	318	80
95	396	94
140	567	114
176	740	148
203	870	161
221	1024	173
239	1570	185

Figure 5.9: Comparison table of performance between s2js and our client/server approach for a Mobl analysis *running in the browser*

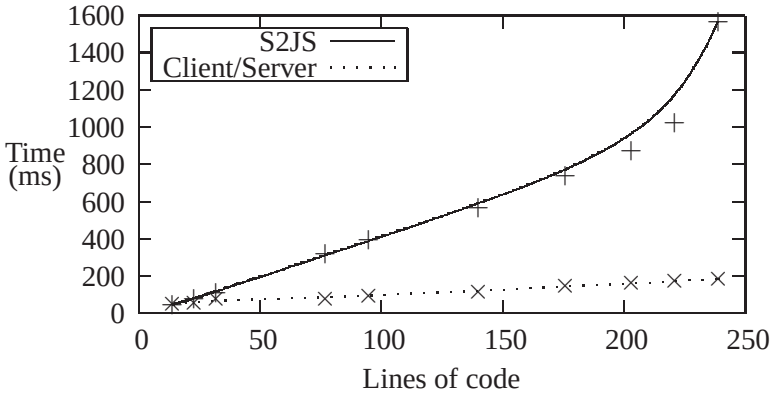

Figure 5.10: Comparison graph of performance between s2js and our client/server approach for a Mobl analysis *running in the browser*

Recall that s2js runs *fully* in the browser, and the measurements from Section 5.4.4 are based on a client-server based setup where there was a latency between 20*ms* and 30*ms* from client to server. The measurements were all performed on the same set of *generated* Mobl source files. Clearly, given these conditions, the client-server based approach wins. These conditions are however optimal. When there would be significantly more latency it might be more beneficial to switch to the locally executed s2js approach. In Section 6.5 we will present a balancing algorithm to automatically decide which analysis strategy to perform.

Chapter 6

Generation

Because using S2JS in combination with JSSGLR, creating a custom Stratego entry point, and incorporating it into a front-end requires quite some juggling with files, we have created “spoofax2ace”. Spoofax2ace is a command line tool which takes an arbitrary Spoofax project and converts it into a full blown web editor.

6.1 Front-end

The earlier discussed JavaScript back-ends are in their current form not very useful. In order to actually obtain a code editor we also need a front end which communicates with the generated back-end. Obviously this front-end has to run in the browser and should be customizable to allow the implementation of the GUI aspects our Web Editor shall require. It must for instance be possible to attach an error marker to text in the editor, along with some explanation as to the nature of the error.

6.1.1 Ace

The open-source Ace (Ajax.org Cloud9 Editor) editor, mentioned in Section 1.2, is fully written in JavaScript. It is the successor of the well known web-editor called Mozilla Skywriter[1] which is, in turn, the successor of the Bespin editor[2]. Ace is built up using a number of modules. There are for example modules to add a style-theme, GUI elements and *custom programming languages.* The current implementations for such custom languages however have little support for syntax, let alone semantic checks. In fact most programming languages the Ace editor and web-based editors such as the amy editor [3], ecoder [4], and codemirror [5] support are implemented using a series of regular expressions.

[1] https://mozillalabs.com/en-US/skywriter/
[2] https://mozillalabs.com/en-US/projects/bespin/
[3] http://www.amyeditor.com/
[4] http://ecoder.quintalinda.com/
[5] http://codemirror.net/

The quality and amount of feedback a programmer actually gets when programming in such an editor is very limited. There is for instance only keyword highlighting and ideally some basic forms of syntax checks such as making sure an opening bracket is eventually succeeded by a matching closing bracket.

Ace does have full syntactic and semantic support for the JavaScript programming language thanks to the inclusion of an external project, called Narcissus[6], which provides a javaScript based parser and semantic check for JavaScript. Obviously such a project does not exist for any arbitrary programming language which likely makes the research in this thesis interesting for developers of the Ace editor.

A major extra advantage of using Ace as our target editor is the fact that Ace is used as primary code editor in the Cloud9 IDE[7]. The Cloud9 IDE consists of a number of general modules which work very well in its client-server architecture such as the storing of a file into the cloud or pushing/pulling commits from/to a version management repository. A Cloud9 IDE can be hosted on any platform since it is fully written in JavaScript itself and runs on NodeJS. Cloud9 IDE serves purely in a browser runnable client side code and does not require any third party plug-ins such as Silverlight or Java. In the spoofax2ace tool we will present in this chapter we have included a Cloud9 IDE plug in as one of the targets to which a Spoofax project can be converted to.

6.1.2 Previous work

In [49] a working proof of concept syntactically-aware Ace editor is presented. The most interesting part in this work for our proof of concept is the implementation of a hook into the inner workings of Ace with respect to events. They have for instance customized some often-occurring aspects such as the insertion of a new line. When a new line is inserted into the editor they have made sure that all tokens in the next lines shift a position down, omitting the requirement to re-execute the entire syntactic and semantic analysis for the currently open piece of source code.

6.1.3 Back-end to front-end

The Ace editor internally manages the state of a program using a list of tokens. Each token contains for instance information about the symbol it is attached to, its position, the coloring and if applicable error marker. In order to actually display the editor feedback to a programmer we have to extract the required information from the result of a Stratego program. Since we already have implemented a wrapping entity for a Stratego program in Section 4.6.4 we have extended this wrapper to actually return Ace tokens. We have implemented a term visiting function which exhaustively extracts information from the traversed AST, and the errors/warnings/notes which were reported by the original Stratego program.

[6] http://mxr.mozilla.org/mozilla/source/js/narcissus/
[7] https://c9.io/

Finally we have added a listener to the “the editor text has changed” event in Ace. When this event fires we call the wrapped JavaScript implementation of the Stratego program which parses, analyzes and tokenizes the source code into usable Ace tokens. Thanks to the previous work regarding WebWorkers described in Section 4.4.1 we were able to implement this functionality without dramatically affecting the user interface responsiveness. Using the implemented proof of concept on the basic Stratego wrapper described in Section 4.6.5 we have implemented a semantically-aware Web editor for the Entity language, which was introduced in Section 2.4.1.

In Figure 6.1 we have included a screenshot of the semantic analysis of the Entity language from Section 2.4.1 running in the Chrome browser.

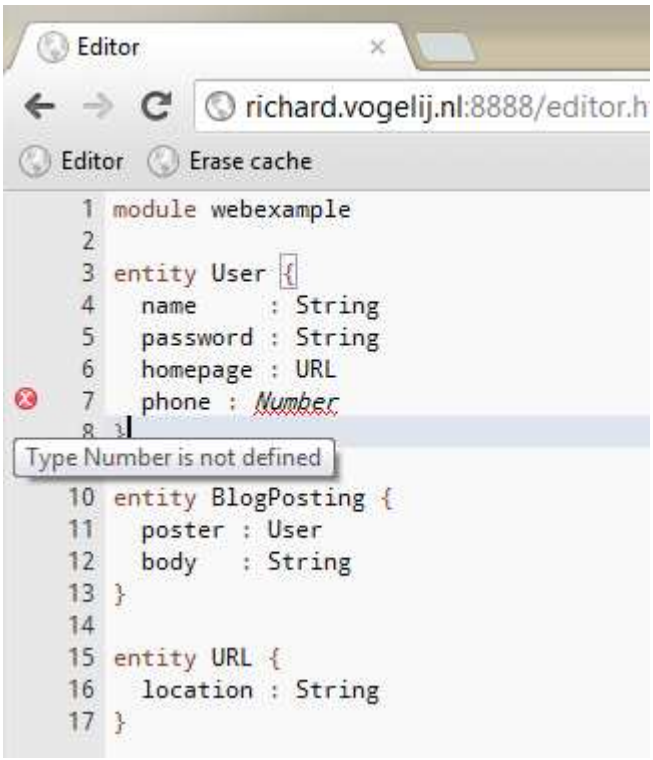

Figure 6.1: A semantically aware Ace Web editor running the Entity language example

6.2 Spoofax

Since it would be beneficial to automate the process of obtaining a Web Editor from a Spoofax project we need to investigate where the required language-specific dependencies are located. In this section we will discuss which Spoofax based dependencies there are and how we incorporated them into Spoofax2ace.

6.2.1 Settings

We have implemented a tool which obtains the required, project specific, information from a Spoofax project. Such information could for instance be the name of the Stratego based entry point for the build product, described in Section 2.4.1. Thankfully, Spoofax stores such language-specific settings in a specialized configuration file for which we have implemented a parser which can return the various project-specific variables to the Spoofax2Ace generation program.

6.2.2 Build products

Spoofax transforms a contained Stratego program into an intermediate ATerm representation of the Stratego program, namely a *.ctree* file. A language designer could potentially influence this ATerm file to implement some custom required behaviour in the resulting Java implementation of the Stategо program. Because of this possibility, we have decided to support such a *.ctree* file as actual input for the s2js compiler. Spoofax regenerates this *.ctree* file whenever a piece of Stratego code is changed. The second major dependency our Spoofax2Ace tool relies on is the availability of an SGLR parse table. Spoofax also initiates the regeneration of such a parse table based on the alteration of its source SDF definitions. The generation of the *.ctree* and *.tbl* files are handled through a set of Ant[8] build files.
Since it would be beneficial to not rely on Spoofax to actually produce the files Spoofax2Ace requires in order to generate a JavaScript implementation of the analysis, we have attempted to automate the invocation of these Ant build files. Even though this works in most cases, we have been unable to iron out some problems which occasionally rise due to custom build steps and complexities language designers could implement. Due to these issues we require the required files to be available and generated by Spoofax for the time being.

6.3 Dependencies

Because there are multiple involved technologies, Spoofax2Ace couples a number of other projects together in order to be able to produce a Web editor. Currently these dependencies are:

- JSSGLR (Section 4.4)
- S2JS (Section 4.6)
- The Stratego Libraries

The first two requirements should be obvious due to our previous discussions regarding parsing, semantic checking and the technologies we will use to achieve this. The Stratego Libraries contain often used strategies which most Stratego programs reference. The location of the actual files can however change from system to system which is why we require a user of Spoofax2Ace to specifically define their location.

[8] `http://ant.apache.org/`

6.4 Target platforms

Even though we are targeting JavaScript as platform, there are some subtle differences in the potential build targets for Spoofax2Ace which have a potential relevant effect on the resulting binaries. We have for instance made a separate target for implementations which are going to be benchmarked. The actual measurements and optional preheating runs can pose a significant decrease in production-performance. Also, during the development of the s2js compiler we wanted to be able to make use of profiling tools in order to pinpoint obvious problems in the JavaScript implementation. The profilers we have used, Firedebug [9] and Chrome's internal profiler are unable to function on code which runs in a WebWorker, described in Section 3.1.4. Therefore we have added a build target which runs the analysis in the UI thread which is of course not very usable in production. Currently, Spoofax2Ace can generate binaries for the following targets:

- NodeJS Executable
- Ace Editor
  - Production
  - Extended Debugging
- Cloud9 plug in files
- Chrome and FireDebug profilable implementation

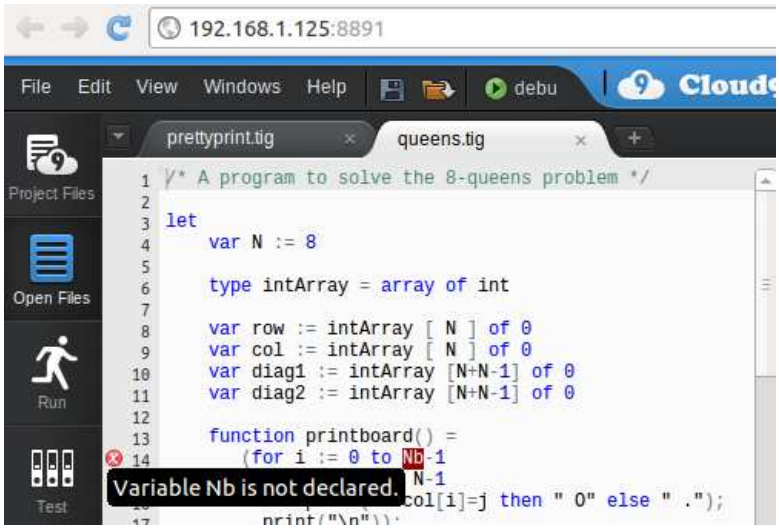

Figure 6.2: A Cloud9 Web IDE showing Tiger-language semantic feedback

[9] https://addons.mozilla.org/nl/firefox/addon/firebug/

We have also included some options to only generate a syntax checker and omit the semantic checks in order to speed up benchmarking. To actually generate one of these targets a Spoofax project root has to be specified. Optionally an example program for the target language can be specified which will be the generated editor's default content. For convenience we have also included a minimal HTTP server script which serves the generated Ace editor. Because the generated client-side files are static, these files can also be uploaded to a regular web service since there there are no server-side requirements.

In Figure 6.3 we have added a slightly modified screenshot. The modifications we have made in this screenshot are the visibility of multiple error messages at the same time. Showing multiple error messages at the same time is normally not possible because an error message is only shown when a user hovers his mouse over the error marker.

We have also included a screenshot of the Cloud9 IDE in which a language analysis for the Tiger language, introduced in Section 3.1.6, is visible in Figure 6.2. The language plug-in for this Cloud9 IDE instance was obtained using the spoofax2ace tool.

As final addition we have added the possibility to target a client-server based implementation in which all calculations are actually run remotely on a server or in the cloud. In Chapter 5 we have discussed this approach.

6.5 Client/Server Balancing

In Chapter 4 and Chapter 5 we have presented our two main approaches. The fully client and the client-server based approach. Our editor approaches can be combined into a single back-end which can take either of these strategies to perform the program analysis. In this section we will discuss the possibilities in automating the decision regarding which strategy to use.

The notion of balancing we present in this section is an obvious next step after designing both our fully client and client-server based approaches. This was however one of the final additions to this thesis and even though our proof of concept implementation is functional the contributions and ideas we present in this section are mainly conceptual.

6.5.1 Approaches

Initially, one might think the best strategy is the fastest client-server approach. There are however a number of possible scenarios in which this is undesirable. For instance the added costs we described in Section 5.1.2 when providing a server with more workload is a reason to move load more to the slower client side. A user of the editor could also prefer the fully locally running analysis due to for instance roaming costs when using a dongle while travelling.

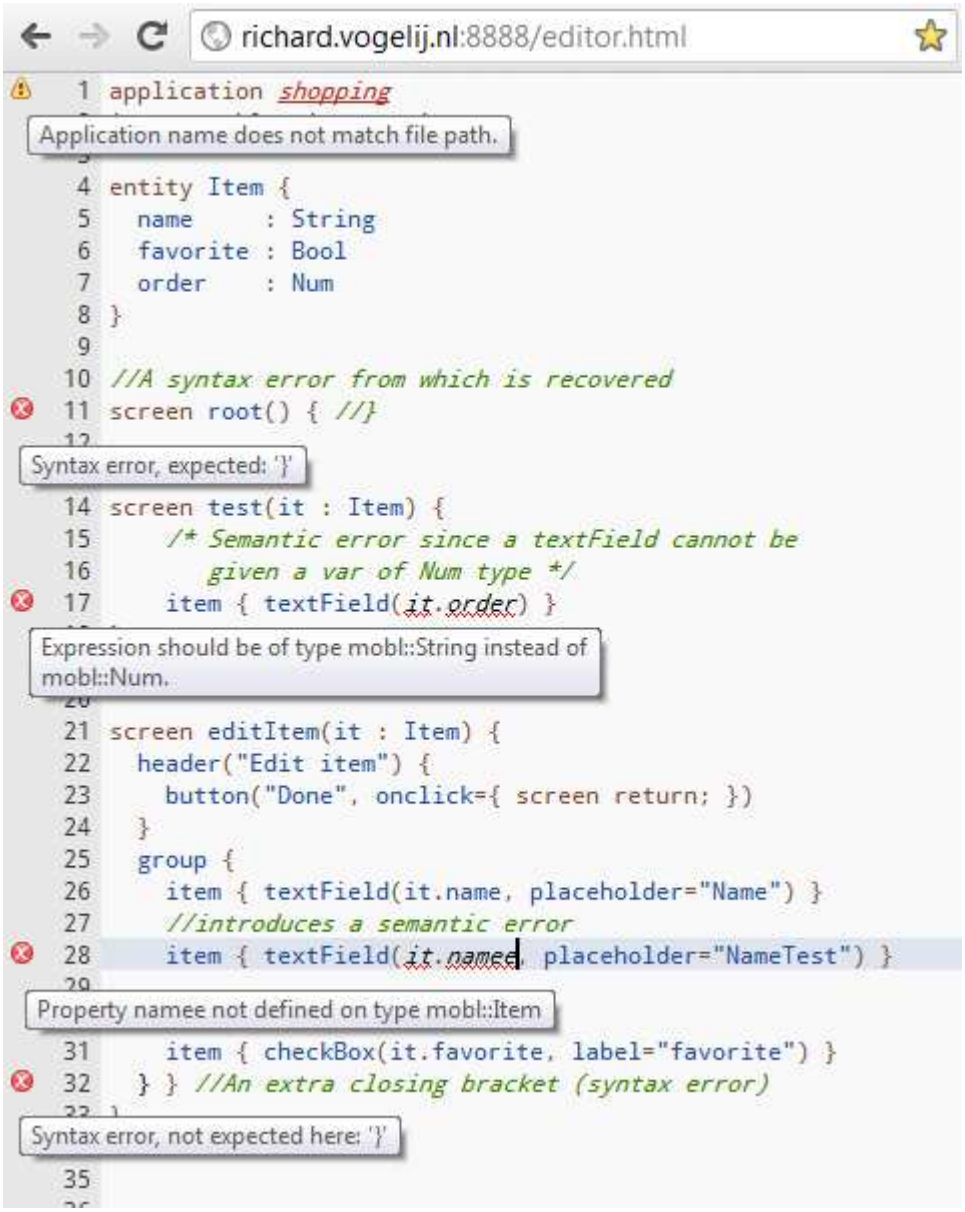

Figure 6.3: A showcase of the features of our generated web-editor for a mobl program

Fully in the browser

Our initial approach in Chapter 4 revolves around running the syntactic and semantic analysis fully in the browser. Even though the analysis does not currently execute as fast as a Desktop based implementation it is certainly functional. Compared to the client/server approach, the fully client based approach is far less dependent on outside influences such as

the unavailability of a stable internet connectivity or a failing remote server.

Client/Server

In Chapter 5 we have presented our server based proof of concept design. We can in fact divide that solution up into two parts, namely the CPU friendly but bandwidth heavy "raw-data" sender version and the bandwidth light but CPU heavy *diff/match/patch* version.

This division could be relevant because on fast computers the most beneficial way to go will most likely be the *diff/match/patch* based version of our server based analysis. Especially when bandwidth is scarce. However, on certain devices which have less CPU power such as a tablet, it could in fact be more beneficial to utilize the bandwidth heavier, but less CPU intensive, approach. Especially if the editor runs on a device with a fast internet connection. In fact, because JavaScript is becoming increasingly important due to the rise of Web applications, JavaScript engines are continually optimized and the gap between Java and JavaScript performance is decreasing.

6.5.2 Determining the optimal approach

Based on the work we have presented in this document we define the following three possible options:

1. Fully client side, browser based
2. Server side calculations, transmitting raw data
3. Server side calculations, transmitting patch data

We would like to present an algorithm which determines the optimal approach for the current situation in which a user is editing code. By this we strive to have the algorithm switch to a fully client-side mode in for instance the event of a connection failure. If all options are possible, all options should be occasionally attempted and weighted in order to obtain a preference toward the most suitable underlying technology for a current situation.

This attractiveness should be influenceable by the user and the Web administrator who hosts the Web editor in order to represent dynamic conditions regarding for instance costs while roaming. Our asynchronous algorithm is defined in Figure 6.4.

In this algorithm, $R[n]$ stands for an array of execution times of the various analysis implementations which are supported. When such a mode is started a penalty P is added to the appropriate row $R[i]$. When the analysis finishes the penalty P is subtracted and the execution time is added from $R[i]$. In the event of a failure an extra penalty is added on top of the initial penalty P which is likely not going to be subtracted again.

A timer at an interval I is also defined. Its purpose is to vaporize the values in R. The vaporization rates are defined in V for each analysis mode. The value must be $\in [0,1]$. The closer to 1 this constant is defined, the faster its R will vaporize, meaning that approach will

Initialization:
$R[0..2] \leftarrow 0$;
$V[0..2] \leftarrow 0.01$;

Perform Analysis:
$mode \leftarrow Min(R).Index$;
$S \leftarrow Now$;
$R[mode] \leftarrow R[mode] + P$;
$call(mode)$;

Analysis Finished:
$elapsed \leftarrow Now - S$;
$R[mode] \leftarrow R[mode] + elapsed - P$;

Analysis Unsuccessful:
$R[mode] \leftarrow R[mode] + P$;

At interval I:
$for(i \leftarrow [0..2])$
  $if(R[i] > 1)$
    $R[i] \leftarrow R[i] * V[i]$

Figure 6.4: Event-based pseudo code of our client-server balancing algorithm

be attempted more often. This allows a hosting entity to define a value for V in which calculations are done client-side where possible, but keep the possibility open to automatically switch to a server-backed analysis. We have embedded this algorithm in the product the spoofax2ace tool generates. We have executed some minor tests in various circumstances and the proof of concept implementation chose strategies as expected when we for instance switched off our networking connectivity or sabotaged network latency.

Chapter 7

Future work

In this section we will speculate on some useful subjects for future work, improving and extending the various approaches and ideas we have presented in this document.

7.1 Optimizations

Because our Stratego to Java to JavaScript approach, discussed in Section 4.5, was quite prohibitive in introducing JavaScript specific optimizations, we have created the s2js implementation in order to facilitate the implementation of specialized JavaScript optimizations. We have already implemented a number of optimizations which allowed us to roughly match the performance we got from the Stratego to Java to JavaScript approach by pinpointing performance bottlenecks using profiling tools, as discussed in Section 6.4.

Even though we have suspended our efforts in optimizing the s2js approach in favor of our client-server approach, there are still interesting angles for future work in continuing the s2js optimization process. The s2js is, in our opinion, the most promising fully client-side based analysis and deserves to be investigated and developed further.

As mentioned in Section 4.4, we have used GWT to port the Java based JSGLR parser to JavaScript. In Section 4.6 we describe a custom tree builder which outputs simplified, manually implemented, JavaScript terms. The inner workings of JSSGLR are however still using much simulated Java functionality. We suspect that implementing the SGLR parser algorithm in native JavaScript, putting effort into JavaScript-specific optimizations could produce a faster SGLR implementation than GWT could produce out of JSGLR.

7.1.1 Stability

The client-server based approach we have presented in Chapter 5 was highly optimized by adding the diff/match/patch algorithm. There are some issues which could rise in a number of steps of this approach. For instance our fragmentation protocol has been implemented without a consistency check on a reconstructed string at the remote side. Even though we did not encounter any issues regarding corrupt messages in our extensive tests, adding a

CRC or hash would still be a useful addition in order to prevent potential problems during production.

7.1.2 Security

Currently, the server based approach implements no security at all. Because all program source code must at some point be fully transferred to the server in order to perform the analysis, there is a potential security issue. The most simple solution to solving this issue is to configure the HTTP server which hosts the WebSocket servlet to only serve a HTTPS connection. The most elegant addition would however be to incorporate a credentials system which ideally also handles server-side access to referenced files.

7.2 Extensions

7.2.1 Editor services

We have only implemented the errors, warnings and notes a typical Spoofax based language analysis returns. Spoofax however also provides the definition of hover help, code folding, code outlining, content completion and reference resolving. Even though our efforts in getting the Stratego based analysis to run in a browser were successful, there is still some future work required to implement these extra editor services. Fundamentally all of these editor services can be implemented using Stratego+SDF and can therefore also run in the browser based on the work presented in this thesis. The main difficulty in implementing these back-end functions are to obtain an AST node from a location in the source code. This required information is however available in the back-end and could be efficiently stored in order to obtain a tree node object based on a line and character number. From these editor services content completion is most likely to be the most challenging because, obviously, at the moment of initiating a content completion request there is likely no syntactically correct program. As mentioned in Section 2.2.4, the research in recovering erroneous ASTs [13, 36] is likely a suitable approach.

7.2.2 File API

In s2js we have introduced a major simplification by disallowing File I/O operations. As stated in Section 4.6.3, Stratego library functions eventually reach a primitive function. With File I/O based functionality this is no exception. Currently we return empty data in order to make sure a Stratego program running in JavaScript does not fail. A useful point of future work would be the implementation of these File I/O related primitive functions and investigating the possibilities of some sort of AJAX or WebSocket based file system simulation. This simulation should of course trigger actual requests to a server which contains the files which are referenced.

Alternatively, by extending the s2js compiler, work can be done to determine which files can ever be referenced by some input Stratego program and actually embedding these file

contents into the JavaScript result itself or some other resource which is cached at the user's machine.

7.2.3 Partitioning

The algorithm which handles the balancing between fully client and client-server approaches we have presented in Section 6.5 is rather simple. We simply choose between running an analysis *fully* at the client side or *fully* at the server side. An interesting field of further research would be to investigate the issues which rise when attempting to balance smaller portions of work. It could for instance be beneficial to execute a syntactic check locally in the client itself frequently and a semantic check which runs on a server with a smaller frequency. Another example would be the execution of a parser at the client side, and sending (a portion of) an AST to a server which would then execute a semantically aware operation based on previous knowledge.

7.2.4 Bootstrapping

Since s2js, the Stratego to JavaScript compiler, is written in Stratego itself, it would be an interesting achievement to compile s2js to JavaScript and run it in the browser. If this implementation which runs in the browser is used to compile the Stratego code s2js consists of a bootstrap would have effectively been made. The advantage of bootstrapping s2js would implicitly mean that there is no longer a fundamental requirement for either the Stratego to C (strc) or Stratego to Java (strj) compiler in order to compile and run a Stratego program in the browser.

7.3 Collaboration

Because of the nature of the Web is all about connections and communication, a natural next step would be to introduce a form of (real-time) collaboration. The Google document application provides a functionality to watch a collaborator edit a shared document in real-time. Because in most software projects there are multiple developers working on the same project at the same time, real-time collaboration in which programmers can be physically apart can still engage in for instance a session of peer-programming [38]. At the time of this writing, the Cloud9 IDE is starting to support real-time collaboration.

7.4 Editor state URIs

In our research paper "Software Development Environments on the Web: A Research Agenda" [38], an interesting angle for web-editors is introduced. Because a web-editor runs fully in the browser there fundamentally are very little to no machine or installation specific configuration settings. All required data could be stored in the cloud making it possible for a programmer to turn off his laptop and continue his work straight away by simply opening his editor instance from another computer. Continuing on the possibilities it could be of great help to be able to store a certain state of the editor and obtain a URI which points

to the editor instance in that particular state. In open source projects there could be a major shift in the way bug-reports are presented because bugs could be pinpointed in the code itself in a certain state after which an author of the involved code could simply open up the URI and fix the problem. Managing these states would however be a complicated area of research because of the sheer potential size such a state would consist of.

Chapter 8

Related work

8.1 Source code editors

JS Fiddle JS Fiddle[1] is an online tool in which a web developer has four windows. An HTML, a CSS, a JavaScript and a result window. The code view supports simple syntax highlighting. Using a run button the result window is filled with the resulting HTML page. As the name suggests, the main intention of this tool is to help developers quickly try out things without having to redeploy or save multiple files every time.

Codemirror The Codemirror[2] editor is a JavaScript based component which can be embedded in a web page. The CodeMirror project is maintained by one person, Marijn Haverbeke. It supports a large number of languages in the form of syntax highlighting and for some languages auto completion. The support for every language has been manually crafted and heavily relies on the use of regular expressions. The Codemirror project is one of the oldest web editors around.

Ace Ace, the Ajax.org Cloud9 Editor[3] is similar to Codemirror. It is also based fully on JavaScript and natively supports a large amount of languages. The well known Web editor projects Mozilla Bespin and Skywriter were merged to form the Ace project. Ace support many features such as code folding, bracket matching and auto indentation. Quite notably the Ace editor is used on the GitHub [4] website providing users an elegant way to edit and commit changes to their repositories without the requirement to checkout the files to their local machine first.

[1] http://jsfiddle.net/
[2] http://codemirror.net/
[3] http://ace.ajax.org/
[4] https://github.com

8.2 Web IDEs

With the uprising of Web-applications the field of Web IDEs is expanding at a rapid pace [2]. There are a number of projects which provide a Web IDE. In this section we will discuss a number of these Web IDEs.

Cloud9 The Ajax.org Cloud9 editor [5] is a Web IDE which is actively being developed. Due to its open source nature the community can make additions which are often adopted.The Cloud9 IDE supports a number of programming languages such as C-Sharp, Java and PHP. The main programming language the Cloud9 IDE supports is currently JavaScript. Cloud9 provides practically all features a typical Desktop IDE also would. There is for example support for a file system, step/trace debugging and deploying a program into a production environment such as Azure[6]. At the time of the writing of this thesis, Cloud9 has also recently released a collaboration feature which offers real-time collaborative editing of a program in a group, similar to Google Documents.

eXo Cloud The Exo Cloud IDE [7] is very similar to Cloud9 IDE. It too provides practically all functionality a Desktop based IDE would. The IDE currently supports ten programming languages which is less than Cloud9. The editor supports syntax highlighting but lacks real-time feedback regarding syntax or semantic errors. The Exo Cloud IDE also supports deploying to production environments and collaborative coding.

CodeRun The CodeRun Web IDE [8] is a code editor which only supports C-Sharp, JavaScript and PHP. Regretfully there are not much technical details available regarding the editor portion of the IDE. We have tested this IDE and were particularly impressed with the code completion feature for C-Sharp which produces the exact same results MicroSoft Visual Studio does. Code completion is not available for the other supported languages. We have confirmed that the editor supports syntax highlighting, but due to the lack of technical information and insight into the inner workings we were unable to determine whether the editor can actually show syntax or semantic errors. During our tests we were only able to get some feedback regarding errors in the program being edited by initiating a build and viewing build errors show up in a log list. As the other IDEs we have presented in this related work section, this IDE also supports the deployment to various production sites.

MiDebug MiDebug [54] is a web-based IDE for embedded system programming. They have created a browser plug-in which can directly communicate with a physical micro controller. Using this link they actually compile the program which is being edited server side, after which the program is deployed to the physical device. Step and trace debugging is made possible through the use of a persistent connection between the server back-end and

[5] https://c9.io/
[6] https://www.windowsazure.com
[7] http://cloud-ide.com
[8] http://www.coderun.com/ide/

the physical device through JavaScript WebSockets. The source code editor which was used is based on the Codemirror project. The MiDebug project is currently being used as a teaching aid for students at the Electrical Engineering Department at the University of California.

Collabode Much of current related work is done behind a collaboration mask. The work done for Collabode is performed under the collaboration flag but in fact is based on Web editors, making the project an interesting topic for our research. In “Real-Time Collaborative Coding in a Web IDE” [24, 23], a Web IDE is presented which features full syntactic and semantic support for Java. They utilize Eclipse at the server side in order to obtain syntactic and semantic feedback. This approach was also proposed in [59]. The front-end is implemented using Ace. The focus of their work is in the issues which rise when multiple people are editing the same file at the same time. They for instance discuss questions as to what should happen when a peer is causing errors in a file which is being edited by multiple programmers at the same time. They present an algorithm to cope with such issues which, as stated in Section 7.3, will be interesting for future work regarding the design presented in this thesis.

Chapter 9

Contributions and Conclusions

9.1 Contributions

We have investigated the feasibility of generating a semantically aware web-editor. In the process we have laid the groundwork for much potential future research. We have produced a portable implementation of our proof of concepts which can either be run from the command line or, in the target scenario, a browser. Further more we have presented a reusable approach to uniformly obtain benchmark results from multiple origins, both locally but also remotely executed code in Section 4.3.

Quality checks With the exception of actual GUI components, all of the JavaScript based software and proof of concept implementations we have presented in this thesis can be run on the command line using a JavaScript interpreter such as NodeJS. We have implemented a number of angles for each component in order to gain statistics in an automated way. These measurements can easily be repeated and even reused in order to extend our load-balancing algorithm with.

9.1.1 JSSGLR

Even though we did not initiate the JSSGLR project we have certainly made useful contributions. We have taken extensive measurements in order to confirm the scalability of the JavaScript based SGLR parser and included an alternative, light weight, output format for use in the S2JS output in Section 4.6.

9.1.2 Proofs of concept

We have presented two major possible approaches in implementing a browser-based semantically aware source code editor. With exception of the availability of a recent installation of a web browser, both proof of concepts do not require a user to install any extra arbitrary software such as browser-plug ins. We have used our generator from Chapter 6 to derive editors for three languages. We have included screenshots of the various implementations

while focusing on specific aspects for the Entity language in Figure 6.1, the Tiger language in Figure 6.2 and finally the Mobl language in Figure 6.3.

Client

The analysis which runs fully in a browser is in our opinion the most interesting approach. We have extended and optimized the s2js compiler in order to obtain a functional semantic analysis executing back-end which entirely runs on JavaScript. We have incorporated multiple build targets which provide the ability to produce a back-end which is optimized for release, debugging or performance benchmarking.

Server

Our second main approach was the design of the client-server based editor in which the actual semantic analysis logic is executed on a remote server instead of in the browser at the client in Chapter 5. Even though our initial angle was mainly aimed at running the actual analysis fully on the browser, a server side will always be required in order to safely store work. Also, looking at the future in which collaboration based editors are becoming more popular, our research and proof of concept design for a client-server based analysis could prove to be useful.

Balancing

The final contribution we have made is to tie our previous proof of concept implementations together in an attempt to extract and use the best portions of all. We have done this by designing and implementing a load-balancing algorithm in Section 6.5 which makes an effort to choose the best strategy to take in order to obtain the required analysis results. In order to also make it possible for a hosting provider to tweak these settings we have introduced the notion of weights which make it possible to bias the algorithm toward a certain strategy such as a fully off-line analysis or a hybrid behavior.

9.2 Conclusions

The previous section has been a summary of the contributions we have made. In this section we will mainly discuss and answer our research questions we have presented in Section 1.4.

9.2.1 Research questions

- *Can a web-based, Desktop IDE quality, source code editor be created for any context-free programming language?*
  We have identified the features current IDEs provide in their source code editing components. We have enumerated and illustrated some well known forms of feedback such editors provide in Section 1.3. During our research we have concluded that all these features can eventually be based the availability of syntactic and semantic knowledge about the program which is being modified in Chapter 2. We have also

determined an SGLR parser can recognise all context-free languages in Section 2.2.4. Obviously, on the Web there are restrictions regarding the technologies which can be used as explained in Chapter 3. These restrictions essentially mean the requirement to base an implementation on JavaScript. A front-end to actually handle user input and present feedback also has to be used or created. We have shown that it is indeed possible to obtain a JavaScript implementation of an SGLR parser and semantic analysis in Chapter 4 which positively answers this research question.

- *Is generating a semantically aware Web based source code editor feasible?*
  Since we have based our work on Spoofax and technologies Spoofax uses we were able to take advantage of the declarative world Spoofax is based on. In Chapter 4 and Chapter 5 we have presented two approaches in obtaining a web based source code editor from a Stratego program and an SGLR parse table. These approaches consist of a number of steps and rely on the availability of multiple dependencies. Spoofax2ace, which is described in Chapter 6 is an attempt at automating most of these steps by retrieving required language specific parameters from a Spoofax project and performing all the necessary steps in order to generate a functional web-editor. Even though there are currently a small amount of manual steps, enumerated in Section 6.2.2, involved due to possible specific primitive functions of an underlying Stratego implementation we answer this research question with a *yes*.

9.2.2 Remarks

During the course of this master's thesis we have investigated what the language specific components in a typical source code editor of IDE quality are. We have enumerated a list of common editor features and identified the shared core technologies these features are built on. In the Spoofax project, these core technologies lie in the SGLR parser for syntactic, and Stratego for semantic functionalities. We have discussed JSSGLR, the JavaScript SGLR parser and s2js, the Stratego to JavaScript compiler in which we have made these technologies available for a browser environment. These tools have been added together in spoofax2ace, the Spoofax to Ace Web-editor generation utility.

We have performed extensive measurements in order to compare the various techniques we have considered. From these measurements, we have concluded that our client/server based approach yields results with the best performance. We are however of opinion that a fully client-side analysis would be the most elegant solution. In order for this approach to become more viable there are some optimizations which could be implemented. Also recent developments in the field of JavaScript engines suggest a promising prospect. Looking back at the proof of concept implementations and the performance we were able to achieve we are very content with our results.

Bibliography

[1] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.

[2] T. Aho, A. Ashraf, M. Englund, J. Katajamäki, J. Koskinen, J. Lautamäki, A. Nieminen, I. Porres, and I. Turunen. Designing ide as a service. *Communications of Cloud Software*, 2011.

[3] A.W. Appel. *Modern compiler implementation in ML*. Cambridge Univ Pr, 1998.

[4] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[5] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, and P. Friese. Xtext user guide. *Dostupné z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html*, 2008.

[6] T. Berners-Lee, L. Masinter, M. McCahill, et al. Uniform resource locators (url). *CERN*, 1994.

[7] T. Boudreau. *NetBeans: the definitive guide*. O'Reilly Media, 2002.

[8] E. Bozdag, A. Mesbah, and A. Van Deursen. A comparison of push and pull techniques for ajax. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 15–22. IEEE, 2007.

[9] M. Bravenboer, K.T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008.

[10] N. Chomsky. *Aspects of the Theory of Syntax*, volume 119. MIT Press (MA), 1965.

[11] S. Cook, G. Jones, S. Kent, and A. Wills. *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, 2007.

[12] D. Crockford. The application/json media type for javascript object notation (json). 2006.

[13] Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In Mark G. J. van den Brand and Jeff Gray, editors, *Software Language Engineering (SLE 2009)*, volume 5969 of *Lecture Notes in Computer Science*, pages 204–223, Heidelberg, October 2009. Springer.

[14] E. Dolstra and A. Löh. Nixos: a purely functional linux distribution. In *ACM Sigplan Notices*, volume 43, pages 367–378. ACM, 2008.

[15] F. Durán, M. Roldán, J.C. Bach, E. Balland, M. Van Den Brand, J. Cordy, S. Eker, L. Engelen, M. De Jonge, K. Kalleberg, et al. The third rewrite engines competition. *Rewriting Logic and Its Applications*, pages 243–261, 2010.

[16] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[17] S. Efftinge. Xtext reference documentation. *last accessed on May*, 2009.

[18] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[19] I. Fette and A. Melnikov. The websocket protocol. 2011.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[21] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., 2003.

[22] S. Ginsburg. *The mathematical theory of context free languages*, volume 3. McGraw-Hill New York, 1966.

[23] M. Goldman, G. Little, and R.C. Miller. Collabode: collaborative coding in the browser. In *Proceeding of the 4th international workshop on Cooperative and human aspects of software engineering*, pages 65–68. ACM, 2011.

[24] M. Goldman, G. Little, and R.C. Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 155–164. ACM, 2011.

[25] C.A. Gutwin, M. Lippold, and TC Graham. Real-time groupware in the browser: testing the performance of web-based networking. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 167–176. ACM, 2011.

[26] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdfreference manual. *ACM Sigplan Notices*, 24(11):43–75, 1989.

[27] Z. Hemel and E. Visser. Mobl: the new language of the mobile web. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 23–24. ACM, 2011.

[28] Zef Hemel and Eelco Visser. Declaratively programming the mobile web with mobl. Technical Report TUD-SERG-2011-024, Delft University of Technology, Delft, The Netherlands, August 2011.

[29] Zef Hemel and Eelco Visser. Mobl: the new language of the mobile web. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 23–24. ACM, 2011.

[30] Zef Hemel and Eelco Visser. Programming the Mobile Web with Mobl. Technical Report TUD-SERG-2011-01, Delft University of Technology, January 2011.

[31] S.C. Johnson and inc Bell Telephone Laboratories. *Yacc: Yet another compiler-compiler*. Bell Laboratories, 1975.

[32] L. Kats, K.T. Kalleberg, and E. Visser. Generating editors for embedded languages. Technical report, Technical Report Series TUD-SERG-2008-006, Delft University of Technology, Software Engineering Research Group, 2008. http://swerl. tudelft. nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-006. pdf, 2008.

[33] L.C.L. Kats. Building blocks for language workbenches. *month*, 2011.

[34] L.C.L. Kats, K.T. Kalleberg, and E. Visser. Domain-specific languages for composable editor plugins. *Electronic Notes in Theoretical Computer Science*, 253(7):149–163, 2010.

[35] L.C.L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. *ACM Sigplan Notices*, 45(10):918–932, 2010.

[36] Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In Gary T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *ACM SIGPLAN Notices*, pages 445–464, New York, NY, USA, October 2009. ACM Press.

[37] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463. ACM, 2010.

[38] Lennart C. L. Kats, Richard G. Vogelij, Karl Trygve Kalleberg, and Eelco Visser. Software development environments on the web: A research agenda. In *Proceedings of the 11th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (Onward 2012)*. ACM Press, 2012.

[39] D.E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[40] R. Kurki-Suonio. Notes on top-down languages. *BIT Numerical Mathematics*, 9(3):225–238, 1969.

[41] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen, and M. Englund. Cored: browser-based collaborative real-time editor for java web applications. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1307–1316. ACM, 2012.

[42] P. Lubbers, B. Albers, and F. Salim. Using the web workers api. *Pro HTML5 Programming*, pages 241–262, 2011.

[43] S. McPeak and G. Necula. Elkhound: A fast, practical glr parser generator. In *Compiler Construction*, pages 2725–2725. Springer, 2004.

[44] T. Mikkonen and A. Taivalsaari. Using javascript as a real programming language. 2007.

[45] E.W. Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[46] T. OReilly. What is web 2.0. *Design patterns and business models for the next generation of software*, 30:2005, 2005.

[47] T.J. Parr and R.W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[48] L. Powers and M. Snell. *Microsoft® visual studio 2008 unleashed*. Sams, 2008.

[49] S. Rabbelier. Declarative specification of web-based integrated development environments, 2011.

[50] D. Raggett, A. Le Hors, I. Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[51] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, 1970.

[52] A. Russell. Comet: Low latency data for browsers. *alex. dojotoolkit. org*, 2006.

[53] D.J. Salomon and G.V. Cormack. Scannerless nslr (1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, 1989.

[54] C. Shen, H. Herman, Z. Charbiwala, and M.B. Srivastava. Midebug: microcontroller integrated development and debugging environment. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, pages 133–134. ACM, 2012.

[55] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. *Internet Computing, IEEE*, 14(6):80–83, 2010.

[56] M. Tomita. *Generalized LR parsing*. Springer, 1991.

[57] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Compiler Construction*, pages 21–44. Springer, 2002.

[58] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, et al. The a sf+ s df meta-environment: A component-based language development environment. In *Compiler Construction*, pages 365–370. Springer, 2001.

[59] A. Van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi. Adinda: a knowledgeable, browser-based ide. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 203–206. ACM, 2010.

[60] E. Visser. Scannerless generalized-lr parsing. Technical report, Citeseer, 1997.

[61] E. Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *Rewriting techniques and applications*, pages 357–361. Springer, 2001.

[62] E. Visser. Webdsl: A case study in domain-specific language engineering. *Generative and Transformational Techniques in Software Engineering II*, pages 291–373, 2008.

[63] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

Appendix A

Glossary

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

AST: Abstract Syntax Tree

SGLR (parser): Scannerless Generalized Left-to-right Rightmost derivation (parser)

JSGLR: Java based SGLR parser

SDF: Syntax Definition Formalism

GWT: The Google Web Toolkit

ATerm: Annotated Term

strj: The Stratego to Java compiler

strc: The Stratego to C compiler

s2js: The Stratego to JavaScript compiler

s2a: The Spoofax to Ace conversion tool

ACE: The Ajax.org Cloud9 Editor

JSON: JavaScript Object Notation