A JavaScript Mode for Yi

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

DENIZ DOGAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

A JavaScript Mode for Yi

DENIZ A. M. DOGAN

Examiner: PATRIK JANSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2009

Abstract

Yi is a text editor written in the lazy functional programming language Haskell, which makes it possible to define precise editing modes using an abstract syntax tree provided online using the lazy and incremental parser library in Yi.

We have developed a JavaScript mode for this editor using this parser library to accurately point out possible errors in the source code. The mode accurately highlights syntactical errors as the user types and provides a verifier to check the semantics of the source code. It supports most of the syntax from JavaScript 1.8 and can readily be extended with more functionality. The mode can also be used as a starting point for future developers of C-like modes for Yi.

Writing a responsive parser for Yi proved non-trivial, because of the trade-off between parser performance and accuracy. This report describes how to write a mode for Yi and the different problems encountered during the working process of the JavaScript mode. It also describes in what ways the problems were solved.

Sammanfattning

Yi är en textredigerare skriven i det lata funktionella programspråket Haskell, som gör det möjligt att definiera noggranna redigeringslägen med hjälp av ett abstrakt syntaxträd som tillhandahålls av det lata och inkrementella parsningsbiblioteket i Yi.

Vi har utvecklat ett JavaScript-läge till denna redigerare med hjälp av detta parsningsbibliotek för att exakt utpeka möjliga fel i källkoden. Läget markerar syntaktiska fel medan användaren skriver och tillhandahåller en verifierare för att kontrollera semantiken i källkoden. Det stödjer större delen av syntaxen i JavaScript 1.8 och kan enkelt utökas med mer funktionalitet. Läget kan även användas som en utgångspunkt för framtida utvecklare av C-lika lägen till Yi.

Det visade sig att det inte är trivialt att skriva en parser till Yi p.g.a. avvägningen mellan prestanda och noggrannhet. Denna rapport beskriver hur lägen till Yi skrivs och de olika problemen vi stötte på under utvecklingen av JavaScript-läget. Den beskriver även hur vi löste problemen.

Acknowledgments

First and foremost I would like to thank my supervisor Jean-Philippe Bernardy for his guidance and support throughout this project.

I would also like to thank Anders Karlsson and Stevan Andjelkovic for many fruitful discussions about Haskell, Yi and the JavaScript mode. Further kudos goes to Patrik Jansson and Jan Rochel for reviewing this report, and also Tobias Olausson, Mosaver Ahmad, Shahrouz Zolfaghari, Johan Knutzen, Ivan Majdandzic, David Karlsson, Johan Ahl, Andreas Sjöström and Erik Brendengen. Thanks to Piotr Budziszewski for being my opponent for this thesis.

Special thanks to Steve Yegge for writing js2-mode and for taking the time to answer any questions I had about it.

Contents

1 Introduction

JavaScript is a programming language, originally developed by Netscape, which has become increasingly important in web development since its introduction in 1995. It is a dynamically and weakly typed language with a C-like syntax, based on ECMAScript [4]. Contrary to popular belief, the language is not directly associated with the Java programming language or Sun Microsystems. The language was initially called *LiveScript*, but the name was eventually changed because the support for Java in Netscape Navigator was first available at about the same time as JavaScript [7].

1.1 Motivation

There are lots of different JavaScript development environments available today, both as separate editors and as plug-ins for IDEs. However, many of these suffer from drawbacks, some more than others.

This section lists the most common drawbacks that have been identified find in the existing environments for JavaScript development. Extreme corner cases, such as e.g. Microsoft's *Notepad*, have been excluded as they were not intended for programming in the first place.

Too heavyweight This category includes environments which use too much memory and/or CPU time, e.g. *Eclipse*.

Too simplistic Environments and editors which lack support for the most basic features, such as some form of "clever" indentation of blocks, e.g. *GNU nano*.

Too restricted Some programs require a specific operating system to be able to use them and some are only free for a short period of time. This category includes e.g. *Notepad++* and *IntelliJ IDEA*.

Proof-of-concepts One of the most well-known examples of this is Mozilla Labs' *Bespin* [11] which is an in-browser editor and looked promising at first. However, it turned out to be only a nice proof-of-concept, suffering from severe performance issues on some computer systems.

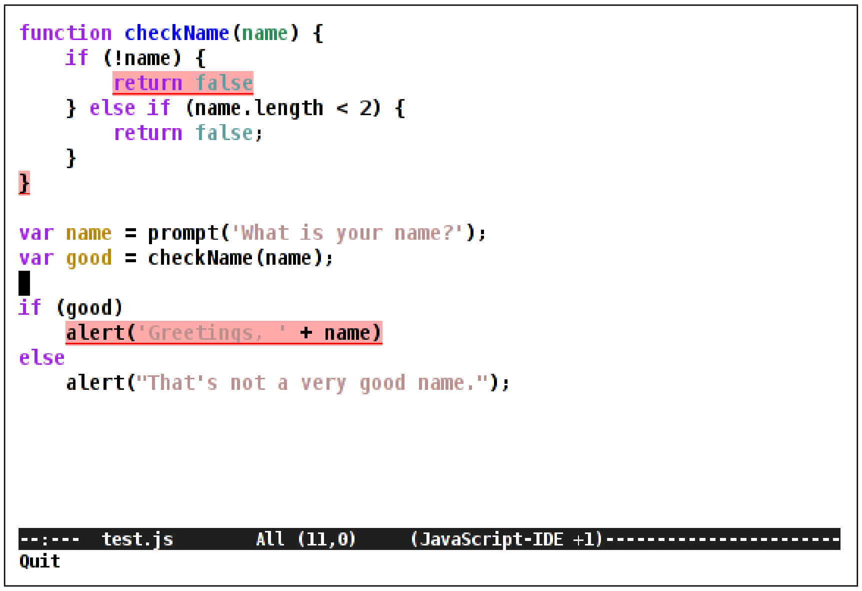
Yi is a text editor written in Haskell which supports advanced syntax highlighting for some programming languages and file formats [1]. It is under active development and has come a long way since the project was started by Don Stewart in 2004 [17]. Yi will be discussed further in section 3.

1.2 Aim

The idea for this project was much inspired by *js2-mode*, a precise JavaScript mode for Emacs [9], written by Steve Yegge, which has support for syntax highlighting and error checking among other features [20]. It works by building an abstract syntax tree of the edited file to accurately report errors to the user.

js2-mode had a few annoying quirks though, such as the suboptimal indentation, the lack of support for JavaScript 1.8 and the mode being sluggish on larger JavaScript files.

The aim of this project is to create a JavaScript mode for Yi, with both the necessary precision to find common JavaScript syntax errors and sufficient performance to be able to use like any other editor. It should be able to accurately highlight regions in which the parser found errors and warn the user about potentially "dangerous" code. An example of dangerous code is functions that are inconsistent in their returning behavior, e.g. one path of execution returning a value while another one does not.[1]



```
function checkName(name) {
    if (!name) {
        return false
    } else if (name.length < 2) {
        return false;
    }
}

var name = prompt('What is your name?');
var good = checkName(name);

if (good)
    alert('Greetings, ' + name)
else
    alert("That's not a very good name.");
```

```
--:---  test.js        All (11,0)      (JavaScript-IDE +1)----------------------
Quit
```

Figure 1: js2-mode in Emacs, displaying a JavaScript file with a few errors: missing semi-colons and inconsistent return behavior. This is largely what we tried to accomplish in the JavaScript mode for Yi.

1.3 Delimitations

The Yi editor is well-documented [1] [2] [17] and has been under active development since 2004. However, prior to this thesis, there was no thorough documentation on how suitable the editor was for mode development. Therefore, for this project we decided not to make any major modifications to the core of Yi (described in section 6), specifically not the parsing library. This was to give an accurate idea of how fitting the Yi project was for developing a

[1] While functions which do not explicitly return anything implicitly return **undefined** (see section 2.4), it is usually not what the programmer had intended and should be avoided.

new mode for it. This report could then be used as the basis for future projects revolving Yi modes, as any major problem encountered will be discussed.

2 How JavaScript works

This section is intended as a short introduction to JavaScript, and mainly its syntax. You do not need to read the entire section, but you should familiarize yourself with how the language works to understand what is being discussed in later sections.

JavaScript is a dynamically and weakly typed programming language, which takes syntactic inspiration from other C-like languages. It is a prototype-based language, providing only a few primitive objects, such as arrays, strings and numbers. JavaScript is case-sensitive and ignores any extra whitespace characters in between tokens.

The language is a superset of ECMAScript [4] and developed by the Mozilla Foundation. One could say that JavaScript is the language that is understood by *Mozilla Firefox* [6] and not entirely like the language understood by *Windows Internet Explorer* [14], which is *JScript* [13]. JScript is also the name of Internet Explorer's JavaScript interpreter.

This thesis focuses only on JavaScript 1.8.

2.1 Reserved words

The set of reserved words tends to change between JavaScript versions. The ECMAScript standard [4] lists the words in Figure 2 as reserved.

Apart from the reserved words, there are also words reserved for future use, to be used if the need for them should arise. These are listed in Figure 3. Since they are not reserved in the same way as e.g. `function` is, they can still be used for e.g. variable names, but programmers are encouraged to avoid them.

```
break    case   catch        continue   default   delete
do       else   false        finally    for       function
if       in     instanceof   new        null      return
switch   this   throw        true       try       typeof
var      void   while        with
```

Figure 2: Current reserved words according to the ECMAScript specification.

2.2 Statements

Statements are what make up a JavaScript program. There are several different statements in JavaScript, most of which have been borrowed directly from other programming languages, such as C.

`abstract`	`boolean`	`byte`	`char`	`class`
`const`	`debugger`	`double`	`enum`	`export`
`extends`	`final`	`float`	`goto`	`implements`
`import`	`int`	`interface`	`long`	`native`
`package`	`private`	`protected`	`public`	`short`
`static`	`super`	`synchronized`	`throws`	`transient`
`volatile`				

Figure 3: Words reserved for future use according to the ECMAScript specification.

Among other statements, there are the compound statements `if..else` statements, `for` loops, `while` loops, `do..while` loops, `for..in` and `try..catch`. Their behavior should already be fairly apparent to the reader and therefore many compound statements have been omitted in this report.

Just like in many other languages, you can group statements in a block statement, which is simply written as `{`, followed by zero or more statements, followed by `}`.

The semicolon is used after each statement to separate it from other statements, but is in fact optional in JavaScript. However, it is considered a bad programming practice to omit the semicolon, because of potential ambiguities that may occur [5].

Any valid JavaScript expression is also a valid statement.

2.3 Identifiers

Since JavaScript supports Unicode, the set of valid identifiers is larger than in many other languages. Almost any sequence of Unicode characters is a valid identifier unless it is a reserved word.

The ECMAScript standard states that a valid identifier begins with a dollar sign ($), an underscore, any Unicode letter or a Unicode escape sequence (e.g. `\u0055`). It can be followed by virtually anything [4]. This means that `\u0055` standing on its own is in fact a valid variable name, albeit probably not a very good one.

2.4 Variables

Variables are declared using the `var` keyword followed by a non-empty comma-separated list of valid identifiers and optional assignments [5]. See Figure 4 for examples. Values which have not been assigned any value are implicitly assigned the value `undefined`.

`undefined` is a special value in JavaScript. It is different from `null` and is only used in special cases, such as uninitialized variables and as the return value in functions which do not explicitly return a value.

Leaving out the `var` keyword in a variable assignment implicitly declares them, but then always in the global scope [5].

```
var x;            // x == undefined
var x, y;         // x == undefined, y == undefined
var x = 3;        // x == 3
var x = 3, y;     // x == 3, y == undefined
var x = y = 3;    // x == 3, y == 3
```

Figure 4: A few example variable declarations in JavaScript

2.5 Arrays

Arrays in JavaScript are similar to arrays and lists in many other languages. They can be created using either the shorthand `[]` syntax or the longer `new Array()`. The user can define arrays with contents on the fly, e.g. `[1, 2, 3, 4]` which returns an array filled with the numbers 1 through 4.

Indexing in arrays is done using the familiar `x[i]` syntax, where `x` is an array and `i` is the zero-based index, as in `[1, 2, 3, 4][0]`, which returns `1`, the first element in the array.

A significant difference between arrays in JavaScript and those of many other languages is that in JavaScript lists are heterogeneous, i.e. they can contain values of different types [5]. This means that e.g. `[1, "hello"]` is a valid array in JavaScript. This particular array holds a number and a string.

2.6 Functions

The syntax for declaring new functions in JavaScript is straight-forward as can be seen in Figure 5. Functions can take any number of arguments and may optionally return a value using the `return` statement. Functions which do not explicitly return any value always return `undefined`.

Functions may be declared inside other functions and inner functions are not available outside of the function in which they were declared. JavaScript functions are also first-class and may therefore be passed just like any other expressions, a feature which is useful for passing callback functions in event-driven programming or for higher-order functions. Being first-class, *anonymous functions*, or *function literals*, are supported. Function literals are written just like any other function, but with the identifier omitted [5].

Since version 1.8, JavaScript supports a shorter function declaration syntax which is just like the traditional function declaration syntax, but with the curly brackets and the `return` keyword omitted.

```
// The traditional syntax.
function squareAndMultiply(x, y) {
  return (x * x) * (y * y);
}

// Uncommon, but valid syntax.
var squareAndMultiply =
  new Function("x", "y", "return (x * x) * (y * y)");

// New syntax, supported since JavaScript 1.8.
function squareAndMultiply(x, y) (x * x) * (y * y)
```

Figure 5: A few different versions of the same function `squareAndMultiply`, which takes two parameters, `x` and `y`. It returns the product of the squares of each parameter. The lowermost version is only supported in JavaScript version 1.8 and newer.

2.7 Objects in JavaScript

In JavaScript, creating an object is just a matter of explicitly writing its properties using a specific JavaScript notation. This format is called JSON (JavaScript Object Notation) and both acts and looks much like dictionaries in Python. An example JavaScript object is shown in Figure 6.

The properties of objects are retrieved using a period followed by the property name, e.g. `ronald.address`. They can also be retrieved using array syntax, e.g. `ronald["address"]` or even a mix of both, as in `ronald.address["street"]`.

```
var ronald = {
  name: "Ronald Smith",
  birth: 1963,
  firstName: function () {
    return ronald.name.split(" ")[0];
  },
  address: {
    street: "Abbey Road",
    city: "London"
  }
};

var x = ronald.firstName() + " on " + ronald.address.street;
```

Figure 6: An example JavaScript with an object representing Ronald Smith, born in 1963. It contains his name (a string), his birth year (a number), his address (another object with two properties) and a function for retrieving his first name. The program assigns the value "Ronald on Abbey Road" to the variable `x`.

2.8 Strings

As we saw in Figure 6, strings are written using the double-quote syntax found in many other languages. They can also be written using the single-quote syntax, just like in e.g. Python. Single characters in strings can be accessed using array indexing (see section 2.5), but unlike languages such as C, strings are *not* represented as arrays of characters. Instead, a single character is simply represented as a string consisting of one character.

Strings are concatenated using the `+` operator and hold some basic methods, such as `substring` and `split`, for extracting substrings and splitting a string up in parts, respectively.

JavaScript strings may consist of any Unicode characters.

2.8.1 Escape sequences

Like many other programming languages, JavaScript supports some escape sequences. The most important escape sequences are shown in Figure 7.

Some languages which resemble JavaScript in these escape sequences allow the user to write strings across multiple lines by putting a trailing backslash character at the end of each line (see Figure 8). This is explicitly forbidden in ECMAScript, but all of the major JavaScript interpreters support this.

As a last note on escape sequences, any unrecognized escape sequence in a string will simply be interpreted as whatever is on the right hand side of the backslash, e.g. `"\#\%\$"` will be interpreted as `"#%$"`.

`\t`	horizontal tab	`\n`	newline
`\r`	carriage return	`\"`	double quote
`\'`	single quote	`\\`	backslash
`\xXX`	hexadecimal character	`\xXXXX`	hexadecimal character

Figure 7: Some escape sequences recognized in JavaScript strings.

```
str = "This is a very long string and reading it becomes\
 much easier if written this way.  It is unfortunate that\
 it is explicitly forbidden in ECMAScript.";
```

Figure 8: A multiline string as written in many programming languages.

2.9 Regular expressions

Regular expressions is another built-in object in JavaScript and even has its own shorthand syntax for defining them, namely forward slashes. E.g. the statement `var rex = /A+B/` assigns the variable `rex` the regular expression matching at least one `A` followed by a `B`. Regular expressions may also be created using the slightly wordier `new RegExp()`, e.g. `new RegExp("A+B")` which is equivalent to `/A+B/`.

2.10 Comments

The comment syntax in JavaScript is just like the one in C, where anything on the same line after and including `//` is a comment. Multiline comments begin with `/*` and end with `*/`. Multiline comments may not be nested [5].

Since JavaScript has a close connection to web development, HTML comments are also comments in JavaScript. HTML comments begin with `<!--` and end with `-->`.[2]

3 What is Yi?

This section is intended as a short introduction to how Yi and Yi modes work. Only the basic features in the editor will be explained and the terminology that will be used in the rest of this thesis. It is assumed that the reader has some basic understanding of how parsers work.

3.1 User configuration

Yi was written with extensibility in mind, taking inspiration from programs such as *xmonad* [18], a window manager written in Haskell, and the *Vim* [15] and Emacs editors. In xmonad, the end-users write their configurations in a special Haskell module which is then recompiled and integrated with the window manager without having to terminate it and restart it again. The recompiling and reloading is usually done by hitting a special key combination. In Yi, it is done by executing a special command inside of the editor.

It is also a bit like Emacs, e.g. letting the users create their own interactive functions (also known as “commands”) and then making it easy to execute them from within the editor. Emacs is built around a Lisp interpreter and lets the user write Lisp code to modify the editor environment “on-the-fly”, without having to restart it.

3.2 Keymaps

Yi comes with a few different keymaps which make the editor act like other well-known editors, such as the aforementioned Vim and Emacs editors. It also provides a keymap for users more comfortable with the “standard Windows” key bindings for cutting, pasting, etc. Extensible as the editor is, the user can quite easily write a new keymap if none of the shipped ones are good enough.

3.3 Yi modes

Modes are basically pieces of functionality that the user may enable and disable at any time. This terminology has been borrowed from Emacs and modes in Yi work about the same as they do in Emacs, with some minor differences. In

[2]There are more to the syntax than this, but the details have intentionally been left out of this report, as they are not of great importance for this thesis.

Emacs there are different kinds of modes, such as major modes, minor modes and global modes. The difference is not important for this report, because Yi only has one kind, which is simply called "mode" (but this might be changed in the future).

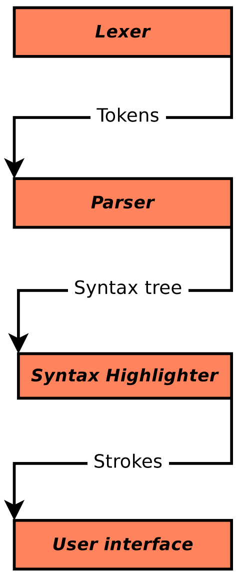
Similar as they may seem, there is a significant difference between developing a mode for Yi and one for Emacs (apart from the fact that Yi uses Haskell and Emacs uses Lisp). In Yi, the programmer of any non-trivial mode will write a lexer, a parser and a syntax highlighter, whereas in Emacs, the majority of all modes only have a syntax highlighter and some indentation functionality. How mode writing for Yi works will be further discussed in section 4.

The different modes in Yi are generally loaded depending on the file extension, meaning that a file with the `cpp` extension would be loaded with the C++ mode and `hs` file with the Haskell mode. The JavaScript mode loads when the user opens a file with the `js` file extension.

4 Writing a mode for Yi

This section is intended as a brief overview on to how modes in Yi are written and used.

4.1 Main components



Most Yi modes consist of three main components: a lexer, a parser and a syntax highlighter. The lexer is generated using Alex and the parser is written using the lazy incremental parser provided by Yi [2]. The syntax highlighter is a function which takes the output from the parser and returns a list of *strokes*. Strokes are data structures which represent regions of characters, also known as *spans*, and styles. Styles include but is not limited to font color, slant and weight. More on this in section 7.

For very simple modes, a parser is not necessary. Instead, the lexer can just map regular expressions to strokes directly, without using the parser as a middle step. One example of such a simple mode is the Whitespace [3] mode shipped with Yi, which highlights the background of any tab characters in green and any space characters in red. Anything else is highlighted as a comment.

For most modes the users will probably want some help with indentation, which is fairly easy to implement for most programming languages using the extensive indentation library provided by Yi.

4.2 Integrating the parts

The syntax of JavaScript is more structured than that of Whitespace, which is why the parser needs to provide a syntax tree to correctly highlight code which looks suspicious. E.g. one has to make sure that any opening curly bracket (`{`) is matched by a closing curly bracket (`}`), that parameters in function declarations are valid identifiers, etc. The parser is essentially a function which takes a list of tokens as its input, provided by the lexer, and returns an abstract syntax tree (AST). For non-trivial modes, the AST is annotated with special tokens which are used to represent any errors that were found.

After the parsing has been done, the AST is passed to the syntax highlighter, which is a function that traverses the tree and outputs a list of strokes depending on its contents.

4.3 Wrapping it up

When the parser and syntax highlighter have been developed, they need to be "glued together" to make it available to the end-user as a mode. In this case, it has been done in the module `Yi.Mode.JavaScript`. This module contains information about the mode, including the name of the mode, when it should be loaded, which parser to use and how to perform syntax highlighting. The module could contain several different modes for the same language, e.g. one which uses a very precise parser and one which only performs syntax highlighting.

For the mode to be available to the rest of the users by default, it should also be imported in the default Yi configuration and added to the *mode table*. Whenever the user opens a new file in Yi, the mode table is searched for the first mode that is appropriate, depending on either the file extension, the file contents, or something else.

5 Yi lexers

The lexers for the programming languages supported in Yi are usually generated using the Alex lexer generator [12]. Therefore, when referring to the *lexer file* in the following sections, we are really referring to the Alex source file. When we write simply *lexers*, we are referring to the generated Haskell module performing the lexing.

5.1 How Alex works

The intention of this subsection is to give the reader a short introduction to how Alex works. If you are already familiar with Alex, you may skip this part.

A lexer file consists of *rules*, *macro definitions* and *actions*. Lexer files usually have the `x` file extensions. When feeding a lexer file to Alex it outputs a corresponding Haskell module using the rules it finds in the lexer file.

5.1.1 Alex macros

There are two different kinds of macros in Alex, namely *character set macros* and *regular expression macros*. Lines defining character set macros always begin with a dollar sign (`$`) whereas lines defining regular expression macros begin with a “commercial at” (`@`).

Character sets are sets of characters that constitute meaningful entities, e.g. lower-case letters (usually defined as `[a-z]`) or numbers (`[0-9]`).

Regular expression macros are used to make up more complex entities, such as valid JavaScript identifiers, and defined using Alex’s own regular expression syntax.

To represent Unicode characters in Alex, you use escape sequences such as `\xC0`. These can be used in regular expressions the same way Latin characters can, as in `[\xD8-\xDE]`.

5.1.2 Rules and actions

A *rule* in Alex is a mapping between a macro and an action.

Actions consist of actual Haskell code which tells the lexer what to do when it finds something matching the corresponding macro. The actions, being Haskell code, must all have the same type. The exception to this is the “do nothing” action, which is represented by a single single semicolon and handled by Alex internally.

5.1.3 Lexer states

One can represent different states of the lexer, using so called *start codes*. Each start code will represent one state in the lexer and are essentially names for each respective state. Each state has its own set of rules which only apply when the lexer is in that state.

E.g., in the JavaScript lexer file in Yi, there are three different states: the initial state, the multiline comment state and the HTML comment state. The initial state is the main state, which looks for reserved words, operators, single-line comments, etc. Whenever the lexer sees `/*`, indicating the start of a multiline comment, it switches to the multiline comment state. In the multiline comment state, everything except for `*/` is regarded as a comment. When it reaches `*/`, it jumps back to the normal state again. The HTML comment state acts similarly to the multiline comment state, but for HTML comments.

5.2 Interaction with Yi

The interaction between the Alex-generated lexer and Yi comes through a so called *wrapper*. Wrappers can be seen as templates for lexers (actual Haskell modules performing the lexing) with holes to be filled in by the lexer file. There are some built-in wrappers available in Alex, but Yi provides its own custom one to accommodate to its special online needs.

Among other things, the Alex wrapper in Yi contains an important container data type called `Tok`, displayed in Figure 9. It is parametrized over its token type, which will be called `Token` in this report. It also contains the length (as in the number of characters) and the position (line and column) of the contained token. Because the type `Tok Token` is so common in Yi, it is useful to give it the type alias `TT`.

```
data Tok t = Tok {
    tokT :: t          -- The contained token.
  , tokLen  :: Size   -- Its length.
  , tokPosn :: Posn   -- Its position in the file.
  }
```

Figure 9: The `Tok` data type in the Alex wrapper in Yi.

To make the parsing much more powerful, as we shall see later, the `Token` type should not be simply `String`, but instead an abstract data type where each of its constructors represent a different part of the syntax. An example definition of `Token` is given in Figure 10.

```
data Token = Reserved String
           | Operator OperatorType
           | Ident String
           | Special String
           | Comment CommentType

data OperatorType = Plus | Minus | Multiply | Divide

data CommentType = SingleComment | MultiLineComment
```

Figure 10: An example `Token` data type and other useful types.

6 The incremental Yi parser

In this section, we will try to give the reader a basic idea about how the lazy incremental parser in Yi works. This is important to be able to understand why some of the design choices in the JavaScript parser were made.

Here the term *parser* will be used to refer to both the parsing library provided by Yi and the parser developed using the library. It should at all times be clear which of the two is being referred to. Should it at any time be unclear, the reader may safely assume that *the parser* refers to the parsing library and *our parser* refers to the parser developed using the library.

6.1 The essential combinators

The parser, provided as a GADT (Generalized Abstract Data Type)[3] named `Parser`, holds a few different combinators for the programmer to express how the parsing should be done. The most important ones from a parser programmer's perspective are the `Appl` combinator for sequencing, `Disj` for choice and `Yuck` for "dislike". The `Yuck` combinator will be explained in further detail in section 6.4. The behavior of the `Appl` and `Disj` combinators should be fairly intuitive; `Appl` is used for sequencing two parsers and `Disj` lets the parser choose between two different parsing paths.

`Parser` is an instance of `Functor`, `Applicative`, `Alternative`, and `Monad`. Its instance of `Applicative` declares `(<*>)` an alias for `Appl` and its instance of of `Alternative` makes `(<|>)` an alias for `Disj`. These combinators in large follow the same pattern presented by Hughes, Swierstra and Duponcheel [8] [19]. See figure Figure 11 for the types of the parser interface.

The parser library also provides an equally important function, `symbol`. `symbol` takes a predicate and returns a parser which will use that predicate to determine whether or not it is successful. See Figure 12 for an example.

```haskell
data Parser s a where
  Appl :: Parser s (b -> a) -> Parser s b -> Parser s a
  Disj :: Parser s a -> Parser s a -> Parser s a
  Yuck :: Parser s a -> Parser s a
  ...

-- Convenient alias
type P s a = Parser s a

symbol :: (s -> Bool) -> Parser s s
```

Figure 11: Part of the parser interface, displaying only the types of `symbol` and the combinators `Appl`, `Disj` and `Yuck`.

6.2 More expressive parsers

As seen in the previous subsection, one can use the `Appl` and `Disj` combinators to easily describe what the parser should expect. But in reality, one cannot use primitive strings to describe every single possible path of execution. The parser has to be put together using broader building blocks to be able to express e.g. "any identifier followed by any operator".

To accomplish this, the lexer has to be modified. Abstracting away the details, the parser will take a list of tokens as input, provided by the lexer, and work its way through them, trying to make sense of the order of them. Using

[3]GADTs are an extension to GHC, the Glasgow Haskell Compiler, which lets the user decide on the return type of each constructor explicitly [16].

```
-- An abstract syntax tree.
data Program = Hi String String
             | Good  String String

-- Parser for the "greeting language".
greeting = Hi  <$> symbol (=="hello") <*> symbol (=="world")
       <|> Bye <$> symbol (=="good")  <*> symbol (=="bye")
```

Figure 12: Outline for a parser which parses ”hello world” or ”good bye”. `<$>` is just another name for `fmap`, which is defined in the instance of `Functor` for `Parser`.

simple strings as tokens is out of the question, so instead a new data type for tokens will be used to be able to use pattern matching in the parser.

Assume that we have the token data type `Token` as shown in Figure 13. This definition is conveniently placed in the lexer file.

```
data Token = Ident String | Oper String
```

Figure 13: A token data type. `Ident` represents identifiers and `Oper` operators. The `String` in each constructor represents the the actual text which makes up each token, e.g. `Ident "hello"` represents the identifier `hello` and `Oper "+"` the addition operator.

These constructors are used to represent a fantasy mini-language consisting solely of identifiers and operators. The lexer has been set up to put any identifiers it finds in the `Ident` constructor and operators in `Oper`. Anything else is ignored in this example.

Having made this change, the parser will no longer take `[String]` as its input, but rather `[Token]`, giving it much more power, specifically the ability to do pattern matching on the constructors, which represent the “meaning” of each token. See Figure 14 for an example using this new token data type.

6.3 Defining an abstract syntax tree

So far, we have ignored any definition of an AST and intentionally kept it out of the discussion as some abstract entity. We did this to put emphasis on deeper concepts and the behavior of the parsing library.

Most readers who have developed a parser in Haskell before are probably already familiar with the typical way to define an AST data type. An example can be seen in Figure 15. The AST allows creation of trees composed of an identifier followed by another tree or just a single operator.[4] The `Token`s in

[4]This is not a tree, as it has no branches, but it’s just for illustration.

```
-- Parser for identifiers.
pId = symbol (\t -> case t of
                       Ident _ -> True
                       _       -> False)

-- Parser for operators.
pOp = symbol (\t -> case t of
                      Oper _ -> True
                      _      -> False)

-- Parser for expressions with an infix operator.
expr = pId <*> pOp <*> pId
```

Figure 14: An example parser, `expr`, using the `Token` data type from Figure 13. This parser accepts any two identifiers with an infix operator. `pId` and `pOp` are parsers for identifiers and operators, respectively.

```
data Program = PIdent Token Program
             | POper  Token
             | PErr   Token
```

Figure 15: A data type for an AST. It can be constructed as either an identifier followed by another program or an operator. It also has a constructor for representing errors of any kind.

the first two constructors represent the identifier token and the operator token, respectively. The `Token` in `PErr` will be discussed in further detail in section 6.7.

Using this AST definition, the previous example from Figure 14 is modified to build an AST, as shown in Figure 16. Note the use of `<$>` which is another name for `fmap`, which is defined in the `Functor` instance of `Parser` as `fmap f = ( pure f <*>)`. `pure` in turn comes from the `Applicative` instance.[5]

Assume that the lexer makes identifiers of any alphanumeric sequence of characters, operators out of the standard mathematical operators and ignores anything else. Given the input "`x y z +`" the revised parser will give us the AST shown in Figure 17.

```
program :: P Token Program
program = PIdent <$> pId <*> (POper <$> pOp <|> program)
      <|> PErr   <$> Yuck ...
```

Figure 16: A revised version of the parser from Figure 14, building a real AST.

[5]The definitions will not be explained any further than this, as they would require going into too many details, which is outside the scope of this report.

```
PIdent (Ident "x")
  (PIdent (Ident "y")
    (PIdent (Ident "z")
      (POper (Oper "+"))))
```

Figure 17: The output from the parser from Figure 16 when the lexer is given the input "`x y z +`".

6.4 Parser preference and recovery

Just having sequencing and choice combinators in a parsing library is not very useful on their own. One would really like to have more control over the parsing than that, e.g. to express operator precedence and help the parser understand ambiguous grammars. This is where the `Yuck` combinator comes into play.

6.4.1 Resolving ambiguities

Suppose that we have a parser which at some point gets to choose between two separate paths which are identical, but mean different things. Recall that JavaScript 1.8 defines a new function declaration syntax (see Figure 5 in section 2.6). This new syntax introduces some ambiguity, e.g. in the declaration `function helloWorld() { }`, where `helloWorld` could be a function which either returns an empty object or a function which does nothing. It is in fact the latter case, but this needs to be expressed this to the parser somehow, so that it chooses the correct path and does not return an incorrect AST.

The outline for a simple JavaScript parser is shown in Figure 18. Notice how the last part of the `funBody` parser can choose to parse either an object or an empty body when parsing the above example. To persuade the parser into choosing the latter, we can apply the `Yuck` combinator as shown in Figure 19.

6.4.2 Recovering from failure

One of the nicest thing about the Yi parser is its ability to let the programmer express what to do should the parsing fail at some point. This is called *recovering* and to accomplish this, one can use the `recoverWith` combinator. The `recoverWith` combinator is just another name for `Yuck`, which was previously used to resolve ambiguities. In fact, the different names for this combinator comes from its two different uses: for recovering and for "disliking" paths.

In all of the previous examples, we have intentionally ignored the fact that in most programming languages, while still writing code in the editor, the source is *not* syntactically valid. The reason for leaving it out of the discussion until now has been to be able to focus on the other parts of Yi, without dragging in too many concepts at once.

Using what has been explained so far, Yi will simply crash reporting an error telling the user that the parse failed if the she enters something unparsable.

```
-- Entry-point of the parser.
program = many stmt

-- Statement parser.
stmt = FunDecl <$> funDecl
   <|> VarDecl <$> varDecl
   <|> ForLoop <$> forLoop
   <|> ...

-- Function declaration parser.
funDecl = FunDecl <$> res "function" <*> ident
                  <*> params <*> funBody

-- Function body parser.
funBody = FunBody     <$> spc "{" <*> program <*> spc "}"
      <|> FunBodyExpr <$> expr

-- Expression parser.
expr = ExprNum <$> number
   <|> ExprObj <$> spc "{" <*> keyValues <*> spc "}"
   <|> ExprStr <$> string
   <|> ...
```

Figure 18: The outline for a parser for a subset of JavaScript. It is ambiguous in the last pass of `funBody`, which could parse either a block with no statements or an empty object, since their syntax is exactly the same. The parser `res` is for reserved words and `spc` is for "special characters".

```
funBody = FunBody     <$> spc "{" <*> many stmt <*> spc "}"
      <|> FunBodyExpr <$> Yuck expr
```

Figure 19: A better definition of `funBody`. Notice the use of `Yuck` to tell the parser that it should dislike ("yuck") that path, urging it to choose the first, correct one.

Naturally, this is unacceptable in a development environment, so one has to let the parser make a choice to fail on its own, falling back to a *recovery parser* should no other path be accepted.

See Figure 20 for an example parser where we have to recover. This parser will only accept programs beginning with an identifier and will crash if it comes across anything else. A better version can be seen in Figure 21 which will recover gracefully if the first token in the input is not a valid identifier. The only thing left now is to figure out what do recover *with*, which will be discussed in section 6.7.

```
program = pId <*> (pOp <|> program)
```

Figure 20: A parser which expects an identifier followed by either an operator or another program.

```
program = pId <*> (pOp <|> program)
      <|> recoverWith ...
```

Figure 21: The same parser as in Figure 20 but will fail gracefully, utilizing the `recoverWith` combinator.

6.5 Sequencing without failing

As can be read in the previous subsection, we can "fail gracefully" using `recoverWith` , to choose what parser to use if the correct parser. However, it was only defined what would happen should the first "subparser" in the sequence fail. In reality, one has to take care of each separate subparser separately.

See Figure 22 for an example and note how it only recovers should the first `expr` parser fail. If `expr` succeeds, the parser expects the `oper` parser and the second `expr` parser to succeed, but will not attempt to recover on failure.

```
binExpr = BinExpr    <$> expr <*> oper <*> expr
      <|> BinExprErr <$> recoverWith ...
```

Figure 22: Only recovering on the first subparser.

Since this situation is so common, the code would quickly become unreadable (consider longer parser sequences) if one inserted useful recoveries in each parser sequence. Instead, we wrote a few small useful functions which do the "standard recovery" in case the desired parser fails. These functions are what we call *please parsers*, e.g. `plzTok` or `plzExpr`. The names of these functions should be interpreted as "please, a token" and "please, an expression", respectively. See Figure 23 for the revised version of the earlier, unsafe, expression parser.

```
binExpr = BinExpr    <$> expr <*> plzOper <*> plzExpr
      <|> BinExprErr <$> recoverWith ...
```

Figure 23: A safer version of the parser from Figure 22.

6.6 Finding the best path

In section 6.4.2 we saw how we can apply the `Yuck` combinator to make the parser dislike a certain path. This section explains in more detail how the parser decides which path it likes the best.

The parser interface implements a rather simple heuristic for determining the best of the possible paths. The basic idea is to count the number of `Yuck`s along all possible paths that it has to choose from and pick the one with the lowest count. Any path which leads to failure is discarded and only used if there are no other options.

Since the number of different paths grow quickly in most useful parsers, the program would be rendered unresponsive if the parser went "all the way" down each different path to determine the optimal one. To set a limit to the searched depth, the Yi parser defines a function `dislikeThreshold` which states how many `Yuck`s on the same path it should accept before discarding that path.

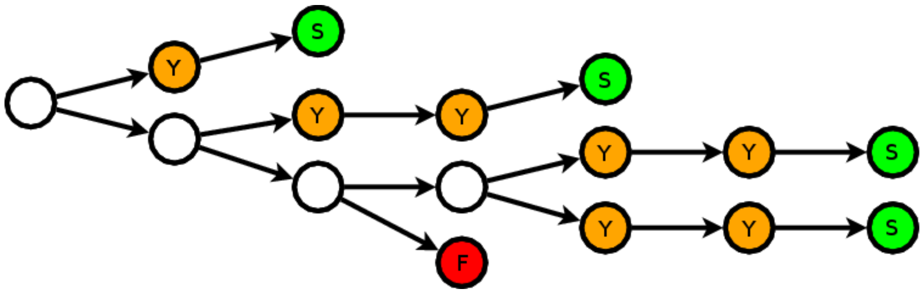


Figure 24: The example of what the different parsing paths may look like in a very short parser. White represents sequenced parsers, orange represents `Yuck`, red represents parse failure and green represents success.

See Figure 24 for an example of what the paths may look like for a simple parser. There are five paths in total, of which one fails (the red node) and four succeed (the green nodes). None of the paths will reach `dislikeThreshold`, which is set to five `Yuck`s at the time of writing.

Using the heuristic mentioned above, the parser will pick the path leading to the leftmost green node, even though it looks bad at first. This is because there are two `Yuck`s at all of the other paths, except for the path which fails. The failing path is given the lowest priority of all of the paths, since no other path fails altogether.

6.7 Annotating the AST with errors

The `recoverWith` combinator accepts any parser as its only argument, which allows us to express exactly what we want the parser to do when it needs to recover. In this thesis, we have found that the two most useful strategies for recovery are what we call *accept-anything recovery* and the *insertion recovery*.

The idea of the accept-anything recovery strategy is simple: just accept any one token and use that instead of whatever we really wanted. The insertion recovery strategy is just as simple and basically creates a new token which is used instead of what it found. In the JavaScript parser, we make extensive use of both.

The accept-anything strategy is easily implemented, using the functions we described in section 6.1, as `recoverWith (symbol (const True))`. This parser will accept any token and so we have avoided a crash. However, there is a problem with this approach: if there are no more tokens to consume, the parser will *still* crash. To circumvent this problem, we use the insertion recovery strategy on top of the accept-anything strategy.

See Figure 25 for a parser using a combination of these strategies, preferring to consume the next token, but if that fails too, inserts an error token.

```
-- A convenience alias.
rec = recoverWith

expr = ExprNum <$> number
   <|> ExprObj <$> spc "{" <*> keyValues <*> spc "}"
   <|> ExprStr <$> string
   <|> ExprErr <$> (rec (symbol (const True))
                 <|> rec (rec (pure errorToken)))
```

Figure 25: A parser for expressions which implements both the accept-anything and the insertion recovery strategy. As before, `spc` is a parser for special characters.

7 Syntax highlighting

Syntax highlighting can be done in a few different ways in Yi, we will describe the most common way at the time of writing.

As we explained before, the syntax highlighter will take the AST generated by the parser as input and return a list of strokes. Recall that strokes represent spans of characters and styles. Yi has a special data type called `Span` for representing spans in the buffer and its definition can be seen in Figure 26 together with the definition of `Stroke`.

Notice that `Stroke` is the same type as `Span StyleName`, i.e. a span of "styling". In turn, `StyleName` is a predefined style, set in the Yi theme. Examples of styles are `defaultStyle`, `errorStyle`, `builtinStyle`, etc. This means that the actual colors and typefaces that are presented in the user interface depend on the user's current theme, giving the user the control to decide e.g. how errors should be displayed.

The leaves of the AST will in this example have the type `Tok Token` (`TT`), the most common construction in Yi modes today, which holds a token and

```
data Span a = Span { spanBegin :: !Point
                   , spanContents :: !a
                   , spanEnd :: !Point
                   }

type Stroke = Span StyleName
```

Figure 26: The definition of the data type `Span` in the source code for Yi, parametrized over the type of its contents, and the definition of Stroke.

information about its position and size. The Alex wrapper in Yi provides a function `tokToSpan :: Tok t -> Span t` which, as its type suggests, returns a `Span` representing the given `Tok`.

The AST is not the only argument passed to the syntax highlighter. The other arguments are the location of the cursor (also known as *point*), the position of the first visible part of the edited file and the position of the last visible part of the file. These arguments can be used to perform e.g. parentheses matching, i.e. highlighting of the opening parenthesis if point is currently at a closing parenthesis and vice versa[6].

The resulting `[Stroke]` is usually created using difference lists, a well-known technique giving us better performance than ordinary lists.

Note that the `Stroke` data type contains no information about the characters it is styling, it only cares about the location of them.

8 The JavaScript mode

This section will describe the state of the JavaScript mode at the time of writing. We will list the known drawbacks and explain how we solved some problems that we encountered.

8.1 The JavaScript lexer

The JavaScript lexer is, as we described in section 5, generated using Alex. For conciseness, when we write *lexer* in this section, we will always be referring to the Alex file, as opposed to what we did before.

This section will assume that the reader is already somewhat familiar with how Alex works. We will only be touching the surface of how the generated lexer interacts with Yi, as it would require much more discussion than fits in this report.

Writing the JavaScript lexer was quite straight-forward. We have defined many different character set and regular expression macros to represent the different tokens in JavaScript. A few simple ones are shown in Figure 27. As can be seen, the `$large` and `$small` define a few extra ranges of characters that

[6]This has been done in one of the many Haskell modes for Yi.

are outside of the ASCII character set, for some Unicode compatibility, although they are nowhere near complete.

```
@ops = "+" | "-" | "*" | "/" | ...
$large = [A-Z \xc0-\xd6 \xd8-\xde]
$small = [a-z \xdf-\xf6 \xf8-\xff]
$special = [\(\)\,\;\[\]\{\}\:\?]
```

Figure 27: An excerpt from the JavaScript lexer. `@ops` is a regular expression macro matching the different operators in JavaScript. The other three lines are character set macros defining which characters that are upper-case, lower-case and have some special meaning, respectively.

The internal state in the lexer is simply an `Int` that will always be one of 0, 1 and −1. State 0 denotes the initial state, i.e. the entry point of the lexer; state −1 denotes the multiline comment state for comments in `/*` and `*/`; state 1 denotes the HTML comment state.

In the lexer, we also define the `Token` data type, as shown in Figure 28. In the lexer rules, we will be using the different constructors of this data type to make tokens out of the strings matching each separate lexer macro.

```
data Token = Res !Reserved
           | Number !String
           | Op !Operator
           | Special !Char
           | Number !String
           | ValidName !String
           | Comment !CommentType
```

Figure 28: Part of the `Token` data type, separating reserved words, numbers, operators and special characters among other things.

Using our macros, we define the initial state in the lexer and set up which rules that should apply in it, as can be seen in Figure 29. Notice how we discard any whitespace characters, but take care of anything else.

`cs` is an alias for `actionStringConst`; a higher-order function that takes a `String -> token` function and returns an `Action`. The `String` in the passed function is the string that triggered the rule. `m` is an alias for `actionAndModify`, which discards the string entirely and modifies the internal state of the lexer.

The `opToOp` and `resToRes` functions are custom functions which takes a string and returns the right constructor depending on that string. Parts of these functions can be seen in Figure 30. The reason that we don't want to use strings is that strings are generally slower to compare.

The multi-line comment state can be seen in Figure 31. Since nested comments are forbidden in JavaScript, we can simply interpret any occurrence of `/*`

```
$whitechar+   ;
$special      { cs $ (Special . head) }
@number       { cs $ Number }
@ops          { cs $ (Op . opToOp) }
@reservedid   { cs $ (Res . resToRes) }
@varid        { cs $ ValidName }
"/*"          { m (const comment) $ Comment Start }
```

Figure 29: Part of the initial lexer state.

```
opToOp :: String -> Operator
opToOp "+" = Add'
opToOp "-" = Subtract'
...

resToRes :: String -> Reserved
resToRes "break" = Break'
resToRes "case"  = Case'
...
```

Figure 30: Parts of the `opToOp` and `resToRes` functions.

as normal comment text. Again, we discard any whitespace characters, just like we did in the entry-point lexer state. The HTML comment state looks much similar to this state, which is why we have chosen to exclude it here.

```
$whitechar+   ;
"*/"          { m (const 0) $ Comment End }
[^\*]+        { c $ Comment Text }
.             { c $ Comment Text }
```

Figure 31: The multi-line comment state in the JavaScript lexer at the time of writing. There are still a few bugs to sort out, but it works somewhat nicely. Notice how we go back to state 0 (the entry-point state) when we reach the end of a comment.

One last important thing that we define in the lexer is `tokenToStyle`. It is a simple function which takes a `Token` and returns the appropriate style to use for that token. So we pattern-match on the constructor of the input `Token` to return the right style, e.g. if it is a number, we return `numberStyle`, etc. This function is essential to perform syntax highlighting.

8.2 The JavaScript parser

The goal we had with the parser was that it should be precise enough to detect as many syntax errors as possible without becoming unreasonably slow. Given the somewhat loose, but not simple, syntax of JavaScript, this was a hard task to accomplish so we had to make some simplifications, as we shall see later in this section.

8.2.1 Representing errors

In Figure 25, we made use of something called `errorToken`, but intentionally kept it abstract to explain it here. Every non-trivial mode has some notion of what an error token is, which will be used when recovering to denote an error in the AST. The error token should have the type `TT` to typecheck with the other leaves in the AST. In the JavaScript mode, `errorToken` is defined as `Special '!'` at position 0 in the file and consists of no characters.

`Special` is the constructor for special characters, such as brackets, backslashes, commas, semicolons, basically any character which has a special meaning in JavaScript, but is not an operator. Since `!` is not a special character, but an operator, using `Special '!'` will not be ambiguous to the syntax highlighter. For convenience, we have also written a predicate named `isError` which takes a `TT` and returns `True` if it represents an error and `False` otherwise.

8.2.2 Handling comments

The syntax of JavaScript allows comments between any two tokens, which became a problematic part in writing the parser. The reason for having to handle comments at all is that we have to syntax highlight them, so we can't just discard them.

The ideal way to handle comments would be to modify the lexer to separate comments from the rest of the AST and then pass a tuple such as `(Tree, Comments)` to the parser. That way, the parser wouldn't have to deal with comments at all and the comments would simply be passed along to the syntax highlighter. Unfortunately, the Alex wrapper currently doesn't support this and since rewriting the wrapper wasn't the goal with this project, we let that be and looked for other solutions.

Another way would be to make the default style in the JavaScript mode be whatever the user has chosen as the comment style. This way, whenever the lexer comes across a comment, it can just discard it, in the same way that it discards whitespace, and all comments would still be sort of "implicitly" highlighted by the default style. This is not a very robust solution for many reasons, e.g. consider what would happen if the user has chosen a comment style with a background color; then any whitespaces would also have a background color, which would be unacceptable.

In the end, testing has shown that the best way to handle comments today is a rather naïve way: in every part of the code, program the parsers to expect an optional comment between any two tokens. This strategy has been implemented in Anders Karlsson's Haskell mode for Yi [10].

However, considering that we want to keep the AST easily traversable, we decided on making comments only valid where statements are valid. Having the AST cluttered with comments would make it cumbersome to extract the useful information from it. That is a trade-off we were not willing to make, so we believe that allowing comments only at the statement level is a reasonable drawback. See Figure 32 and compare it to Figure 33 to see the difference in the data type for function declarations when allowing comments only at a statement level and anywhere in the code.

```
data Stmt = FunDecl TT TT TT [TT] TT Body
          | Comment TT
          | ...
```

Figure 32: The beginning of the data type for statements, allowing comments only where statements are allowed. Each `TT` represents a token that is part of the statement. In `FunDecl`, they represent the reserved word **`function`**, the name of the function, the left parenthesis, the parameters of the function, the right parenthesis and the function body.

```
-- Represents comments.
type Cm = Maybe Stmt

data Stmt = FunDecl TT Cm TT Cm TT Cm [TT] Cm TT Cm Body
          | Comment TT
          | ...
```

Figure 33: The same data type as the one shown in Figure 32 but allowing comments anywhere in the source code.

8.2.3 Making the parsing more efficient

Efficient parsing of JavaScript proved quite difficult, since we had to be able to recover many times in some parts of the parser.

Consider a naïve function declaration parser like the one in Figure 34. Note that we are using recovery at the name of the function, the brackets around the parameters, the opening curly bracket and the closing curly bracket. The details for the `parameter` parser has left out, but naturally we will have to do some recovering in that as well. With each sequenced recovery, the number of parsing paths to choose from can grow exponentially.

To solve this, what we want is really to find the errors as early as possible. In order to do that, we split the function declaration parser into "edible" parts.

```
data Statement t = FunDecl t t t [t] t [Statement t] t
                 | ...

-- Parser for function declarations.
funDecl = FunDecl <$> res Function <*> plzTok name
                  <*> plzSpc '(' <*> many parameter
                  <*> plzSpc ')' <*> plzSpc '{'
                  <*> many statement <*> plzSpc '}'
```

Figure 34: A naïve parser for function declarations.

We make the parameters a separate entity of the function declaration and the same with the function body. The result of this is shown in Figure 35.

```
data Statement t = FunDecl t t (Parameters t) (FunBody t)
                 | ...

data Parameters t = Parameters t [t] t
                  | ParErr t

data FunBody t = FunBody t [Statement t] t
               | FunErr t

-- Parser for function declarations.
funDecl = FunDecl <$> res Function <*> plzTok name
                  <*> parameters <*> funBody

-- Parser for function bodies.
funBody = FunBody <$> spc '{' <*> many statement
                  <*> plzSpc '}'
      <|> FunErr  <$> recoverWith ...

-- Parser for parameters.
parameters = Parameters <$> spc '('
                        <*> plzTok name `sepBy` spc ','
                        <*> plzSpc ')'
         <|> ParErr <$> recoverWith ...
```

Figure 35: A parser for the same thing as in Figure 34, but divided into a few different smaller parsers. This prevents a potential exponential explosion of possible paths for the parser to choose from.

Using this little trick, it should be apparent that we can improve the parser performance greatly. We have applied this technique in some other places in the parser as well, e.g. in the for-loop parser and the variable declaration parser. However, we tried to keep the applications of this trick to a minimum, to keep the AST somewhat legible and easily traversable.

8.2.4 Verifying the semantics of the code

Part of the initial goal for this project was to not only do incremental syntax checking, but also to check some of the semantics of the source code. Considering the massive performance problems the JavaScript parser suffered from to begin with, we never tried to perform online verification, as having that would only put more unnecessary pressure on the editor, possibly resulting in sluggish and unresponsive behavior.

Instead, we decided to write a separate verifier, which is called on-demand by the user. The verifier takes the syntax tree from the parser, forcing it to parse the entire source file, and does some basic error-checking and reports the results to the user. It was written in about a day and at the time of writing only looks for multiple function declarations and unreachable code.

The verifier is called using the keyboard sequence `C-c C-l` (Emacs notation for holding Control and then pressing C and then L). When the verifier has verified the source code, it opens up a new window in Yi into which it prints the results of the verification, as can be seen in Figure 36.

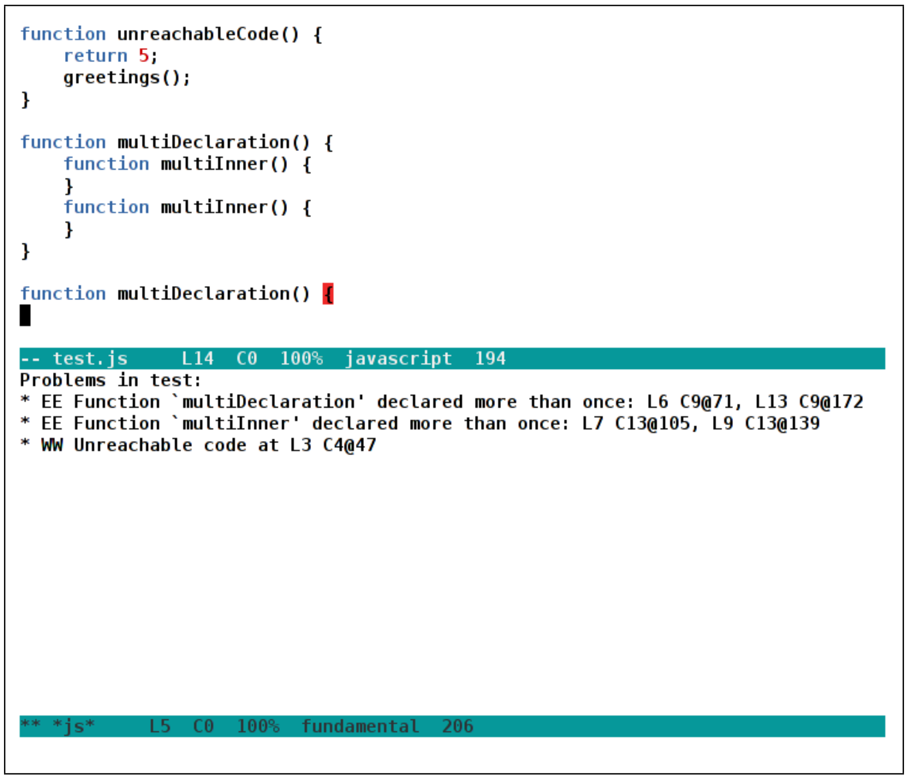
8.3 Indentation behavior

One of the most important thing in a development environment is that it should have good support for indentation. In Emacs, most modes automatically indent the current line correctly according to the programming language that is being used when the user hits the tab key anywhere on that line. This behavior has been adopted to Yi and the JavaScript mode as well.

The functionality is provided in the form of different functions in Yi, working in the `BufferM` monad, which is a special monad for handling buffers in Yi. Due to lack of time, we could not implement indentation which is 100 percent correct, but instead let the user "switch" between a few different indentation levels. With each press of the tab key, the indentation cycles one step in the list of selected indentation levels. The implemented indentation works roughly as follows:

1. Get the user's preferred indentation width, I.
2. Get the current indentation of the first non-empty line preceding the current one, B.
3. If point is at the beginning of the source code, switch between no indentation and I.
4. Otherwise, switch between B, $B + I$ and $B - I$.

We also implemented a function called `newlineAndIndentB`, which inserts a new line in the editor at point and indents it to the level of indentation that the previous line had. We bound this function to the Return key, to relieve the user from having to press tab for each new line she makes.



```
function unreachableCode() {
    return 5;
    greetings();
}

function multiDeclaration() {
    function multiInner() {
    }
    function multiInner() {
    }
}

function multiDeclaration() {

-- test.js     L14  C0  100%  javascript  194
Problems in test:
* EE Function `multiDeclaration' declared more than once: L6 C9@71, L13 C9@172
* EE Function `multiInner' declared more than once: L7 C13@105, L9 C13@139
* WW Unreachable code at L3 C4@47

** *js*     L5  C0  100%  fundamental  206
```

Figure 36: A Yi session with a demo JavaScript source file. Note how the incremental parser has highlighted the opening curly bracket for the last function declaration, since it is missing a closing bracket. The bottom window displays the output from the verifier, which has found multiple function declarations, both on the top level of the program and duplicate inner function declarations. It has also found that the call to `greeting` will never be made and thus warns the user about that.

8.4 Syntax highlighting

To be useful at all, the parser has to be able to not only find errors, but somehow indicate where they occur. The indicator in Yi is usually provided by the syntax highlighter which sets an "error style" (red background in the default Yi theme) on the regions at which the parser failed. In other words, we have to be able to narrow down the red background to regions of code where the error was located.

In the JavaScript parser, we have a few different data types which make up the AST. The top-level data type is `[Statement]`, with `Statement` having constructors for function declarations, variable declarations, loops, etc. Expressions also have their own data type, `Expr`, and so on. Each of these different data types have a constructor representing error, except for `Statement`, which doesn't need one, as it falls back on the recovery in the expression parser. This is possible because expressions are valid statements as well, so if the user enters an unrecognized statement, it will try using the expression parser. If the expression parser also fails, the expression recovery parser is used.

Since we have these different data types which can all be syntax highlighted, we wrote a simple type class named `Strokable` (named after the `Stroke` type) which can be seen in Figure 37. It has a single function `toStrokes`, which returns `Endo [Stroke]`, since we are using difference lists for improved performance, as briefly mentioned in section 7.

```
class Strokable a where
    toStrokes :: a -> Endo [Stroke]
```

Figure 37: The `Strokable` type class.

To define which constructor that represents failure in each data type, we made another type class, named `Failable`. It has a predicate called `hasFailed` which checks if the input represents failure (syntax error). This predicate is used extensively in the definitions of `toStrokes` for our data types.

For an example instance of `Strokable`, see Figure 38. The definitions of `errorStroker` and `normalStroker` have been left out, as they are not needed to understand the idea.

```
data Stmt t = FunDecl t t (Parameters t) (Block t) | ...

instance Strokable (Stmt TT) where
  toStrokes (FunDecl fun name ps blk) =
    let (<>) = mappend
        s = if hasFailed blk
              then errorStroker
              else failStroker name
    in s fun <> s name <> toStrokes ps <> toStrokes blk

failStroker :: TT -> TT -> Endo [Stroke]
failStroker x | isError x = errorStroker
              | otherwise = normalStroker

isError :: TT -> Bool
isError (Special '!') = True
isError _             = False
```

Figure 38: Part of the instance of `Strokable` for `Stmt`. It determines a stroker to use for the reserved word `function` and the identifier representing the function name by seeing if the block of the function declaration has failed. If so, it strokes the two tokens using the error style. Otherwise it uses the errorStroker only if the identifier token has failed.

9 Conclusions

This project has shown that it is possible to write precise and somewhat efficient incremental parsers using the incremental parsing library in Yi, although one has to make trade-offs between performance and preciseness.

9.1 Results

The state of the JavaScript mode for Yi is still unstable at the time of writing, but will will not crash on any input as far as we can tell.

The parser performance is acceptable for small JavaScript programs and catches a majority of the possible syntax errors that can be made. Investigations have shown that the inline array parser is one of the slowest parsers in the mode and can probably be greatly improved.

The for-loop parser currently does not allow variable declarations in the initialization field, but assignments are acceptable. Solving this problem should not take too much effort, but requires some careful refactoring.

Comments are only allowed where statements are. While it should not be a problem to allow comments between any two tokens, the code for the parsing would become cluttered with optional comment parsers and is better left to do when the mode reaches a more stable state. We are also aiming to make the tree easily traversed by the verifier, which would not be as easy with comments between tokens.

The JavaScript mode consists of 1113 lines of code at the time of writing. Just about half of it lies in the parser module, roughly 25 percent in the lexer (the Alex source file) and the rest is in the verifier and mode file. At the time of writing, the latest release of Yi is version 0.6.1 and is available on Hackage through `http://hackage.haskell.org/package/yi`. The latest development version is available on the darcs repository at `http://code.haskell.org/yi/`.

9.2 Future work

As we have mentioned earlier, we wish to keep the tree easily traversed by the verifier. A solution which would make the tree easily traversed would be to write a conversion step in the middle which takes the AST provided by the incremental parser and strip it down to make it easily traversed. This step could also remove any comments in between tokens, as they are not needed to verify any of the source code.

If future modes prove the incremental parsing to be too slow for normal use in an editor, one could consider writing an asynchronous parser for Yi. By asynchronous, we mean a parser which is run in the background when the user is not writing anything or moving the cursor. As soon as the user starts writing, if the parser hasn't completed, it would just abort the parsing and start again when the editor is idle again.

Alternative parsing strategies could be investigated, e.g. replacing the "disliking" combinator `Yuck` with a "liking" combinator, defining which paths that should have higher precedence than other paths, instead of defining which paths that should have lower precedence than others.

We have high hopes for the Yi editor and can definitely see it competing with other major editors such as Emacs and Vim for users, even though it may take some time to get there.

References

[1] Jean-Philippe Bernardy. Yi: an editor in Haskell for Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 61–62, Victoria, BC, Canada, 2008. ACM.

[2] Jean-Philippe Bernardy. Lazy functional incremental parsing. In *Proceedings of the second ACM SIGPLAN symposium on Haskell*, Edinburgh, UK, 2009. ACM.

[3] Edwin Brady and Chris Morris. Whitespace. `http://compsoc.dur.ac.uk/whitespace/index.php`, 2004.

[4] Ecma International. *ECMAScript Language Specification*, 3rd edition, December 1999.

[5] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 5th edition, 2006.

[6] Mozilla Foundation. Mozilla Firefox. `http://www.mozilla.com/firefox/`, 2009.

[7] Danny Goodman. *JavaScript Bible.* Hungry Minds, Gold edition, 2001.

[8] R. John M. Hughes and S. Doaitse Swierstra. Polish parsers, step by step. *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, 2003.

[9] Free Software Foundation Inc. GNU Emacs. `http://www.gnu.org/software/emacs/`, 2009.

[10] Anders Karlsson. Robust & Precise Incremental Parsing of Haskell. Master's thesis, Chalmers University of Technology, 2009.

[11] Mozilla Labs. Bespin. `http://bespin.mozilla.com`, 2009. An in-browser JavaScript editor.

[12] Simon Marlow. Alex. `http://www.haskell.org/alex/`, 2007. A lexical analyser generator for Haskell.

[13] Microsoft. Descriptions of Java, JScript, and JavaScript. `http://support.microsoft.com/kb/154585`, 2007.

[14] Microsoft. Windows Internet Explorer. `http://www.microsoft.com/windows/internet-explorer/default.aspx`, 2009.

[15] Bram Moolenaar. Vim. `http://www.vim.org`, 2009. VI Improved, a fork of the older *vi* editor.

[16] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, jul 2004.

[17] Don Stewart. yi: just syntax. `http://www.cse.unsw.edu.au/~dons/yi.html`, 2007. The old homepage for Yi.

[18] Don Stewart and Spencer Janssen. xmonad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, page 119, New York, NY, USA, 2007. ACM.

[19] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. *Second International Summer School on Advanced Functional Programming Techniques*, 2008.

[20] Steve Yegge. js2-mode: a new JavaScript mode for Emacs. `http://steve-yegge.blogspot.com/2008/03/js2-mode-new-javascript-mode-for-emacs.html`, 2008. Blog entry announcing the release of js2-mode.