|  |
| --- |
| University of Nottingham |

A JavaScript Optimiser in Haskell

*Nick Brunt*

|  |
| --- |
|  |

# Abstract

JavaScript is a popular imperative programming language. It is primarily used as a client-side scripting language for websites, to make the user interface more dynamic and user-friendly. As a component of a website, it has to be downloaded to the client machine whenever the website is accessed, much like any images or text. This leads to the issue of optimisation. If the size of the JavaScript file can be reduced without compromising its function, the site will load faster.

In this project the functional programming language Haskell will be used to write an optimiser which compresses a given JavaScript file. We will begin by creating a program that parses JavaScript code and will spend the remaining time reassembling it in a more compact manner. To achieve this, we will discuss and execute several different compression techniques, from the simple (removing whitespace and comments) to the more complex (partial evaluation and identifier shrinking).

|  |
| --- |
|  |

# Table of Figures

[Figure 1. Server-side updating 1](#_Toc324192235)

[Figure 2. Client-side updating 2](#_Toc324192236)

[Figure 3. jQuery 1.7.1 file size comparison 3](#_Toc324192237)

[Figure 4. Context diagram 7](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192238)

[Figure 5. System diagram 7](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192239)

[Figure 6. Overall System Design 15](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192240)

[Figure 7. Process diagram 16](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192241)

[Figure 8. Typical Compiler Structure 17](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192242)

[Figure 9. Data journey: From characters to syntax tree 23](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192243)

[Figure 10. Example of identifier scope and level 32](file:///D:\Documents\Documents\Uni\Dissertation\docs\Dissertation.docx#_Toc324192244)

[Figure 11. Example of errors being caught by Google Chrome's JavaScript Console 49](#_Toc324192245)

[Figure 12. Example of Haddock documentation 52](#_Toc324192246)

[Figure 13. Automatically generated, syntax highlighted code 52](#_Toc324192247)

|  |
| --- |
|  |

# Table of Contents

[Abstract i](#_Toc324192248)

[Table of Figures ii](#_Toc324192249)

[Table of Contents iii](#_Toc324192250)

[Introduction 1](#_Toc324192251)

[Aim 1](#_Toc324192252)

[Background 1](#_Toc324192253)

[Motivation 2](#_Toc324192254)

[Specification 4](#_Toc324192255)

[Specific objectives 4](#_Toc324192256)

[Assumptions 6](#_Toc324192257)

[Data flow diagrams 7](#_Toc324192258)

[Constraints 8](#_Toc324192259)

[Limitations 9](#_Toc324192260)

[Potential solutions 9](#_Toc324192261)

[System Design 13](#_Toc324192262)

[Prototyping 13](#_Toc324192263)

[Approach 13](#_Toc324192264)

[Why it didn’t work 14](#_Toc324192265)

[What was learned and carried forward 15](#_Toc324192266)

[Overall system design 15](#_Toc324192267)

[Modular system structure 16](#_Toc324192268)

[Process diagram 16](#_Toc324192269)

[Module descriptions 16](#_Toc324192270)

[Lexer 17](#_Toc324192271)

[Parser 23](#_Toc324192272)

[Parse tree 23](#_Toc324192273)

[Happy parser 25](#_Toc324192274)

[Code Compressor 26](#_Toc324192275)

[Simple optimisations 27](#_Toc324192276)

[More complex optimisations 31](#_Toc324192277)

[Partial evaluation 36](#_Toc324192278)

[Interface 39](#_Toc324192279)

[Implementation 42](#_Toc324192280)

[Problems encountered and their solutions 42](#_Toc324192281)

[System Testing 44](#_Toc324192282)

[Impact of each compression method 47](#_Toc324192283)

[System Maintenance 51](#_Toc324192284)

[Documentation 51](#_Toc324192285)

[Makefile 53](#_Toc324192286)

[Making modifications 55](#_Toc324192287)

[Naming conventions 55](#_Toc324192288)

[Evaluation 57](#_Toc324192289)

[Critical appraisal 57](#_Toc324192290)

[Comparison of Performance against Objectives 57](#_Toc324192291)

[Possible Extensions 59](#_Toc324192292)

[What was learned about Haskell 60](#_Toc324192293)

[Conclusion 61](#_Toc324192294)

[Bibliography 62](#_Toc324192295)

[Appendix 63](#_Toc324192296)

[Example Input and Output 63](#_Toc324192297)

[Code Listing 64](#_Toc324192298)

|  |
| --- |
| Chapter 1 |

# Introduction

## Aim

To develop a program which optimises a JavaScript file by reducing the number of characters used to express its intentions. The newly compressed JavaScript code must maintain the semantics of the original uncompressed version.

## Background

A website can be made up of several components. At its simplest, a text file containing HyperText Mark-up Language (HTML) is sufficient, perhaps linking to one or two images. When the World Wide Web was first unrolled in 1991, this was the standard definition of a web page. However, this approach was limited and inflexible. Every slight change to the document had to be made server-side which meant that the page had to be completely reloaded every time an update was required. This was a real inconvenience, especially pre-broadband.

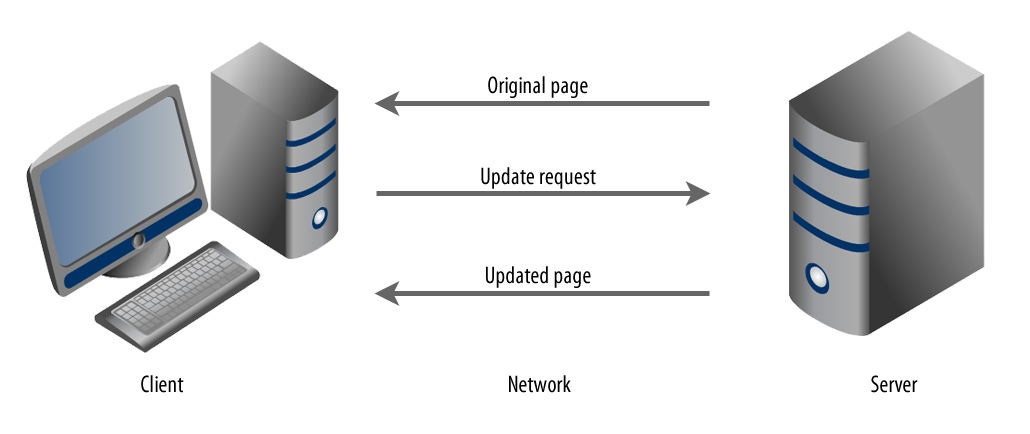


Figure . Server-side updating

It was quickly realised that a client-side scripting language was needed that could perform operations on the web page without having to send a request to the server. In September 1995, Netscape Navigator released LiveScript which did exactly that. LiveScript could either be embedded in the webpage, or included as a separate file. The script could be executed on the client machine and would modify the layout and content of a webpage without reloading it.

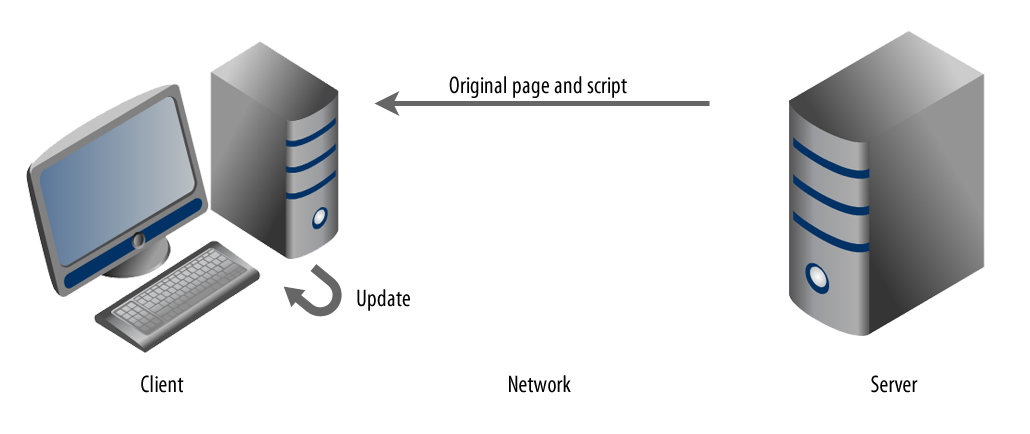


Figure . Client-side updating

Later in 1995, LiveScript was renamed to JavaScript (despite it having nothing to do with Java) and its popularity has increased ever since. It is now almost impossible to find a website that does not utilise JavaScript in some way.

With its popularity comes a problem. It is very tempting to use as much JavaScript as possible when designing a website, so as to enhance the user experience and provide more functionality. However, this can lead to very large JavaScript files which have to be downloaded when a website is accessed. This is where JavaScript compression (or minification as it is sometimes called) comes in. The smaller the JavaScript files are, the faster the webpage loads, so reducing the size of a script file without losing its functionality is clearly a worthwhile and desirable action to take.

If a developer is not using a compression program during the development process, it is tempting to keep comments and identifier names short so as to reduce character usage. With the knowledge that comments will be removed and identifiers shortened, a developer is free to write as much expressive, well laid out code as they like without having to worry about file sizes.

## Motivation

All extensive JavaScript libraries provide a compressed version of themselves. For example, jQuery is a library that provides many features and is included in millions of websites across the Internet. Their latest version (1.7.1) in its uncompressed “developer” format is 229KB whereas the compressed version is 92KB, only 40% of the original size (jQuery.com).

Figure . jQuery 1.7.1 file size comparison

For a single user, this may not make a huge difference, but when you bear in mind how many people surf the web every day, the savings scale up rather quickly. Consider the following:

If jQuery is included in 1 million websites and each of those websites is viewed on average 100 times per day, that’s 100 million requests for the jQuery library. Having compressed the library, 12.8 TB of data does not need to be transferred. This corresponds to considerable savings on both network and server costs.

If the compressor had managed to shave off just *one* more byte of data, a further 100 MB could have been saved. People are spending increasingly more time online and websites are using increasingly more dynamic content. When every character counts, JavaScript optimisers are very necessary.

|  |
| --- |
| Chapter 2 |

# Specification

The primary objective of this project is to build a Haskell program which takes a JavaScript file as input and outputs a new JavaScript file which will behave in the same way when executed as the original. By “behave in the same way” we mean that any input should produce exactly the same output as it would have with the original code. It is possible that the space and time complexity of the code will change, but as long as the results of execution remain identical, this is acceptable. It is expected that some of the methods used will positively affect time and space complexity, however this is not the focus of the paper.

The new file must either be of a smaller size than the original, or exactly the same size, it must not be larger. It may be impossible to compress some files if they are already in their most optimal form. For example, the following code cannot be reduced any further:

alert**(**Hello**,** World**!);*function*** a**(**x**,**y**){*return*** x**\***y**};**

## Specific objectives

*Input, processing and output requirements*

The system must:

1. Accept a JavaScript file as an argument on the command line
2. Translate the contents of that file into a form it can understand and modify without further user interaction
3. Remove the following tokens from the source code
   1. Single-line comments
   2. Multi-line comments
   3. Unnecessary whitespace
   4. Semi-colons before closing braces
4. Shorten identifiers where possible (identifiers being variable and function names)
5. Convert the following expressions to their shorthand equivalents
   1. Ternary conditionals (where possible)
   2. Array declarations
   3. Object declarations
6. Perform partial evaluation on expressions and statements where possible
7. Return the compressed JavaScript to the user along with compression statistics

*Performance requirements*

1. The system must be simple to operate through a command line interface
2. The system must return the results in a timely manner

Specific objectives explained

Some of the specific objectives require more information and are explained in detail here.

S.O. 2 requires that the system translates the input into a form it can understand and modify. The initial file will be no more than a string of characters, which is difficult to deal with. The input must be parsed and converted into a Syntax Tree with meaningful values. This will make it possible to perform complex operations on the data.

S.O. 3d requires the system to remove semi-colons before closing braces. In JavaScript, semi-colons are used to terminate statements and curly braces ({}) are used to wrap blocks of statements. It is unnecessary for the final statement in a block to have a semi-colon. For example, the following two blocks of code are both equally valid:

***for*** **(**i **=** 0**;** i **<** 10**;** i**++)** **{**

count **+=** i**;**

alert**(**count**);**

**}**

***for*** **(**i **=** 0**;** i **<** 10**;** i**++)** **{**

count **+=** i**;**

alert**(**count**)**

**}**

S.O. 4 requires the system to shorten identifiers where possible. Variables that are local to a function can safely be shortened, whereas global variables may be referenced from elsewhere and therefore cannot safely be modified without changing the code’s functionality.

The shorthand expressions mentioned in S.O. 5 are displayed below:

1. Tertiary conditionals

**if** **(**a**)** res **=** 1**;** **else** res **=** 2**;**

would become

res **=** a **?** 1 **:** 2**;**

1. Array declarations

a **=** **new** Array

would become

a **=** **[]**

o = **new** Object

would become

o = **{}**

1. Object declarations

S.O. 6 requires that the system perform partial evaluation on expressions and statements where possible. Partial evaluation will be explained in more detail later on, but here are two examples of how it works.

**Example 1**

a = 25.6 \* 18 + 3 / x;

becomes

a = 463.8/x;

**Example 2**

b **=** "Hello" **+** ' World'**;**

becomes

b **=** "Hello World"**;**

Note that while techniques such as partial evaluation can improve the speed of execution, the main focus of this project is to reduce the code *size* and as such only partial evaluations which result in fewer characters overall will be included in the final output. For example, partially evaluating the sum 4 / 3, will produce something along the lines of 1.3333333… which takes up more space than simply leaving the expression 4 / 3.

## Assumptions

The system is allowed to make certain assumptions.

1. The input is valid JavaScript as specified in the official specification. (ECMA-262 Official Specification)
2. The input is correctly typed.

## Data flow diagrams

The following context diagram shows how the system will interact with the user.

User

User

JavaScript file

Compressed

JavaScript file

Figure . Context diagram

This system diagram shows the inner workings of the proposed system, from first contact with the user up until the end of execution.

User

User

JavaScript file

List of tokens

Parse

Tree

Parse Tree

Parse Tree

Parse

Tree

JavaScript file

Figure . System diagram

## Constraints

Hardware

It is expected that anyone running this software will have a reasonably modern computer and therefore plenty of processing power and memory. However, in the interests of versatility, the system should not be too process intensive. A specification of a 2GHz single core CPU, 512MB RAM and 10MB free hard drive space should be sufficient.

Software

The program should be designed to run on any UNIX or Windows based machine. It should not require a windowing system to operate. It is expected that the program will be implemented as part of the deployment process of a website. That is, it will be invoked automatically by a script. Given this assumption, it is important that it is self-contained and can run in a linear fashion, taking one file as input and returning one file as output, perhaps with some extra analysis information.

Time

The system should be completed and fully documented by 8th May 2012.

User’s knowledge of information technology

It is assumed that the user has at least a basic knowledge of how to operate a computer including proficiency in using command line interfaces. The program must support the addition of a help flag when invoked, which will force it to display instructions on how to operate itself, including a detailed list of all possible options. Below is an example of the ls program being invoked with the help flag:

$ ls --help

Usage: ls [OPTION]... [FILE]...

List information about the FILEs (the current directory by default).

Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.

-a, --all do not ignore entries starting with .

-A, --almost-all do not list implied . and ..

--author with -l, print the author of each file

-b, --escape print C-style escapes for nongraphic characters

--block-size=SIZE scale sizes by SIZE before printing them. E.g.,

`--block-size=M' prints sizes in units of

1,048,576 bytes. See SIZE format below.

-B, --ignore-backups do not list implied entries ending with ~

etc...

## Limitations

Areas which will not be investigated

There are two main aspects to JavaScript optimisation. One is focused on reducing the number of characters used to express a JavaScript program and the other is focused on improving the computational performance of said program. This paper’s primary focus is the former of these two aspects. It will not place any importance on the performance of the code, beyond it having the same overall effect as the original.

Areas considered for future work

Common sub-expression elimination is a process which could potentially reduce the size of a program. It works by detecting expressions which are frequently repeated in the code and abstracting them out into a separate function, variable or macro (however JavaScript does not support macros). The following code shows an example of common sub-expression elimination applied to some simple sums.

a = b \* c + g;

d = b \* c \* d;

The expression b \* c is repeated here, so could potentially be calculated separately and placed in a temporary variable. This way it would not have to be calculated more than once.

tmp = b \* c;

a = tmp + g;

d = tmp \* d;

In this particular example the character count increases when the process is applied. However for longer expressions, it could have a more positive effect.

This method of compression is beyond the scope of the project, but would be a suitable extension.

## Potential solutions

Although it is stated in the title of this paper that Haskell will be used to write this program, it is worth considering other potential solutions. Several options will be investigated before an explanation is given for settling on Haskell.

To develop a system based on the specification above, a programming package with the ability to create executables will be required. This will mean that the program will be runnable on a computer that does not have the programming package installed. It also will be necessary to build a parser in order to translate the input into a form that can be manipulated by the program. With these criteria in mind, several solutions are available.

|  |  |  |
| --- | --- | --- |
| Name of possible solution | Description | Analysis |
| C/C++ | C and C++ are popular imperative programming languages with both high- and low-level features. | Both languages are well supported and have compilers for almost any operating system. |
| Java | Originally developed by Sun Microsystems, Java is an object oriented programming language which derives much of its syntax from C/C++. | Java typically compiles its applications to bytecode which means that they can be run on any Java Virtual Machine (JVM). This means that they will work on any operating system with a JVM installed. |
| Haskell | Haskell is a strictly functional programming language with static typing. | There are many compilers for Haskell, chief among them being the Glasgow Haskell Compiler (GHC) which can compile to executables. |
| Python | Python is a high level, object oriented language. It focuses on code readability and compactness. | Although primarily a scripting language, Python can be packaged into an executable. Its mix of imperative and functional styles makes it a very versatile language. |

Table . Potential programming packages

Why Haskell?

Haskell was chosen for a number of reasons.

1. **Its strictly functional style makes it perfect for developing language parsers.**

“… functional programming can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments. In turn, a functional programming language is one that supports and encourages the functional style.” – Programming in Haskell (Hutton, 2007, p. 2)

The main difference between functional and the more standard imperative languages (like C, Java, Pascal etc.) is how effects are dealt with. A function in Haskell has a static type signature which defines exactly what the input and output of the function will be. No other output will be valid. A function in an imperative language does not act exactly like this. Although a type signature may define the basic inputs and output, the function is free to have other effects while executing. For example, the following C function will take an integer as input and output another integer; however it will also increment a global variable and print a message to the command line:

***int*** ***double*(*int*** input**)** **{**

myGlobal**++;**

printf**(**“Doubling %d”, input);

***return*** input **\*** 2**;**

**}**

In a purely functional language, this sort of function is not possible (without Monads which we’ll discuss later). This has drastic performance consequences when it comes to compiling the program. A purely functional language can be treated like a strict mathematical calculation and can thus be hugely optimised. Time and space complexity can be greatly improved so the resulting program runs faster.

The type system in Haskell also reduces the number of run-time bugs in the program by flagging them at compile time. This is much more difficult to detect in an imperative language as the type signature of the function does not reveal everything about how the function will behave.

For this particular project, it is important that no details of the original program are forgotten when it is parsed. The strict type system will help to ensure that each parsing function does exactly what is intended.

1. **There are many lexer- and parser-generators available which compile to Haskell source code.**

Based on Haskell’s suitability for lexing and parsing, several parser generators and parser combinators have been produced.

A parser generator is a program which can be fed a language grammar and will produce a program which parses that language. A language grammar is a set of formal rules which declare how a programming language can be constructed. Given the language alphabet, the grammar specifies all possible valid ways of constructing instructions using words from the language.

In this project we will use Alex[[1]](#footnote-1) to generate the lexer, and Happy[[2]](#footnote-2) to generate the parser for JavaScript. Both these programs produce Haskell modules which will tie in very easily with the rest of the program.

A parser combinator is a higher-order function[[3]](#footnote-3) which accepts multiple parsers as input and returns a single parser as output. An example of this would be Parsec[[4]](#footnote-4). We will not go into detail about parser combinators as they are generally more useful when you do not have a grammar to work from.

1. **The Glasgow Haskell Compiler (GHC) can compile Haskell code into executables.**

Given the --make flag and a Haskell source file, GHC follows the dependencies and builds an executable file (exe) from any multi-module Haskell program. For example, a project may contain three files:

Main.hs

with the following dependencies:

Source1.hs

Source2.hs

Using the following command, GHC will produce Main.exe:

ghc --make Main.hs –o Main.exe

1. **The resulting executables do not require Haskell to be installed on the system in order to run.**

A standalone executable file has no dependencies besides the operating system it was compiled for. In many cases, executables are cross-platform compatible. As this project does not use any complex windowing controls, it is likely that the resulting exe will work on Windows, UNIX, Linux and Mac with no modifications.

|  |
| --- |
| Chapter 3 |

# System Design

## Prototyping

When beginning any project, it is a good idea to create prototypes to test and explore different approaches to solving the problem at hand. In this case, the initial approach proved unscalable and was not robust enough. During this section we will explore what was attempted and the reasons why it ultimately failed.

### Approach

At first, the system was approached from a very targeted perspective. Each section of code which had to be optimised was matched by a regular expression and then replaced by the appropriate compressed code. For example, when removing single line comments, the following regular expression (regex) was used to find each comment which in turn was replaced with nothing:

//[^\n]\*

This regex looked for two forward slashes followed by a stream of characters which were not newlines. In effect, it matched everything following the slashes, up until the end of the line.

The same approach was used for all the other character groups which needed to be detected. Here is a small selection:

|  |  |  |
| --- | --- | --- |
| Character group | Regular Expression | Example match |
| Multi-line comments | /\\\*[^\*]\*\\\*+([^/][^\*]\*\\\*+)\*/ | /\* test \*/ |
| Conditional comments | /\\\*@[^\*]\*\\\*+([^/][^\*]\*(@\\\*)+)\*/ | /\*@ test \*/ |
| Single quote strings | '([^'\\\\]|\\\\.)\*' | ‘test’ |
| Double quote strings | \"([^\"\\\\]|\\\\.)\*\" | “test” |
| Spaces | \\s+ |  |
| JavaScript regular expression | [^\*/]/(\\\\[/\\\\]|[^\*/])(\\\\.|[^/\n\\\\])\*/[gim]\* | /pattern/g |

Table . Regular Expression examples

There are two main types of regular expression: POSIX and PERL. To begin with, POSIX was used, but it quickly became apparent that the shorthand capabilities of PERL regular expressions would make the job a lot easier.

The full source listing of the prototype can be found in the appendices, page 130.

### Why it didn’t work

Although a fully working prototype was developed which could remove comments and whitespace, it did not scale well. On around 100 lines of code, the program executed in a couple of seconds, but on 1000 lines, it took several minutes. The reason for this inefficiency was the way in which character groups were matched. Take the above example of a single line comment. The regular expression simply matches two forward slashes followed by a list of any characters up until a newline. However, what happens if the forward slashes are in a literal string?

***var*** a **=** "Hello // World"**;**

This is perfectly valid JavaScript, but the regular expression will wrongly detect a single line comment where none exists. To get around this problem, each literal string had to be detected in advance and any comments found which actually started in a string had to be ignored.

When we are dealing only with things like single- and multi-line comments, this does not represent a big problem. However, when it comes to matching whitespace, the issue is compounded. The regular expression which matches spaces, tabs and newlines will match every space in the program. To stop it removing spaces which are actually needed (like those in strings or between expressions); a huge list of every necessary space had to be compiled in order to filter out the correct ones.

spaceExceptions :: Code -> Matches

spaceExceptions cs = getMatches dString cs ++

getMatches sString cs ++

getMatches cComments cs ++

getMatches jsRegex cs ++

getMatches plusplus cs ++

getMatches wordBound cs ++

getMatches dollarVar cs ++

getMatches startDollar cs ++

getMatches endDollar cs ++

getMatches spaceShorthand cs

This generates a very long list, even with a relatively short input program. The getMatches function uses the regular expression fed in to match every single corresponding character group in the program. Once the list of exceptions was accumulated, the original list of all spaces was filtered and any remaining spaces were removed from the input code.

The space and time complexities for this approach were huge. Although it could have been improved upon, for instance by using the cons operator instead of append, it was clear that the system was being designed from the wrong perspective. There was already an abundance of issues with simply removing whitespace and comments – and that was supposed to be the easy part.

### What was learned and carried forward

Although the entire prototype was effectively discarded, several lessons were learned. It became obvious that a fully featured parser was necessary to translate the input program into something that could be sensibly manipulated and controlled. A parser only scans the input code once and creates a Syntax Tree, which can then be recompiled in whatever way is desired. This vastly reduces the amount of processing necessary and should reduce the execution time drastically, especially for large inputs.

Some things were salvaged from the process and several methods designed for the prototype were carried forward to the final solution. These included functions for reading and writing files to and from disk, reading arguments from the command line, and calculating compression ratios.

## Overall system design

The proposed new system has been named “JSHOP” which stands for JavaScript Haskell Optimiser. The system will be referred to as such from this point forward.

The following diagram shows the overall inputs, outputs and storage mediums of the new system.

Figure . Overall System Design

JSHOP.exe

Option flags

Filename

Compression Ratio and other stats

Output File (.js)

Input File (.js)

Keyboard

Keyboard

VDU

Secondary Storage

Secondary Storage

## Modular system structure

### Process diagram

A process diagram splits the main program into separate tasks derived from the data flow diagrams and specific objectives. It makes it easier to decide how to develop the system in a modular fashion.

Figure . Process diagram

Optimisation System

Enter filename and option flags

Translate input into understandable format

Generate Syntax Tree

Perform simple optimisations

Perform complex optimisations

Perform partial evaluation

Return to user with stats

### Module descriptions

The new system can now be split up into several distinct modules, each designed to carry out a specific task based on the main processes. These will now be discussed in detail.

The diagram on the next page shows the components usually used to design a compiler. This project is similar to a compiler in many ways except that the target language is the same as the source language. Given this similarity, almost every stage will be identical.

This diagram was copied from lecture notes for the Compilers module in the School of Computer Science at the University of Nottingham (Nilsson D. N., 2011). The design of the new system will be based on this structure in many key ways. The main difference will be in the Checker section. As the input language is also the output language, there is no need to type check or verify the validity of the input code. As stated in the Assumptions section in Chapter 2 (page 6), it is expected that the input is already valid JavaScript and correctly typed. This section of the “compiler” will be substituted with an Analyser which will map out variable and function declarations. Let us now delve into each section in more detail.

Scanner

Optimiser / Code Generator

Checker

Parser

sequence of characters

sequence of tokens

Abstract Syntax Tree (AST)

Intermediate Representation (IR)

target code

Lexical Analysis

Syntactic Analysis/Parsing

Contextual Analysis/checking Static Semantics (e.g. Type Checking)

Optimisation and Code Generation (possibly many steps involving a number of intermediary representations)

Front End

Middle Section / Back End

Figure . Typical Compiler Structure

## Lexer

When the initial uncompressed file is fed into the system, it will simply be a stream of bytes, or more specifically, a stream of characters. As it is, this means nothing to the system and cannot be manipulated, except by crude methods such as regular expressions as discussed in the prototyping section. In order to make the input more useable, the stream of characters must be scanned and translated into a stream of *tokens* which mean something to the system.

These tokens must be defined to represent all possible words and symbols that the input language contains. For example, if the input language is English, there must be tokens to represent verbs, nouns, adjectives, punctuation, paragraphs etc. As the input language is JavaScript, the following tokens were chosen:

**data** Token

= WS -- ^ Whitespace

| SLCom -- ^ Single line comment (necessary for line counting

- discarded after lexer)

| LitInt Integer -- ^ Integer literals

| LitFloat Double -- ^ Float literals (Using Haskell's Double for safety)

| LitStr String -- ^ String literals

| Id String -- ^ Identifiers

| Regex String -- ^ Regular expressions

| ResId String -- ^ Reserved identifier

| ResOp String -- ^ Reserved operator

| Other String -- ^ Unknown other symbol

| EOF -- ^ End of file (input) marker

**deriving** (Eq, Show)

The most common of these tokens will be the reserved identifier (e.g. var, for, if) and reserved operator (e.g. ;, <, >=). Each identifier and operator could be explicitly defined as a separate token, but due to their numerousness, grouping them like this seems like the most sensible choice.

The next step is to write a program which will scan the stream of characters and, based on the previously defined data type, generate a list of tokens. Such a program is called a lexer. As decided in the “Potential solutions” section of Chapter 2, the lexer-generator Alex will be used to create this program. Alex has a similar syntax to Haskell although it does differ in some ways. It uses basic regular expressions to define character groups for matching tokens. However, unlike the method described in the prototyping section, it only reads through the input once in a linear manner.

The first step in building the lexer is to define the special characters and numbers that are accepted in JavaScript.

-- Special characters

$whitechar = [ \t\n\r\f\v]

$spacechar = [ \t]

$special = [\(\)\,\;\[\]\`\{\}]

$digit = 0-9

$alpha = [a-zA-Z]

-- Symbols are any of the following characters except (#) some special cases

$symbol = [\!\#\$\%\&\\*\+\.\/\<\=\>\?\@\\\^\|\-\~\,\;] # [$special \**\_**\:\"\']

$graphic = [$alpha $symbol $digit $special \**\_**\:\"\'\,]

$octit = 0-7

$hexit = [0-9 A-F a-f]

$nl = [\n\r]

$charesc = [abfnrtv\\\"\'\&\/]

@escape = \\ ( $charesc | x $hexit+ )

The next step is to define a list of all the reserved identifiers and operators that make up the JavaScript language. These were taken directly from the JavaScript Pocket Reference (Flanagan, 2002, pp. 3,10-11).

@reservedid = 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|

-- reserved words for possible future extensions

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

-- and finally, let's hope not

goto

@reservedop =

"." | "[" | "]" | "(" | ")" | "++" | "--" | "-" | "+" | "~" | "!" | "\*" | "/" | "%" | "<<" | ">>" | ">>>" | "<" | "<=" | ">" | ">=" | "==" | "!=" | "===" | "!==" | "&" | "^" | "|" | "&&" | "||" | "?" | ":" | "=" | "\*=" | "+=" | "-=" | "/=" | "%=" | "<<=" | ">>=" | ">>=" | "&=" | "^=" | "|=" | "," | ";" | "{" | "}"

Once these have been defined, a set of rules can be written to generate a list of tokens from the input.

-- String -> Token

tokens :-

<0> "//" [$spacechar $printable]\* $nl? { \s -> SLCom }

Single-line comments are defined as a double forward slash followed by any printable or whitespace characters and then a newline.

<0> $white+ { \s -> WS }

Whitespace is defined by the built in macro $white which captures spaces, tabs and newlines.

<0> @reservedid { \s -> ResId s }

<0> @reservedop { \s -> ResOp s }

Reserved operators and identifiers are captured as defined above.

<0> @float { \s -> LitFloat (read ("0" ++ s)

:: Double) }

Floats in JavaScript can be a simple number written in base ten with a decimal fraction, or they can include an exponent (e.g. 1.51e-6). The following macro was developed to capture this:

@float = $digit\* "." $digit+ ((e|E) ("+"|"-")? $digit+)?

So as to prevent possible errors, floats are stored by the system as doubles, which have twice the precision. There is a slight difference between the way Haskell and JavaScript store exponent notation numbers in that JavaScript does not require a leading zero before a decimal point and Haskell does. For example, the following is valid in JavaScript but not Haskell: .5e+10

To get around this, a leading zero is appended to the beginning of every JavaScript float so as to ensure that an error does not occur. If the zero is not needed, it is discarded automatically anyway.

<0> @hex { \s -> LitInt (read s :: Integer) }

Hexadecimal numbers are in base sixteen and are written with a preceding “0x” so the following macro was developed to capture them:

@hex = "0" ("x"|"X") $hexit+

They are stored by the system as integers as it makes calculations later on a lot simpler. In some cases, integers will be converted back into hexadecimal format if it reduces the number of characters (see page 29).

<0> @oct { \s -> LitInt (read ("0o" ++ tail

s) :: Integer) }

*“Although the ECMAScript standard does not support them, some implementations of JavaScript allow you to specify integer literals in octal (base-8) format. An octal literal begins with the digit 0 and is followed by a sequence of digits, each between 0 and 7.”* – JavaScript: The Definitive Guide (Flanagan, 2001)

Octal numbers are in base eight and are written with a preceding “0” in JavaScript. However in Haskell they are written with a preceding “0o”. The following macro was written to capture the JavaScript version:

@oct = "0" $octit+

The preceding 0 is then removed from the number and “0o” is added before converting it to an integer in the same way as hexadecimal numbers.

<0> @decimal { \s -> LitInt (read s :: Integer) }

Integers are simply a stream of digits with no fractional part. They are defined as such:

@decimal = $digit+

<0> @id { \s -> Id s }

*“Identifiers are composed of any number of letters and digits, and \_ and $ characters. The first character of an identifier must not be a digit, however.”* – JavaScript Pocket Reference (Flanagan, 2002)

$firstLetter = [$alpha \**\_** \$]

@id = $firstLetter [$alpha $digit \**\_** \$]\*

<0> \" @dString\* \" { \s -> LitStr s }

<0> \' @sString\* \' { \s -> LitStr s }

Strings in JavaScript can either be enclosed by single or double quotation marks:

@dString = $graphic # [\"\\] | " " | $nl | @escape

@sString = $graphic # [\'\\] | " " | $nl | @escape

<0> @regex { \s -> Regex s }

Regular expressions in JavaScript are tricky to match properly because it is not possible to specify context in the lexer. In essence, a regular expression is encased between forward slashes with some optional modifiers afterwards: /pattern/modifiers. However, this pattern of characters can also appear where there is a division sum followed by a comment, for example:

return x / 2; // Comment

To prevent the lexer finding regular expressions where there are none (and then probably failing to scan the rest of the file as it will almost always result in a bug), the regex matcher actually matches one extra character too. This allows the lexer to effectively dictate the context of the regex. This final character is removed from the regular expression later and added back onto the stream so that scanning can continue properly (see LexerMonad.hs in the appendices, page 73).

@reEscapedChar = \\.

@reCharClass = \[[^\]]\*\]

@reBody = @reEscapedChar | [^\[\/\\] | @reCharClass

@reMods = "g" | "i" | "m"

$reFollow = [\)\,\;\ \.\} $nl]

@regex = \/ @reBody\* \/ @reMods\* $reFollow

-- Equivalent to \/(\\.|[^\[\/\\]|\[[^\]]\*\])\*\/[gim]\*[\)\,\;\ \.\} $nl]

Forward slashes are no longer valid characters to have after a regular expression, so cases like the above division error can no longer occur. Testing has not thrown up any errors since this method was implemented. The only other way around this problem is to develop an entire regular expression parser which is beyond the scope of this project.

<0> @other { \s -> Other s }

As a final “catch-all” option, the Other token is defined as:

@other = $symbol

This concludes the Alex file which is then compiled to Haskell (Lexer.hs).

The lexing program is then controlled by LexerMonad.hs which scans the input one token at a time as it is required by the parser. The state monad is used to record the remaining input, current line number, and the latest token amongst other things.

**data** LexerState

= LS {

rest :: String, -- ^ The remaining input

lineno :: Int, -- ^ Current line number

nl :: Bool, -- ^ Newline flag

rest2 :: String, -- ^ For use with automatic semicolon

insertion

lastToken :: (Maybe Token) -- ^ The token just lexed

}

**deriving** Show

At this point it makes sense to start explaining what a monad really is. It is assumed that the reader has at least a basic understanding of functional programming techniques, but monads can be a tricky subject to get your head around. The official, formal definition goes something like this:

*A monad is a composable computation description. It represents the separation of composition from the composed computation’s execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon).[[5]](#footnote-5)*

Although this describes monads very well, it can be a little daunting at first. In essence, monads allow functions to have effects. Let’s take the Input/Output (IO) monad as an example. The type signature of a function using the IO monad would look something like this:

testFunc :: IO()

This means that the function can interact with standard input and output, and read and write files. Separate instructions are linked together using the bind operator ( >>= ). This takes the result of the previous instruction and feeds it into the next instruction as an argument. There is also a shorthand version of this in Haskell called the do notation. For example:

testFunc :: IO()

testFunc = **do**

c <- getChar

putChar c

The first instruction asks for a character to be inputted, and then it passes the result of this request to the second instruction which writes it back to standard input. Monads also support the return operation which acts in a similar way to the return instruction in imperative languages.

testFunc2 :: IO Char

testFunc2 = **do**

c <- getChar

return c

In this project, the state monad is used often, which allows information to be stored and accessed by different functions. A data type is defined (as above) which is then updated as new information is added. In the Lexer, the library-defined state monad, StateT is used. This supports the get and put functions which allow you to retrieve the state and modify it. For example, this function scans comments:

scanComment :: (Token -> P a) -> P a

scanComment cont = **do**

chr <- get

**case** (rest chr) **of**

('\n':xs) -> **do**

put chr {rest = xs, lineno = (lineno chr) + 1}

scanComment cont

('\*':'/':xs) -> **do**

put chr {rest = xs}

monadicLexer cont

(**\_**:xs) -> **do**

put chr {rest = xs}

scanComment cont

The get function retrieves the state which is then updated with further annotations to its stored data structure using the put function.

The lexer function is responsible for finding the next token in a given string. It either returns the remaining string and the token, or just the remaining string if the token was whitespace (as this should be ignored).

lexer :: String -- ^ The remaining input

-> Either (Token,String) String -- ^ Either the token and the remaining

-- string or just the remaining

-- string if the token is to be

-- skipped

lexer input = go ('\n', input)

**where**

go inp@(**\_**,rem) =

**case** alexScan inp 0 **of**

AlexEOF -> Left (EOF, [])

AlexError (c,cs) -> error ("Lexical error at " ++ take 50 cs)

AlexSkip inp' len -> go inp'

AlexToken inp'@(x,xs) len act -> **case** act (take len rem) **of**

WS -> Right xs -- Skip whitespace

-- Regexs match one char too many (see note in Lexer.x) so

-- this corrects it.

Regex s -> Left (Regex (init s), (last s):xs)

token -> Left (token, xs)

## Parser

We started off with a list of characters which could mean anything. This was of no use so we converted it into a list of tokens. This is a lot more useful but it still doesn’t really tell us how the program works. All we know at the moment is what each word or symbol is, not what it means.

The purpose of a parser is to put the list of tokens in *context*. It looks at the layout of the tokens and interprets the purpose of each sequence. It creates a *tree*.

Example input: var x = 5;

Step 1: List of characters

Step 2: List of tokens

Step 3: Parse Tree

Statement

VarStmt

VarDecl “x”

LInt 5

|  |
| --- |
| ResId “var” |
| WS |
| Id “x” |
| WS |
| ResOp “=” |
| WS |
| LitInt 5 |
| ResOp “;” |

|  |
| --- |
| v |
| a |
| r |
|  |
| x |
|  |
| = |
|  |
| 5 |
| ; |

Figure . Data journey: From characters to syntax tree

### Parse tree

The first step in designing a parser is to define the parse tree. The parse tree is the representation of the entire program and how everything relates to everything else. For example, an if-else statement has three parts: the condition, the code to execute if the condition is true, and the code to execute if the condition is false. This can be represented in the following way:

**data** IfStmt

= IfElse Expression Statement Statement

The conditional part is an Expression which will evaluate to true or false and the two branches are Statements which may or may not be executed when the condition is decided.

The parse tree for JavaScript can be determined by examining the official specification (ECMA International). This lays out in plain English the syntax and semantics for each JavaScript expression, statement and function declaration. For example, this is what it says about the if statement:

**Syntax**

*IfStatement* :

**if** ( *Expression* ) *Statement* **else** *Statement*

**if** ( *Expression* ) *Statement*

Each **else** for which the choice of associated **if** is ambiguous shall be associated with the nearest possible **if** that would otherwise have no corresponding **else**.

**Semantics**

The production *IfStatement* : **if** ( *Expression* ) *Statement* **else** *Statement* is evaluated as follows:

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, go to step 7.
5. Evaluate the first *Statement*.
6. Return Result(5).
7. Evaluate the second *Statement*.
8. Return Result(7).

The production *IfStatement* : **if** ( *Expression* ) *Statement* is evaluated as follows:

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, return (**normal**, **empty**, **empty**).
5. Evaluate *Statement*.
6. Return Result(5).

(ECMA International, pp. 63-64)

At this point, the semantics information is not required, although it can be helpful to understand how the statement works when designing a suitable data type. The really important part is the syntax at the top which shows exactly what symbols and identifiers are used to make up the possible statements. We can deduce several things about the structure of the parse tree by examining this:

1. There are two different types of if statement, one with an else branch and one without.
2. They both start with the “if” keyword followed by an opening bracket.
3. After this there is an expression followed by a closing bracket.
4. Then follows a statement of some sort.
5. This marks the end of the simple if statement, however the if-else statement then continues with the “else” keyword followed by another statement.

Using this information it is possible to design a data type which can store all the relevant data. The if and else keywords do not need to be stored because they are not variable. However the expression and statement(s) are unknown quantities so need to be stored. With this in mind, the following if statement data type can be created:

**data** IfStmt

= IfElse Expression Statement Statement

| If Expression Statement

**deriving** Show

Using this method, an entire parse tree can be built up for the JavaScript language. Another good example is that of literals. These are the basic literal data types:

**data** Literal

= LNull

| LBool Bool

| LInt Integer

| LFloat Double

| LStr String

**deriving** Show

The null data type does not require any more information as it has only one value, itself. The other literals all store their specific values.

### Happy parser

Once the entire parse tree has been constructed so that the JavaScript program can be stored, the parser itself must be built. As stated in the “Potential solutions” section of Chapter 2, the parser-generator Happy will be used. Happy lets you define the syntax of a language using a Backus-Naur Form (BNF) style specification. It has a similar syntax to Alex as they were both written by the same person (Simon Marlow) and are designed to be used together.

The first step is to define all the tokens which may be encountered. Names are given to each of the possible tokens which can be produced by the lexer. For example:

(The $$ notation represents the value of the token.)

%token

LITINT { LitInt $$ }

LITFLOAT { LitFloat $$ }

LITSTR { LitStr $$ }

ID { Id $$ }

REGEX { Regex $$ }

BREAK { ResId "break" }

CASE { ResId "case" }

CATCH { ResId "catch" }

...

These are called *terminal* symbols. After the tokens are defined, a set of rules must be written which denote the structure and syntax of the language.

* A terminal is a symbol which states a single input and maps it to a single output. For example, the empty statement which in JavaScript is represented by a single semicolon, can be represented by this terminal rule:

statement :: { Statement }

: ';' { EmptyStmt }

...

* A non-terminal is a rule which references another rule. For example, an if statement has multiple representations as we saw earlier on, so it has its own rule.

statement :: { Statement }

: ';' { EmptyStmt }

| ifStmt { IfStmt $1}

...

ifStmt :: { IfStmt }

: IF '(' expression ')' statement ELSE statement

{ IfElse $3 $5 $7 }

| IF '(' expression ')' statement

{ If $3 $5 }

Although it is not mandatory, we are using uppercase to specify terminal symbols, and lower case to specify non-terminal symbols. This makes it much easier to differentiate between the two. An example of this can be seen clearly in the primary expression rule. Literals are primary expressions, but it is simpler and clearer to write a new non-terminal and group them elsewhere.

primaryExpr :: { PrimaryExpr }

: literal { ExpLiteral $1 }

| ID { ExpId $1 }

| THIS { ExpThis }

| REGEX { ExpRegex $1 }

| arrayLit { ExpArray $1 }

| objectLit { ExpObject $1 }

| '(' expression ')'

{ ExpBrackExp $2 }

literal :: { Literal }

: NULL { LNull }

| TRUE { LBool True }

| FALSE { LBool False }

| LITINT { LInt $1 }

| LITFLOAT { LFloat $1 }

| LITSTR { LStr $1 }

The full source listing of the Parser.y file can be found in the appendix, page 81.

## Code Compressor

Once the input has been through the lexer and parser, it is in a state suitable for manipulation and optimisation. It can now be rebuilt from the ground up in the way which we desire as laid out in the specific objectives. The module CodeCompressor.hs is in charge of this operation. A state monad is defined to store the generated JavaScript and the parse tree is traversed to deal with each instruction.

run :: Tree -> JSCC ()

run (Tree sources) = **do**

anSrcSeq sources

resetScope

genSrcSeq sources

genSrc :: Source -> JSCC()

genSrc (Statement stmt) = genStmt stmt

genSrc (SFuncDecl funcDecl) = genFuncDecl funcDecl

genSrcSeq :: [Source] -> JSCC()

genSrcSeq s = mapM\_ genSrc s

...

At each point on the tree where syntax must be generated, the emit function is used to store it as a string until the full program is generated.

genItStmt :: IterativeStmt -> JSCC()

genItStmt (DoWhile stmt expr) = **do**

emit "do "

genStmt stmt

emit " while("

genExpr expr

emit ");"

...

emit :: i -> CC i x ()

emit i = CC $ \ccs -> ((), ccs {sect = i : sect ccs})

While generating the syntax, certain optimisations can be made.

## Simple optimisations

“Simple” optimisations, in the context of this project, can be defined as:

* Removal of whitespace
* Removal of comments
* Array and object shorthand replacement
* Removal of unnecessary semicolons
* Micro-optimisations

Both **comments** and **whitespace** are effectively removed by the lexer. They are detected like any other token and then skipped when it comes to recording the list.

monadicLexer' :: (Token -> P a) -> P a

monadicLexer' cont = **do**

chr <- get

**case** (rest chr) **of**

('\n':xs) -> **do**

put chr {rest = xs, lineno = (lineno chr) + 1, nl = True}

monadicLexer' cont

('/':'\*':xs) -> **do**

put chr {rest = xs}

scanComment cont

**\_** -> **do**

**let** lexResult = lexer (rest chr)

**case** lexResult **of**

-- Specifically check for single line comments. Increment

line number then skip.

Left (SLCom, xs) -> **do**

put chr {rest = xs, lineno = (lineno chr) + 1, nl = True}

monadicLexer' cont

Left (token, xs) -> **do**

put chr {rest = xs, rest2 = (rest chr), lastToken = Just

token}

cont token

Right xs -> **do**

put chr {rest = xs}

monadicLexer' cont

The lexer function, as shown above, either returns the next token and the remaining string, or just the remaining string if the token was whitespace. The scanComment function removes all input up until the next “\*/”:

scanComment :: (Token -> P a) -> P a

scanComment cont = **do**

chr <- get

**case** (rest chr) **of**

('\n':xs) -> **do**

put chr {rest = xs, lineno = (lineno chr) + 1}

scanComment cont

('\*':'/':xs) -> **do**

put chr {rest = xs}

monadicLexer cont

(**\_**:xs) -> **do**

put chr {rest = xs}

scanComment cont

Specific objective 3d in Chapter 2 (page 4) states that **semi-colons before closing braces** should be removed. The simplest way of doing this is to run the output of the code compressor through a function which looks for the pattern “;}” and replaces it with “}”. This function is run after all other optimisations are completed.

cleanup :: String -> String

cleanup [] = []

cleanup (';':'}':xs) = '}':(cleanup xs)

...

This function also deals with converting **array and object declarations** to shorthand:

cleanup str

| Just xs <- stripPrefix "new Object()" str = '{':'}':(cleanup xs)

| Just xs <- stripPrefix "new Object;" str = '{':'}':';':(cleanup xs)

| Just xs <- stripPrefix "new Array()" str = '[':']':(cleanup xs)

| Just xs <- stripPrefix "new Array;" str = '[':']':';':(cleanup xs)

Now that these simple optimisations have been carried out, the size of the input file has been vastly reduced. None of these changes makes a difference to the way in which the program runs, but the footprint is much smaller meaning the JavaScript file, and by extension the website, will load much faster.

**Micro-optimisations** are those that, although not terribly significant, can make a difference to large scale JavaScript files. There are several small actions that can be taken to shave off one or two characters in certain areas.

* **Literal booleans**

True can be represented as 1 in JavaScript, however if you simply put 1, it will be assumed to be an integer. Adding ! evaluates it as a boolean expression. Consequently, true can be expressed as "not false" and false can be expressed as "not true".

genLiteral :: Literal -> JSCC()

...

genLiteral (LBool True) = emit "!0"

genLiteral (LBool False) = emit "!1"

...

* **Literal numbers**

Literal numbers can be expressed in hexadecimal notation in JavaScript. Hex numbers begin with "0x" so are not always shorter. To find the point at which it becomes shorter to express a number in hex, the following function was devised:

findPoint :: Integer -> IO ()

findPoint n = **do**

**let** hex = "0x" ++ (showHex n "")

**if** length (show n) > length hex

**then** putStrLn $ hex ++ " (" ++ (show (length hex))

++ ") is shorter than " ++ show n

++ " (" ++ (show (length (show n)))

++ ")"

**else** **do**

putStrLn $ hex ++ " (" ++ (show (length hex)) ++ "), "

++ (show n) ++ " ("

++ (show (length (show n))) ++ ")"

findPoint (n+1)

After being run in GHCI, this was the relevant result:

> \*Main> findPoint 999999999995

> 0xe8d4a50ffb (12), 999999999995 (12)

> 0xe8d4a50ffc (12), 999999999996 (12)

> 0xe8d4a50ffd (12), 999999999997 (12)

> 0xe8d4a50ffe (12), 999999999998 (12)

> 0xe8d4a50fff (12), 999999999999 (12)

> 0xe8d4a51000 (12) is shorter than 1000000000000 (13)

The chances of there ever being a literal number this large expressed in a script are very low, but at least it is covered. So far, no other compressor has been found that does this. The maximum integer in JavaScript is 9007199254740992 so it is possible for this compression technique to take effect.

compInt :: Integer -> String

compInt n = **if** n < 1000000000000 **then** show n

**else** "0x" ++ showHex n ""

* **Literal Strings**

JavaScript strings can be wrapped in single or double quotes. Escaping quotes requires an extra character (backslash), so it is sometimes possible to switch the type of quote used to wrap the string so that it is no longer necessary to escape.

This function optimises strings with escaped quotes in them.

For example:

> 'te\'st' -> "te'st"

> "te\"st" -> 'te"st'

> "te'st\"in\"g" -> 'tes\'st"in"g'

compStr :: Maybe Char -> String -> String

compStr mbQ str =

**case** mbQ **of**

Nothing -> **if** length fullStr < length str **then** fullStr **else** str

Just q -> q:(genStrBody (init(tail str)) q) ++ [q]

**where**

origType = head str

altType = **if** origType == '\'' **then** '"' **else** '\''

qList = getEscQ str

qType = **if** countElem origType qList > countElem altType qList

**then** altType

**else** origType

strBody = genStrBody (init(tail str)) qType

fullStr = qType:(strBody ++ [qType])

genStrBody :: String -> Char -> String

genStrBody [] **\_** = []

genStrBody ('\\':'\'':xs) '\'' = '\\':'\'':(genStrBody xs '\'')

genStrBody ('\\':'\'':xs) qType = '\'':(genStrBody xs qType)

genStrBody ('\'':xs) '\'' = '\\':'\'':(genStrBody xs '\'')

genStrBody ('\'':xs) qType = '\'':(genStrBody xs qType)

genStrBody ('\\':'"':xs) '"' = '\\':'"':(genStrBody xs '"')

genStrBody ('\\':'"':xs) qType = '"':(genStrBody xs qType)

genStrBody ('"':xs) '"' = '\\':'"':(genStrBody xs '"')

genStrBody ('"':xs) qType = '"':(genStrBody xs qType)

genStrBody (x:xs) qType = x:(genStrBody xs qType)

getEscQ :: String -> [Char]

getEscQ [] = []

getEscQ ('\\':'\'':xs) = '\'':(getEscQ xs)

getEscQ ('\\':'"':xs) = '"':(getEscQ xs)

getEscQ (x:xs) = getEscQ xs

countElem :: Eq a => a -> [a] -> Int

countElem i = length . filter (i==)

* **Literal floats**

JavaScript is dynamically typed, so the number 12.0 is indistinguishable from 12. In some programming languages, it would matter as to whether the number is a float or an integer, but in JavaScript it does not. With this in mind, it makes sense to compress all floats with a decimal value of 0 to integers as it will save two characters. The following function takes a floating point number as input and outputs either the same number, or the integer equivalent if it ends in “.0”.

roundIfInt :: (RealFrac a, Integral b) => a -> Either a b

roundIfInt n = **if** isInt n **then** Right (round n) **else** Left n

**where**

isInt :: RealFrac a => a -> Bool

isInt x = x == fromInteger (round x)

To see how much of a difference all these optimisations make, a test was carried out on five popular JavaScript libraries (more about this will be explained later). It was found that an average file could be reduced to 57.41% of its original size with only these measures in effect.

## More complex optimisations

The next step is trickier. The easy things have been removed and modified, but 57.41% is not enough! We can do better. This section will deal with:

* Reducing the length of identifiers (variable and function names)
* Converting simple conditional expressions to ternary notation

When dealing with variables, the first thing to understand is the concept of scope. A variable in JavaScript is only “alive” or useable during the scope in which it is declared, after the point at which it is declared. Scope is defined by functions. For example, in the following code snippet, myVar1 is available everywhere because it is defined right at the beginning, before any functions are defined. This is called “global scope”:

***var*** myVar1 **=** 3**;**

***function*** testFunc**(**age**,** name**)** **{**

setAge**(**age**);**

setName**(**name**);**

myVar1 **+=** 2**;**

**}**

***function*** testFunc2**()** **{**

**}**

The other variables (age and name) are declared in the function testFunc and cannot be used anywhere else. The currently empty function testFunc2 has access only to myVar1.

To handle this concept, a record of the current scope and level of any live variables has to be maintained at all times while the code is being generated. When a function is entered, the scope is increased by one, and when it is exited it is decreased by one. The level of an identifier is related to its indentation. The following diagram gives a clear definition of scope and level as used in this project:

var **a** = “Hello”;

function **func1**(**arg1**, **arg2**) {

var **b**;

function **func2**(**arg3**) {

var **c**;

function **func3**() {

var **d**;

}

function **func4**(**arg4**) {}

}

function **func5**(**arg5**) {

var **e**;

}

}

Scope 0, Level 0

Scope 1, Level 1

Scope 2, Level 2

Scope 3, Level 3

Scope 4, Level 3

Scope 5, Level 2

Figure . Example of identifier scope and level

The decScope function also provides the option to forget all the variables from the scope we are exiting.

incScope :: CC String () ()

incScope = **do**

ccs@(CCS {currentScope = cs, currentLevel = cl}) <- get

put $ ccs {currentScope = cs+1, currentLevel = cl+1}

decScope :: Bool -> CC String () ()

decScope forget = **do**

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

**let** ids' = **if** forget **then** forgetScopes ids cs cl

**else** ids

put $ ccs {identifiers = ids', currentScope = cs, currentLevel = cl-1}

**where**

forgetScopes :: [(Identifier, (Int,Int))] -> Int -> Int ->

[(Identifier, (Int,Int))]

forgetScopes [] **\_** **\_** = []

forgetScopes (x:xs) s l = **if** fst (snd x) <= s && snd (snd x) == l

**then** forgetScopes xs s l

**else** x:(forgetScopes xs s l)

It would be nice to be able to calculate which variable is being referenced on-the-fly as the code is being compressed; however there are some corner cases where a variable can be used before it is declared. In these cases, it is impossible to know what scope or level the variable being referred to is in. An example of this situation can be seen in this snippet from the jQuery library:

***var*** jQuery **=** ***function*()** **{**

***return*** init**(**rootjQuery **);**

**},** rootjQuery**;**

The variable rootjQuery is used as an argument to the function init before it is declared in the lower scope.

To get around this unfortunate development, it becomes necessary to analyse the entire input and record a map of every variable and function declaration and its scope before the code is compressed. This is handled by the Analyser.hs module. Here is an example of how a function is dealt with in the analyser:

anFuncDecl :: FuncDecl -> JSCC()

anFuncDecl (FuncDecl (Just id) formalParamList sources) = **do**

regID id

incScope

anFormalParamList formalParamList

anSrcSeq sources

decScope False

First, the function name (id) is registered using the regID function (see below). After this, the incScope function is called to increase the current scope of the code compression state monad. The parameter list is analysed, followed by the body of the function, and then the scope is decreased again, with the “forget variables” flag set to false. Here is the regID function:

regID :: String -> CC String () ()

regID id = **do**

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

put $ ccs {

identifiers = ((Variable {

origName = id,

compName = ""

}),(cs,cl)):ids

}

It takes the current scope and level and records the current uncompressed name of the identifier with these values in the identifiers list. This is carried out with every identifier in the JavaScript program so that when they are referred to they can be looked up and compressed appropriately.

Now it is a reasonably simple job to track variables and assign them with new compressed names. When an identifier is encountered in the compressor, the emitID function is called, for example:

genVarDecl :: VarDecl -> JSCC()

genVarDecl (VarDecl id (Just assign)) = **do**

id' <- emitID id

emit id'

emit "="

genAssign assign

The emitID function works by looking up the identifier using the current scope and level, and the uncompressed name. If the identifier has not been compressed yet (it is either being defined for the first time, or is being used before being defined), the newID function is called which compresses it.

emitID :: String -> CC String () String

emitID origID = **do**

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

**let** mbCompID = findCompID origID ids cs cl

**case** mbCompID **of**

Just (compID, (s,l)) -> **if** compID /= "" **then**

return compID

**else** **do**

-- Generate new name

compID' <- newID origID s l

return compID'

Nothing -> **do**

-- Must be either a library function, or global var

put $ ccs {

identifiers = ((Variable {

origName = origID,

compName = origID

}),(0,0)):ids

}

return origID

**where**

findCompID :: String -> [(Identifier, (Int,Int))] -> Int -> Int ->

Maybe (String, (Int,Int))

findCompID origID [] **\_** **\_** = Nothing

findCompID origID ((ids, (s,l)):xs) cs cl =

**if** s <= cs && l <= cl && origID == (origName ids) **then**

Just (compName ids, (s,l))

**else**

findCompID origID xs cs cl

Some identifiers must not be compressed. Global variables may be referenced by outside sources, and some functions imported from libraries or other modules likewise must not be altered, otherwise the script would stop working and the specification, that the output must have the exact same result as the input, would be unfulfilled.

The newID function first checks that the identifier should be compressed, then, assuming it should, generates a new compressed name for it. That compressed name is saved in the identifiers list so that it can be used whenever the identifier is referenced again.

newID :: String -- ^ Original ID

-> Int -- ^ Scope of original ID

-> Int -- ^ Level of original ID

-> CC String () String -- ^ Return the compressed ID

newID origID s l = **do**

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

**let** usedIDs = [compName i | (i,(s',l')) <- ids, s' <= cs, l' <= cl]

-- only from same scope or lower

**let** compID = **if** l == 0 **then** -- 0 = global

origID

**else** genID usedIDs genIDList

**let** ids' = updateCompID origID s l compID ids

put $ ccs {identifiers = ids'}

return compID

**where**

updateCompID :: String -> Int -> Int -> String -> [(Identifier,

(Int,Int))] -> [(Identifier, (Int,Int))]

updateCompID **\_** **\_** **\_** **\_** [] = []

updateCompID origID s l compID ((id,(s',l')):ids)

= **if** origName id == origID && s' == s && l' == l **then**

((Variable {origName = origID, compName = compID}),

(s,l)):ids

**else**

(id,(s',l')):(updateCompID origID s l compID ids)

genID :: [String] -> [String] -> String

genID [] idList = head idList

genID usedIDs (id:ids) = **if** elem id usedIDs **then** genID usedIDs ids

**else** id

When generating a new identifier name, we must generate a list of all possible identifier names and then select the shortest one which has not yet been used. As stated in the Lexer section beginning on page 17:

*“Identifiers are composed of any number of letters and digits, and \_ and $ characters. The first character of an identifier must not be a digit, however.”* – JavaScript Pocket Reference (Flanagan, 2002)

With this in mind, the following function was designed to create an infinite list of all valid names:

genIDList :: [String]

genIDList = [c:s | s <- "":allStrings, c <- firstChar] \\ reservedIDs

**where**

firstChar = ['a'..'z']++['A'..'Z']++['\_']

-- Technically the first char can be a '$' but this is

-- monopolised by jQuery so we'll leave it out for simplicity

-- Note: To make all var names unique, I could use a special

-- character... Maybe 'λ'?

-- Unfortunately this is not valid ASCII so it makes reading and

-- writing files very difficult!

alph = ['a'..'z']++['A'..'Z']++['0'..'9']++['$','\_']

allStrings = [c:s | s <- "":allStrings, c <- alph]

reservedIDs = ["if","in","do","int","for","new","try","var"]

-- There are obviously more, but none shorter than 4 letters.

-- A longer list would decrease performance unnecessarily.

-- Variables longer than 3 letters should never arise.

-- > length $ takeWhile (\xs -> length xs < 4) genIDList

-- > 220525

As explained in the comments for this function, it is necessary to exclude reserved identifiers from this list so as to prevent an identifier being named “if” or similar. However, it would be a waste of system resources to exclude *all* reserved identifiers as there is little to no chance of them ever being encountered. Again, as shown, there would have to be 220,525 variables in scope at once for a four letter identifier to be chosen. This is sufficiently unlikely that it is reasonable to leave this as a documented feature/potential bug.

In most standard imperative languages, generating an infinite list like this would cripple the system and cause the program to hang or crash. However, due to Haskell’s use of lazy evaluation, the list is generated only up to what is required by the function at the time. If the function needs only the first 10 items, then only the first 10 items will be calculated. If the function requires 100 items, then 100 items will be calculated. This makes choosing the next available identifier name a trivial exercise.

Ternary expressions are a simple device for defining the value of a variable conditionally. It is possible to write any ternary conditional as a simple if statement, and this is often done as if statement are slightly simpler to understand, especially for novice programmers. The following two code snippets would have exactly the same result:

*If statement version*

***if*** **(**a **>** 5**)** **{**

result **=** Pass**;**

**}** ***else*** **{**

result **=** Fail**;**

**}**

*Ternary conditional version*

result **=** a **>** 5 **?** "Pass" **:** "Fail"**;**

Clearly the ternary version is more compact so it makes sense to detect if statements of this sort and rewrite them in ternary notation.

genIfStmt :: IfStmt -> JSCC()

genIfStmt ifStmt@(IfElse expr

(ExprStmt (Assignment [Assign leftExprTrue assignOpTrue assignTrue]))

(ExprStmt (Assignment [Assign leftExprFalse assignOpFalse assignFalse])))

= **if** leftExprTrue == leftExprFalse **then**

genTernaryCond expr leftExprTrue assignOpTrue assignTrue assignFalse

**else**

genIfElse ifStmt

genIfStmt ifStmt@(IfElse expr

(Block [ExprStmt (Assignment [Assign leftExprTrue assignOpTrue

assignTrue])])

(Block [ExprStmt (Assignment [Assign leftExprFalse assignOpFalse

assignFalse])]))

= **if** leftExprTrue == leftExprFalse **then**

genTernaryCond expr leftExprTrue assignOpTrue assignTrue assignFalse

**else**

genIfElse ifStmt

genTernaryCond :: Expression -- ^ Condition

-> LeftExpr -- ^ Left side of assignment

-> AssignOp -- ^ Assignment operator

-> Assignment -- ^ Assignment if true

-> Assignment -- ^ Assignment if false

-> JSCC()

genTernaryCond expr leftExpr assignOp assignTrue assignFalse = **do**

leftExpr' <- genLeftExpr leftExpr

genMaybe genPrimExpr leftExpr'

genAssignOp assignOp

genExpr expr

emit "?"

genAssign assignTrue

emit ":"

genAssign assignFalse

emit ";"

This has the effect of translating any if statement of the form shown above into a ternary conditional.

Already we have made some major reductions to the code size, but we can do even better.

## Partial evaluation

Partial evaluation is the process of carrying out certain sums or calculations on expressions within the program. Certain calculations cannot be made until runtime, so cannot be evaluated at all. For instance, the following script relies on input from the user so cannot be reduced or calculated until the point when the user enters information into it:

***var*** a **=** prompt**(**"Please enter your name"**,**""**);**

document**.**write**(**"Hello" **+** a**);**

However, in some cases, all the parameters for a calculation are available before run-time and can be evaluated in advance. For example:

***var*** a **=** 35 **+** 2**;**

or

***var*** b **=** "Hello" **+** " World"**;**

In these cases, partial evaluation will reduce the number of characters and also speed up execution. As explained in Chapter 1, the use of partial evaluation in this project is aimed at reducing the character count of the program, not improving performance. In certain cases, partial evaluation can increase the character count and therefore will not be used. An example is with simple calculations:

***var*** c **=** 4 **/** 3**;**

This could be evaluated to 1.333333… but it is much more compact to leave the sum as 4/3 even though it would be better from a performance point of view to calculate the sum in advance.

To implement this, it is necessary to stop the Code Compressor from emitting primary expressions before they have been partially evaluated. The type signature for all expressions is therefore changed to return Maybe PrimaryExpr instead of () (unit). This means that the primary expression can be passed up the tree until it has to be emitted. Along the way it can be used to calculate sums and the result can be emitted rather than the arguments.

genAddExpr :: AddExpr -> JSCC (Maybe PrimaryExpr)

...

genAddExpr (Plus addExpr multExpr) = **do**

a <- genAddExpr addExpr

**if** isJust a **then** genPrimExpr $ fromJust a **else** emit ""

emit "+"

b <- genMultExpr multExpr

res <- peSimpCalc genPrimExpr a b '+'

return res

...

The function peSimpCalc is the driving force behind the process of partial evaluation. It takes the primary expression generation function as an argument as well as the two operands and the operator. If both the operands are literals, they are partially evaluated. If they are not, they are emitted as usual along with the operator as a sum.

peSimpCalc :: (PrimaryExpr -> JSCC()) -- ^ genPrimExpr function

-> Maybe PrimaryExpr -- ^ First operand

-> Maybe PrimaryExpr -- ^ Second operand

-> Char -- ^ Operator

-> JSCC (Maybe PrimaryExpr)

peSimpCalc gen (Just (ExpLiteral a)) (Just (ExpLiteral b)) op = **do**

pop -- operator

pop -- first operand

**let** (x, mbY) = simpCalcLit a b op

**if** litLength x > origLength **then** **do**

genLiteral a

emit [op]

genLiteral b

return Nothing

**else** **if** isJust mbY **then** **do**

genLiteral x

emit [op]

genLiteral $ fromJust mbY

return Nothing

**else** return $ Just (ExpLiteral x)

**where**

origLength = litLength a + litLength b + 1 -- The 1 is the op

litLength :: Literal -> Int

litLength (LNull) = 4 -- null

litLength (LBool **\_**) = 2 -- !0 or !1

litLength (LInt x) = length $ show x

litLength (LFloat x) = length $ dropPrefix $ show $ roundIfInt x

litLength (LStr s) = length s

peSimpCalc gen Nothing (Just (ExpLiteral b)) **\_**

= genLiteral b >> return Nothing

peSimpCalc gen **\_** (Just b) **\_**

= gen b >> return Nothing

peSimpCalc **\_** **\_** **\_** **\_**

= return Nothing

The simpCalcLit function simply carries out the calculation. If both literals are numbers, the answer is returned as you would expect. If both are strings, they are concatenated together using the following function. As strings in JavaScript can have double or single quotes, this needed to be accounted for

concatStr :: String -> String -> String

concatStr x@('\'':xs) ('\'':ys) = init x ++ ys

concatStr x@('"':xs) ('"':ys) = init x ++ ys

concatStr x@('"':xs) y@('\'':ys) = init x ++ (tail $ compStr (Just '"') y)

concatStr x@('\'':xs) y@('"':ys) = init x ++ (tail $ compStr (Just '\'') y)

concatStr x y = x ++ y

Once the relevant result is returned, it can be emitted as usual, using the custom higher-order function genMaybe.

genAssign :: Assignment -> JSCC()

genAssign (CondExpr condExpr) = **do**

condExpr' <- genCondExpr condExpr

genMaybe genPrimExpr condExpr'

...

genMaybe :: (a -> JSCC()) -> (Maybe a) -> JSCC()

genMaybe genFunc mbExpr

= **case** mbExpr **of**

Just expr -> genFunc expr

Nothing -> return ()

To use more partial evaluation techniques, a more simplified Abstract Syntax Tree would have to be created. The current parse tree is a Concrete Syntax Tree with a lot of extraneous information contained within it about the structure of the language grammar. This is useful in some aspects, but gets in the way when dealing with literal values. If this extra information could be discarded and an AST created, the possibilities for partial evaluation would increase. Recursive functions could be unwinded and some function calls could be negated.

***function*** powerOf**(**x**,** y**)** **{**

***if*** **(**y **<=** 0**)** ***return*** 0**;**

***else*** ***if*** **(**y **==** 1**)** ***return*** x**;**

***else*** **{**

***return*** x **\*** powerOf**(**x**,**y**-**1**);**

**}**

**}**

powerOf**(**x**,**3**);**

would become

x**\***x**\***x**;**

and

***function*** errorMsg**(**msg**)** **{**

alert**(**"Error: " **+** msg**);**

**}**

errorMsg**(**"Invalid input"**);**

would become

alert**(**"Error: Invalid input"**);**

The translation of the CST into an AST constitutes a lot of work, but would be a suitable extension.

## Interface

Although the main functionality of this program is based around compression techniques, it is important to provide a useable, intelligent interface with which to interact with the system. As stated in the “User’s knowledge of information technology” section in Chapter 2 (page 8), the command line interface must support a help flag which explains how to invoke the program with various options.

Typing the command ./jshop.exe --help produces the following message:

$ ./jshop.exe --help

USAGE:

jshop [options] file.js Compress "file.js"

jshop [options] Read input from standard input.

(Terminate with Ctrl+D, Enter in UNIX, or

Ctrl+Z, Enter in Windows. Must be on a new line.)

DEFAULT OUTPUT:

Compressed JS and ratio of input to output.

OPTIONS:

Output

--help, --h

Print help message and stop.

--version, --ver, --v

Print JSHOP version and stop.

--input, --i

Print input.

--tokens, --tok

Print list of tokens.

--tree

Print the Parse tree.

--lstate

Print the Lexer state.

--all

Print input, tokens, Parse tree, Lexer state, output and ratio.

--rembloat

Experimental. Removes bloat from the Parse tree. Can make it

unreadable.

Only works with --tree or --all.

--prettyprint, --pp

Experimental. Pretty prints the output for ease of reading.

Testing

--test --t ["message"]

Run full test suite and stop. Message is optional. If added, test

results will be saved to file.

--showAverages

Displays the average compression ratios for all past tests.

--showTest NUM

Displays a past test of index NUM.

--showAllTests

Displays all past tests.

The simplest invocation of the program is to call it with only one argument, a filename. In this case it will attempt to compress the contents of the file giving the output along with some simple ratio information. Other options can be added to show more information about the state of the input at different stages. For example, the --tree flag will show the parse tree as well as the final compressed code. For example:

$ ./jshop.exe --tree --input tests/test.js

INPUT:

function test(arg1) {}

function test2(arg2, arg3) {}

PARSE TREE:

SFuncDecl (FuncDecl (Just "test")

["arg1"]

[])

SFuncDecl (FuncDecl (Just "test2")

["arg2","arg3"]

[])

OUTPUT:

function test(a){}function test2(a,b){}

STATS:

Reduced by 15 chars, 72.22% of original.

Total execution time: 0.000 secs

|  |
| --- |
| Chapter 4 |

# Implementation

The process of building the system began in September 2011. A prototype was developed, as discussed in Chapter 3, which could remove whitespace and comments using regular expressions. By November 2011, it became clear that this approach was not satisfactory, and work on the new lexer/parser method began.

It was decided early on that a lexer- and parser-generator needed to be used for a language as complex as JavaScript. It would be a poor use of time to attempt to build such a system without the use of generators for these particular tasks. With this in mind, Alex and Happy were chosen and work began on the lexer.

It was decided that the overall framework of the project should be based on Dr. Henrik Nilsson’s Haskell MiniTriangle Compiler (HMTC) (Nilsson D. H., 2006-2011) which is used in the Compilers module for Computer Science at the University of Nottingham. The HMTC project has a strong, well-built source base which utilises the Happy parser-generator and goes on to generate code in a similar way to what is necessary for this system. Many aspects had to be modified however, and the Alex lexer-generator had to be included as it is not used in HMTC.

Many modules have been newly written, but those which borrow from the HMTC framework have it clearly stated in the copyright notice at the top of the source code.

## Problems encountered and their solutions

During the construction of the project, there were various problems and bugs. For the most part, these bugs were detected by testing the code on various libraries as will be explained in the next section. A record of each issue was kept along with its solution and any comments related to it.

|  |  |  |  |
| --- | --- | --- | --- |
| Description | Date found | Date solved | Details |
| Line numbers are calculated incorrectly by the lexer. Single line comments mess up the numbering. LOW PRIORITY. | 14/1 | 28/2 | Added token for single line comments rather than calling them whitespace. Deal with them in monadicLexer'. |
| Spaces removed between unary and postfix operators.  Recreate with:  x = a++ +b; -> x=a+++b; | 9/2 | 15/4 | 27/2 Temp fix: Always add space before ++a and after a++. Same for --. Not optimal. 15/4 Remove spaces in cleanup function now. |
| for (var .. in ..) not parsing | 9/2 | 9/2 | Added new terminal to parser |
| /\*comments\*/ directly after WS are lexed as Regex because of the way WS is skipped by lexer function. Not given chance to detect comment.  Recreate with:  /\*comment\*/ | 25/2 | 26/2 | Changed lexer return type to Either (Token,String) String so that we have a chance to detect comments before lexing again. |
| Literal strings do not allow underscores.  Recreate with:  "test\_123" | 26/2 | 26/2 | Modified $graphic macro in lexer to accept underscores |
| Empty functions are not accepted.  Recreate with:  function() {} | 26/2 | 26/2 | Modified sources nonterminal in Parser.y to accept epsilon |
| Floating point literal numbers are not supported.  Recreate with:  0.1 | 26/2 | 27/2 | Added support for literal bools, null, hex, oct, and floats |
| Input file cannot end with single line comment. "Expecting semi-colon or newline." Doesn't register as comment as it does not end with newline. | 27/2 | 27/2 | Made newline on single-line comments optional in lexer. |
| Assignments after += (or any other assignOp presumably) are not allowed.  Recreate with:  a += b, b = ''; | 27/2 | 27/2 | Changed type of ExprStmt to [Expression] and added non-terminals to Parser to account for this. Also modified autoSemiInsert function to accept commas. |
| Escaped quotes in strings caused errors. (Only escaped quotes that are the same as the type of quote wrapping this particlar string.)  Recreate with:  a = "\""  or a = '\'' | 27/2 | 27/2 | Added escape characters to exceptions to strings in lexer. |
| Comma operator was not implemented. In hindsight, this was the cause of the a += b, b = ''; error (2 above).  Recreate with:  while ( (a, b) == c ) {;} | 27/2 | 27/2 | Removed changes made for previous bug. Changed def of Expression in parser to accept commas as per ECMA-262 def of 'comma operator'. |
| Bracketed expressions immediately after assignments with no ending ; failed. Convoluted or what? Wouldn't have found it if not for prototype.js  Recreate with:  a = function(){}(b.c); | 27/2 | 27/2 | Modified autoSemiInsert function to check for '(' |
| Regex matcher is too greedy.  Recreate with:  a.replace(/\\/g, '/');  (Matches /\\/g, '/ as Regex)  Also a = 0.5 / 2; // Decimal  (Matches / 2; / as Regex) | 27/2 | 28/2 | Made change to regex matcher to fix first bug, then added hack to fix the comment one. See Lexer.x for details. |
| autoSemiInsert before EOF did not work. | 28/2 | 28/2 | Added EOF to autoSemiInsert checks. |
| Assignments do not support ||.  Recreate with:  a = b || function() {} | 28/2 | 28/2 | Added funcExpr to MemberExpr in parse tree, parser and codecomp. |
| Escaped hexadecimals in strings cause lexical error.  Recreate with:  a = "\xA0"; | 28/2 | 28/2 | Added hex to possible escape chars in strings. |
| Empty blocks were parsed as empty object literals instead.  Recreate with:  if (x>y) {} else {} | 28/2 | 28/2 | Added specific rule for blocks to detect empty list. It's a hack really, as the empty list should get picked up in stmtList anyway, but it's just a quirk of Haskell's pattern matching. |
| Assignment operators for shifts are not supported.  Recreate with:  >>= | 28/2 | 28/2 | Added support for >>=, <<=, >>>=, &=, ^= and |= |
| Function calls lose their brackets.  Recreate with:  alert("hello, world"); | 28/2 | 28/2 | Modified genCallExpr to add brackets to CallMember. |
| Call and New expressions did not handle argument lists properly. Missed out parentheses and commas.  Recreate with:  return new obj.thing(a,b,c); | 3/3 | 3/3 | Changed genAssignSeq to genAssignList and added () to new expr. |
| Conditional catches were missing "if".  Recreate with:  try {myTest();}  catch (e if e instanceof TypeError)... | 4/3 | 4/3 | Added emit " if " to CC pattern. |
| CallCall missed parentheses and did not add commas between args.  Recreate with:  slice.call(arguments).join(",") | 4/3 | 4/3 | Added () and changed genAssignSeq to genAssignList. Removed genAssignSeq as there is actually no case when it would be useful. |
| Semicolons after var declarations were not accounted for in the parser.  Recreate with:  var var1, var2 = 5;  Generated var var1, var2 = 5;; | 5/3 | 5/3 | Added optional ';' to pattern for varStmt in Parser.y |
| Regular expressions were still not matching properly especially when parsing already parsed files. | 5/3 | 6/3 | Modified regex following chars. Split regex into sections in the lexer for easier reading. |
| Object properties were shortened and were causing problems.  Recreate with:  var document = window.document; | 5/3 | 6/3 | Changed emitID to emit in MemberCall pattern in CC. |
| Variables which were used before being declared messed up the compressor.  Recreate with:  var jQuery = function() {  return init(rootjQuery );  }, rootjQuery; | 4/3 | 8/3 | Completely rewrote variable shrinker so that it made a full pass over the parse tree and detected all variables before generating the final code. |
|  |  |  |  |

Table . List of issues encountered during construction

## System Testing

Testing took place both during the development process and after the system was completed. There are various methods of testing available to make sure each part of the system is working correctly.

* Lexer and Parser tests: These check that the system can correctly parse any valid JavaScript program.
* Code compression tests: These check that the system can output a valid compressed JavaScript program with no source code errors.
* Equality tests: These check that the input program and the output program both produce the same results having been given the same inputs.

The first set of **Lexer and Parser tests** can be automated and with this in mind, a full test suite was built that could be run with a simple test flag. Running the command ./jshop.exe --test initiates the testing sequence which produces some information that looks something like this:

$ ./jshop.exe --test

Starting test suite

-------------------

Test number 7

Message: No message

STRUCTURE TESTS

Functions

Result: PASS

Input size: 211

Output size: 176

Reduced by: 35

Percentage of original: 83.41%

Expressions

Result: PASS

Input size: 1508

Output size: 832

Reduced by: 676

Percentage of original: 55.17%

Statements

Result: PASS

Input size: 1872

Output size: 996

Reduced by: 876

Percentage of original: 53.20%

LIBRARY TESTS

rico.js

Result: PASS

Input size: 8501

Output size: 5048

Reduced by: 3453

Percentage of original: 59.38%

prototype.js

Result: PASS

Input size: 163313

Output size: 92104

Reduced by: 71209

Percentage of original: 56.39%

jquery.js

Result: PASS

Input size: 248235

Output size: 101236

Reduced by: 146999

Percentage of original: 40.78%

dojo.js

Result: PASS

Input size: 546032

Output size: 115616

Reduced by: 430416

Percentage of original: 21.17%

AJS.4.6.js

Result: PASS

Input size: 40935

Output size: 19639

Reduced by: 21296

Percentage of original: 47.97%

Completed in 2.386 seconds

Average compression: 45.14%

This looks formidable so let’s deconstruct it and explain what is happening.

The --test flag has the additional option of a string containing a comment about the test. In this case it has not been used, hence the “Message: No message”. If it had been used (e.g. ./jshop.exe –test “This test is after removing comments and whitespace”), the message would have been printed, and the test results would have been saved to file so that they could be reviewed at a later date. As no message was included, this particular set of results will not be saved.

The tests themselves can be broken down into two sections: structure tests and library tests. Each individual test is a file that has been loaded from disk and run through the program. The **structure tests** are designed to have as many different aspects of the JavaScript language in them as possible. For instance, here is an extract of expressions.js:

// Call expression

***var*** a **=** **{** get**:** ***function*()** **{} };**

a**.**get**();**

***var*** x **=** a**[**0**];**

// Primary expressions

// Literals

***var*** a **=** ***true*;**

***var*** b **=** ***false*;**

***var*** c **=** null**;**

***var*** d **=** 10**;**

***var*** e **=** 0xFF**;**

...

The point of these tests is to make sure (within reason) that as many different combinations of statements, expressions and function declarations as possible can be successfully lexed and parsed.

The second section of tests, **library tests**, are designed to test large files and advanced features of the language. Five of the most popular JavaScript libraries were chosen as it is expected that they contain valid JavaScript and utilise many features of the language (Pilkerton). It is also very important to make sure that very large files can be successfully processed without getting stuck in loops or taking far too long.

Each individual test gives a set of statistics about the file being compressed:

jquery.js

Result: PASS

Input size: 248235

Output size: 101236

Reduced by: 146999

Percentage of original: 40.78%

The first line, the result, shows whether there were any errors during lexing or parsing. If there *is* an error, this will display FAIL and an error message will be included on the next line explaining what the error is and on which line it occurs.

The input and output sizes represent the number of characters before and after compression. The “reduced by” line shows the difference, and the percentage shows what size the compressed code is, compared to the input.

### Impact of each compression method

One of the interesting parts of this project is to discover how much of a difference each compression method makes on the code size. As mentioned above, it is possible to save test results by adding a message to the test suite flag. This was done at pivotal points in the build process when new methods were added. Using the flag --showAverages, we can see the average compression ratio at each stage:

$ ./jshop.exe --showAverages

Percentage of output to input:

Test 0 average: 57.4128 Removed comments and whitespace

Test 1 average: 56.95208 Removed semicolons before } and between } and (

Test 2 average: 56.776527 Changed literal bools to \!0 and \!1

Test 3 average: 45.653618 Shrink local variable and function names

Test 4 average: 45.619057 Removed post fix and unary spaces

Test 5 average: 45.283073 Added uppercase letters to id list; optimised

strings and numbers

Test 6 average: 45.142143 Partial evaluation of literals

As shown, the average compression ratio over all libraries and test files with all compression methods in effect is 45.14%. This sounds good, but how does it stack up to other compressors available on the Internet? To make a comparison, the jQuery library was run through several different compressors. The version of jQuery used was 1.7.1 which is 248,235 characters long in its uncompressed format. For each compressor, all available compression techniques were selected.

|  |  |  |  |
| --- | --- | --- | --- |
| Name of compressor | URL | Final size of compressed jQuery library | Percentage of original |
| Yahoo! User Interface (YUI) compressor | http://developer.yahoo.com/yui/compressor/ | 104,684 | 42.17 |
| /packer/ | http://dean.edwards.name/packer/ | 114,030 | 45.94 |
| JSMin | http://www.crockford.com/javascript/jsmin.html | 139,092 | 56.03 |
| JavaScript Haskell Optimiser (JSHOP) | N/A | 101,236 | 40.78 |

Table . Comparison of compressors with jQuery library

These results show that the program designed during this project has a better compression ratio than other, highly popular and well used systems. But that’s not much use if the resulting code doesn’t work.

This moves us nicely on to the next set of tests, **code compression tests**. These are more difficult to automate. The original plan was to run the compressed output through a JavaScript validator which would flag up any errors or bugs in the code. Such a validator does exist: JSLint[[6]](#footnote-6). However, JSLint is too meticulous and flags up many, many errors in perfectly valid programs. In fact, running the uncompressed libraries through it throws up a multitude of errors, so it is a futile task to run the compressed ones through as well.

It could be argued that if the error messages are the same for the uncompressed and compressed libraries, the semantics must be the same and the test is passed, however this is not a particularly satisfactory approach. At any rate, some of the compression methods used flag up as errors anyway, for example, JSLint requires that there be spaces on either side of an assignment operator which is actually completely unnecessary and code can be valid without these spaces.

Input:

var a=5;

Output:

*Error:*

Problem at line 1 character 6: Missing space between 'a' and '='.

var a=5;

Problem at line 1 character 7: Missing space between '=' and '5'.

var a=5;

These issues make using a validator an ineffectual exercise. In place of a fully automated test suite, it was decided that JavaScript error consoles in web browsers would be used to detect errors. Firebug in Mozilla Firefox and the JavaScript Console in Google Chrome provide error messages pertaining to JavaScript code on a webpage. These tools make it easy to find and track down errors in the script. With this in mind, two test HTML files were produced which link to the uncompressed and compressed libraries respectively.

**Uncompressed:**

<html>

<head>

<title>Original uncompressed</title>

<script type="text/javascript" src="../structure/functions.js"></script>

<script type="text/javascript" src="../structure/statements.js"></script>

<script type="text/javascript" src="../structure/expressions.js"></script>

<script type="text/javascript" src="../libraries/jquery.js"></script>

<script type="text/javascript" src="../libraries/rico.js"></script>

<script type="text/javascript" src="../libraries/prototype.js"></script>

<script type="text/javascript" src="../libraries/dojo.js"></script>

<script type="text/javascript" src="../libraries/AJS.4.6.js"></script>

</head>

<body>

</body>

</html>

**Compressed:**

<html>

<head>

<title>New compressed</title>

<script type="text/javascript"

src="../outputLibraries/functions.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/statements.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/expressions.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/jquery.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/rico.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/prototype.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/dojo.min.js"></script>

<script type="text/javascript"

src="../outputLibraries/AJS.4.6.min.js"></script>

</head>

<body>

</body>

</html>

Every time the test suite is run, the compressed library files are overwritten, and the web page can be reloaded. Any errors in the JavaScript source files will be flagged up in the browser and can then be examined closely to determine the cause of the problem.

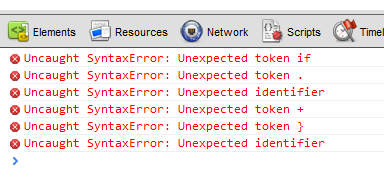


Figure . Example of errors being caught by Google Chrome's JavaScript Console

This set of tests gets us one step closer to a full testing protocol, but even though we know now that the code is valid JavaScript, we can’t be certain that it will function in exactly the same way as the original. There is no easy way of assessing these “**Equality tests**” other than to write some JavaScript code which utilises the compressed libraries and make sure it works.

Here are some examples using the compressed version of jQuery:

<html>

<head>

<title>Equality Testing</title>

<script type="text/javascript" src="../outputLibraries/jquery.min.js">

</script>

<script type="text/javascript">

$(document).ready(function() {

$("a").click(function() {

alert("Hello world!");

});

});

</script>

</head>

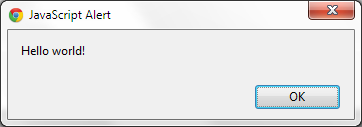
<body>

<a href="#">Test</a>

</body>

</html>

Result of clicking the link:



Another example:

<html>

<head>

<title>Equality Testing</title>

<script type="text/javascript" src="../outputLibraries/jquery.min.js">

</script>

<script type="text/javascript">

$(document).ready(function() {

$("#orderedlist li:last").hover(function() {

$(this).css("color","green");

},function(){

$(this).css("color","red");

});

});

</script>

</head>

<body>

<ol id="orderedlist">

<li>Item 1</li>

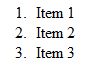
<li>Item 2</li>

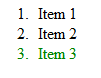
<li>Item 3</li>

</ol>

</body>

</html>

List before hovering: List while hovering: List after hovering:



These tests, although not conclusive, show that many aspects of the library work. More comprehensive testing has yet to reveal any errors.

## System Maintenance

This section is designed to make the task of maintaining the system as easy as possible. There are already several sections in this report which will provide a programmer with information about how the system is designed:

* Data flow diagrams: page 7
* Overall system design: page 15
* Process diagram: page 16
* Module descriptions: page 16
* Lexer: page 17
* Parser: page 23
* Simple optimisations: page 27
* More complex optimisations: page 31
* Partial evaluation: page 37
* Interface: page 39
* System testing: page 44

However, more information can be added to this. The source listing should provide all the information needed as it uses meaningful identifiers and has useful comments explaining each section of code. Documentation has been automatically generated using the Haddock[[7]](#footnote-7) documentation tool. This is automatically generated by the makefile (see below) with the command make doc.

### Documentation

The Haddock documentation tool produces HTML web pages showing the type signature of every public function in each module. Any comments related to that function are also included:

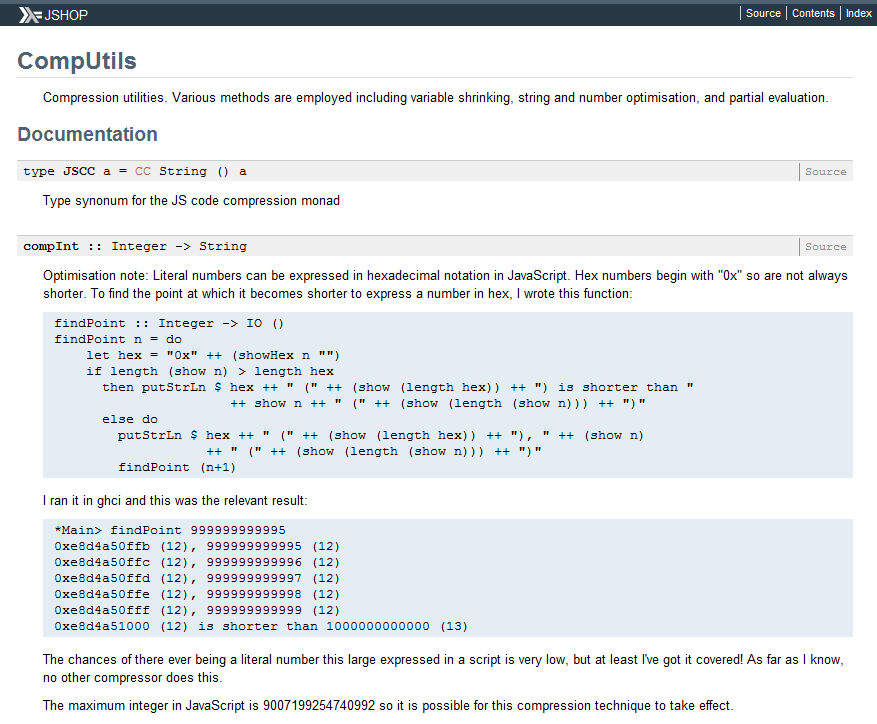


Figure . Example of Haddock documentation

The HsColour[[8]](#footnote-8) program has been used to generate syntax highlighted HTML versions of all the source code. This makes it much more readable and, due to its compatibility with Haddock, means that it can be linked to each of the function type signatures in the documentation. By clicking the “Source” link next to the function signature, the user is taken straight to the relevant function in the relevant module.

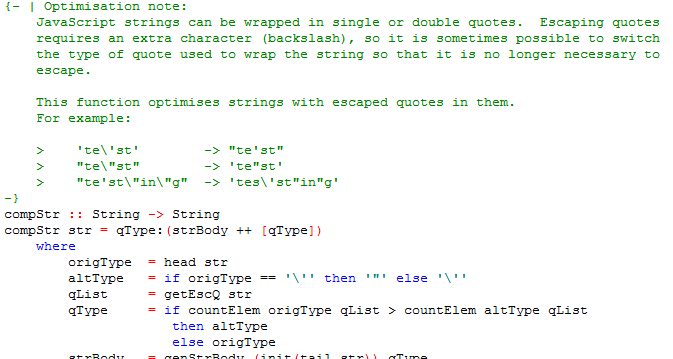


Figure . Automatically generated, syntax highlighted code

### Makefile

For use when compiling the program, a makefile has been written to take care of the more involved aspects of compilation, such as which flags to use on the various compilers. Makefiles, which use similar syntax to shell scripts, are a simple way of automatically running through a list of commands and checking which parts of the program need to be recompiled and which parts do not. For example, the lexer is compiled using Alex, but it does not have to be recompiled every time a change is made to another module.

# Alex (lexer) source files

alexFiles **=** \

Lexer.x

...

#-----------------------------------------------------------------------------

# Compile lexer:

#-----------------------------------------------------------------------------

# Options:

# g - optimise for ghc

# i - generate info file

%.hs**:** %.x

#

# Compiling lexer...

#

alex -gi $(alexFiles)

In English, if the file Lexer.x has been modified since its last compilation, the command alex -gi Lexer.x will be run.

To compile the program, the only command needed is make. This will run the makefile which will decide which parts need to be compiled and which parts do not. For instance, if only the CodeCompressor.hs module has been modified, running make will produce this result:

$ make

#

# Compiling JSHOP...

#

ghc --make -O2 -w Main -o jshop.exe

[ 6 of 14] Compiling CodeCompressor ( CodeCompressor.hs, CodeCompressor.o )

Linking jshop.exe ...

Another positive aspect of makefiles is that separate macros can be defined for specific tasks. For this project, three useful macros have been created: **doc**, **clean** and **really-clean**.

The command **make doc** will execute the following code:

#-----------------------------------------------------------------------------

# Generate Documentation:

#-----------------------------------------------------------------------------

# Options:

# HsColour:

# css - output in HTML 4.01 with CSS

# anchor - adds an anchor to every entity for use with Haddock

#

# haddock:

# odir - output directory

# html - generate documentation in HTML format

# source-base - adds link to source code directory

# source-module - adds link to each individual module

# source-entity - adds link to each individual entity

# title - title to appear at the top of each page

# w - supress warnings

doc**:** clean-doc $(haskellFiles)

#

# Generating syntax highlighted HTML source files...

#

for file in $(haskellFiles) ; do \

HsColour -css -anchor $$file > doc/src/`basename $$file .hs`.html ; \

done

#

# Generating documentation...

#

haddock --odir=doc --html --source-base=src/ --source-module=src/%M.html --source-entity=src/%M.html#%N --title="JSHOP" $(haskellFiles) w

Put simply, this starts by removing all previous documentation, then it generates syntax highlighted HTML files of every module using HsColour, and finally it generates Haddock documentation with links to each function and module provided by HsColour.

The command **make clean** will remove all .hi and .o files that are created when the Haskell modules are compiled. In addition to cleaning up the source directory, this has the side-effect that the next time make is run, every module will be recompiled.

#-----------------------------------------------------------------------------

# Cleaning:

#-----------------------------------------------------------------------------

# Remove all .hi and .o files

clean**:**

#

# Removing Haskell interfaces and objects...

#

-$(RM) $(hs\_interfaces) $(hs\_objects)

Example output:

$ make clean

#

# Removing Haskell interfaces and objects...

#

rm -f Token.hi Lexer.hi LexerMonad.hi AST.hi ParseMonad.hi Parser.hi Diagnostics.hi CodeCompMonad.hi CompUtils.hi Analyser.hi CodeCompressor.hi Utilities.hi TestSuite.hi Main.hi Token.o Lexer.o LexerMonad.o AST.o ParseMonad.o Parser.o Diagnostics.o CodeCompMonad.o CompUtils.o Analyser.o CodeCompressor.o Utilities.o TestSuite.o Main.o

The **make really-clean** command calls the clean macro, but also removes several other things. The idea is to delete everything that is not completely vital to the program.

# Remove ALL unnecessary files leaving only absolute source

really-clean**:** clean clean-doc

#

# Removing extraneous files...

#

-$(RM) Parser.hs

-$(RM) Lexer.hs

-$(RM) error.log

-$(RM) Lexer.info

-$(RM) Parser.info

-$(RM) jshop

### Making modifications

To make the source code as easy as possible to understand, each file contains a comment at the top explaining what is contained within. For example, this is at the top of the CompUtils.hs file:

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: CompUtils \*

\* Purpose: Compression utilities. Various methods are employed including \*

\* variable shrinking, string and number optimisation, and \*

\* partial evaluation. \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

When modifications are made to a file, it is useful to know which functions are available to be used. These can be viewed at the top of each module and are denoted by the import keyword:

**module** CompUtils **where**

-- Standard library imports

**import** Numeric (showHex)

**import** Maybe

-- JSHOP module imports

**import** ParseTree

**import** CodeCompMonad

In each case, it has been made clear which modules are written as part of this project, and which are third-party or built into Haskell. If an import statement has a function name or names in brackets after the name of the module (like showHex above), it means that only the specified function(s) are imported, not all.

### Naming conventions

In every part of this project, care was taken to maintain distinct continuous naming conventions for various identifier names. Keeping to a standard naming convention makes it easier to understand code and allows the reader to quickly understand what sort of identifier a certain entity is, simply by the way in which it is written. The following table shows how various identifiers are named:

|  |  |  |
| --- | --- | --- |
| Entity | Naming convention | Example |
| Module names | CamelCase with first letter capitalised as per the Haskell specification (Haskell Wiki) | CodeCompressor |
| Data types | CamelCase with first letter capitalised as per the Haskell specification | Options |
| Variable names | CamelCase with first letter in lower case as per the Haskell specification (Haskell Wiki) | origName |
| Function names | CamelCase with first letter in lower case as per the Haskell specification | parseCmdLine |

Table . Naming conventions

As long as these conventions are adhered to and the code is kept well commented, it should remain easy to understand and modify in the future.

|  |
| --- |
| Chapter 5 |

# Evaluation

This section will discuss and evaluate the outcome of the project. We will look at what the dissertation set out to achieve and how the final system compares to the original set of objectives. Possible future extensions will be explored and considered. Finally, we will reflect on the use of Haskell and the pros and cons of using a functional programming language for this project.

## Critical appraisal

When building any project based on a strict design brief, it is important to constantly refer to and reflect upon the original objectives. It is all too easy for the project to waver from the proposed path over time. Let’s take a look at the original aim of this dissertation:

“To develop a program which optimises a JavaScript file by reducing the number of characters used to express its intentions. The newly compressed JavaScript code must maintain the semantics of the original uncompressed version.” – Chapter 1, page 1

This is, purposefully, vague. It can be interpreted in many ways; however it is easy to see that the system which has been developed fulfils both criteria well. Given a suitable file (by which we mean a syntactically valid JavaScript file), the system will produce a new file of a similar or smaller size. Great care was taken to ensure that the resulting file maintains the semantics of the original. In every case, a properly parsed expression, statement, or function declaration will be reconstructed in exactly the same way that it was stated originally, perhaps with modified identifiers and more compact syntax.

### Comparison of Performance against Objectives

Each of the specific objectives will now be analysed on an individual basis. Examples of input and output will be shown where possible and appropriate.

*Input, processing and output requirements*

The system must:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Description | System performance in this area | Example input | Example output |
| 1 | Accept a JavaScript file as an argument on the command line | The system allows the user to enter a filename as an argument | ./jshop.exe filename.js | System runs as expected |
| 2 | Translate the contents of that file into a form it can understand and modify without further user interaction | The system lexes and parses the contents of the given file and creates a parse tree with all the program information | function test(a, b) {} | Parse tree:  SFuncDecl (FuncDecl (Just "test") ["a","b"] []) |
| 3a | Remove single-line comments from the source code | Single line comments are successfully removed | // Comment  var x = 5; | var x=5; |
| 3b | Remove multi-line comments from the source code | Multi-line comments are successfully removed | /\* This is a  multi-line  comment \*/ var y = 10; | var y=10; |
| 3c | Remove unnecessary whitespace from the source code | Whitespace, newlines and tabs are all removed unless needed | switch (x) {  case 1:  alert(“Win”);  break;  default:  alert(“Lose”);  break;  } | switch(x){case 1:alert("Win");break;default:alert("Lose");break} |
| 3d | Remove semi-colons before closing braces from the source code | All semi-colons directly before closing braces are removed | function test() {  alert(“Test1”);  } | function test(){alert("Test1")} |
| 4 | Shorten identifiers where possible | Local variable and function names are shortened | function test1(var1, var2) {  var var3 = "hello";  var4 = 3; // Global    function test2(yay, yay1) {  return var3;  }  function test3(woo, woo1) {  return woo1 + var4;  }  } | function test1(a,b){var c="hello";var4=3;function d(e,f){return c}function e(f,g){return g+var4}} |
| 5a | Convert ternary conditionals to their shorthand equivalent | Suitable if statements are converted to ternary | if (5 > 3) {  result = "true";  }  else {  result = "false"  } | result=5>3?"true":"false"; |
| 5b | Convert array declarations to their shorthand equivalent | Array declarations are shortened | var a = new Array; | var a=[]; |
| 5c | Convert object declarations to their shorthand equivalent | Object declarations are shortened | var b = new Object; | var b={}; |
| 6 | Perform partial evaluation on expressions and statements where possible | Literal calculations and string concatenations are partially evaluated | a = 23 + 43;  b = 34.65 - 234.4 + 1200.0;  c = 1 + 2 \* 3 / 4;  d = 'Hello' + ' World';  e = "Goodbye" + " World";  f = "He\"l\"lo"  + ' Wo\'r\'ld'; | a=66;b=1000.25;c=2.5;  d='Hello World';  e="Goodbye World";  f="He\"l\"lo Wo'r'ld"; |
| 7 | Return the compressed JavaScript to the user along with compression statistics | The results are displayed along with the compression ratio and execution time | ./jshop.exe tests/test.js  File contains the switch statement from the example in 3c. | $ ./jshop.exe tests/test.js  OUTPUT:  switch(x){case 1:alert("Win");break;default:alert("Lose");break}  STATS:  Reduced by 30 chars, 68.08% of original.  Total execution time: 0.000 secs |

Table . Comparison or performance against objectives

*Performance requirements*

1. The system must be simple to operate through a command line interface.

An interface utilising argument flags has been implemented. The --help flag displays information explaining how to use the interface. See the Interface section in Chapter 3, page 39.

1. The system must return the results in a timely manner.

The system completes processing very swiftly, rarely taking more than a second to fully compress a file. When the test suite is run (comprising a total of over 30,000 lines of code), the execution time is roughly 2.5 seconds. This is on a fast, quad-core PC, however even on a slower machine, it is reasonable to assume it would complete in less than a minute. This constitutes a “timely manner.”

Having considered every objective in detail, it is clear that the completed system meets each point successfully. Extra compression methods were even added during the build process. These can be found in the Micro Optimisations section in Chapter X, page 29. However, there are several possible extensions which could be addressed in future.

## Possible Extensions

In a project like this, there are several possible types of extension. Perhaps the most obvious would be to develop more compression methods. As discussed in the “Areas considered for future work” section of Chapter 2 (page 9), common sub-expression elimination would be the next most obvious method to implement. This involves identifying repeated code blocks and calculating them separately so that they can be referred to at a later date. The example given in Chapter 2 can be seen here:

a = b \* c + g;

d = b \* c \* d;

becomes

tmp = b \* c;

a = tmp + g;

d = tmp \* d;

Another totally different type of extension, however, would be to create an online tool which allows people to upload JavaScript files or directly enter the code into a textbox. The input would then be run through the program and the output would be given back to the user. There are many implementations of this type of system already in existence. For example, Dean Edwards’ /packer/ compressor can be used in this way. (Edwards)

## What was learned about Haskell

The use of Haskell in this project has been very educational. Functional programming promotes a whole new way of thinking about developing systems. In imperative programming, it is easy to write programs in a very sequential manner. When designing a quicksort algorithm in C, the first thing most programmers would do would be to create variables to hold pointers and counters. In functional programming, you think more about the goal of the algorithm you are writing. What do we want to end up with? The type system is an integral part of promoting this way of thinking. By writing the exact type of the function first, it becomes clear in your mind where you are coming from and where you are going to. Expanding on the quicksort example, the following type signature could be used:

qsort :: Ord a => [a] -> [a]

This tells us that we are converting a list of orderable things into another list of orderable things of the same type. Once we know how the quicksort algorithm works, it is simply a matter of writing it out. In C, this is a complicated task, but not so in Haskell! The algorithm works by recursively placing the next number in the middle of the current section and inserting all sorted lower numbers before it; and all sorted higher numbers above it. In Haskell this can be written pretty much as is.

qsort [] = []

qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger

**where**

smaller = [a | a <- xs, a <= x]

larger = [b | b <- xs, b > x]

This example was taken from Programming in Haskell (Hutton, 2007, pp. 8,9).

In the context of the system developed for this dissertation, this goal-oriented approach was very suitable. A good example would be the algorithm designed to generate a list of all possible identifiers.

genIDList :: [String]

genIDList = [c:s | s <- "":allStrings, c <- firstChar] \\ reservedIDs

**where**

firstChar = ['a'..'z']++['A'..'Z']++['\_']

alph = ['a'..'z']++['A'..'Z']++['0'..'9']++['$','\_']

allStrings = [c:s | s <- "":allStrings, c <- alph]

reservedIDs = ["if","in","do","int","for","new","try","var"]

This particular function makes good use of Haskell’s list comprehensions and lazy evaluation, two more useful features of the language. Again, in an imperative language like C or Java, a function like this would be complicated to define.

The space and time complexities of the system are very good. This is partly due to the optimisation capabilities of the Glasgow Haskell Compiler, and partly due to the design of the parser. Although it cannot be stated with certainty unless an equivalent system was designed, it seems likely that if the entire project was created in an imperative language, it would not be as efficient. Many of the optimisations make good use of pattern matching which is less simple to implement in a standard imperative language.

Although at times it felt like the strict type system got in the way of progress, and compile errors were a regular occurrence, it must be noted that run-time errors have been conspicuous by their absence. By defining a strict type signature for a function, the compiler realises immediately if the main body of the function will ever produce a result that does not abide by the type rules. Once this has been detected and fixed, it makes the likelihood of an error occurring much lower.

Overall, the type system is of great benefit and, if used correctly, almost guarantees a working program. If an imperative language had been used on a project this large, many more errors could be expected.

## Conclusion

Based on the objectives, this project has been a success. Each optimisation method has been implemented and has been proven to reduce the size of the input code. The system is simple to invoke and understand. It runs smoothly and quickly and all known bugs have been removed.

The process of building the system has also been a success. The prototyping stage flagged up the main issues in the project, i.e. the process of converting the input into a malleable, useable state. Once this was recognised and a parser was built, the process of building in optimisations was straightforward and logical.

The testing phase threw up one or two challenges but these were overcome and resulted in a much sturdier system.

The use of Haskell has resulted in a unique project which utilises intelligent, mathematical solutions and techniques to develop a good quality JavaScript compressor which has achieved a better compression ratio than many of the most popular online minifiers.

|  |
| --- |
| Chapter 6 |

# Bibliography

ECMA International. (n.d.). *ECMA-262 Official Specification*. Retrieved February 2012, from ECMA International: http://www.ecma-international.org/publications/standards/Ecma-262.htm

Edwards, D. (n.d.). */packer/*. Retrieved May 1, 2012, from http://dean.edwards.name/packer/

Flanagan, D. (2001). *JavaScript: The Definitive Guide* (4th ed.). Canada: O'Reilly.

Flanagan, D. (2002). *JavaScript Pocket Reference* (2nd ed.). Canada: O'Reilly.

Haskell Wiki. (n.d.). *Programming guidelines*. Retrieved April 28, 2012, from Haskell Wiki: http://www.haskell.org/haskellwiki/Programming\_guidelines#Naming\_Conventions

Hutton, G. (2007). *Programming in Haskell.* Cambridge University Press.

Nilsson, D. H. (2006-2011). *Haskell MiniTriangle Compiler*. Retrieved April 28, 2012, from http://www.cs.nott.ac.uk/~nhn/G53CMP/

Nilsson, D. N. (2011). A Complete (Albeit Small) Compiler. *G53CMP Lectures 2&3*. Nottingham, UK.

Pilkerton, B. (n.d.). *JavaScript Libraries*. Retrieved April 28, 2012, from http://javascriptlibraries.com/

Scott González et al. (n.d.). *jQuery.com*. Retrieved February 22, 2012, from jQuery: http://jquery.com/

|  |
| --- |
| Chapter 7 |

# Appendix

## Example Input and Output

1. Navigate to the directory containing the file jshop.exe
2. Type the following command where file.js is the path to a file with JavaScript in it:

./jshop.exe file.js

1. The following results will be given:

$ ./jshop.exe file.js

OUTPUT:

[Compressed JavaScript will go here]

STATS:

Reduced by xx chars, xx% of original.

Total execution time: xx secs

1. To find more information about how to use the program, type the following command:

./jshop.exe --help

## Code Listing

The following pages show the source code for this project. It is annotated throughout with comments to explain the more complex algorithms. The modules are arranged in what seems the most appropriate order, from tokenising the original JavaScript code, right through to generating the compressed output.

Table of Contents

Main.hs ………………………………………………………………………………………………... 65

Token.hs ………………………………………………………………………………………………. 70

Lexer.x ………………………………………………………………………………………………..... 71

LexerMonad.hs ………………………………………………………………………………………... 73

ParseMonad.hs ………………………………………………………………………………………... 75

ParseTree.hs …………………………………………………………………………………………... 76

Parser.y ……………………….………………………………………………………………………... 81

Diagnostics.hs …………………………………………………………………………………………. 89

CodeCompMonad.hs …………………………………………………………………………………. 90

Analyser.hs ………………….………………………………………………………………………… 95

CodeCompressor.hs ……………….………………………………………………………………….. 101

CompUtils.hs ………………..………………………………………………………………………… 115

Utilities.hs ………………….………………………………………………………………………….. 119

TestSuite.hs ……………………………………………………………………………………………. 123

makefile ………………….…………………………………………………………………………….. 127

Prototype …………………..…………………………………………………………………………... 130

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: Main \*

\* Purpose: Main JSHOP module \*

\* Author: Nick Brunt \*

\* \*

\* Based (loosely) on the HMTC equivalent \*

\* Copyright (c) Henrik Nilsson, 2006 - 2011 \*

\* <http://www.cs.nott.ac.uk/~nhn/> \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

module Main where

-- Standard library imports

import System

import System.CPUTime

import System.IO

import Maybe

-- JSHOP module imports

import Token

import Lexer

import LexerMonad

import ParseTree

import ParseMonad

import Parser

import Diagnostics

import CodeCompMonad

import CodeCompressor

import Utilities

import TestSuite

data Options =

Options {

optHelp :: Bool,

optVer :: Bool,

optInput :: Bool,

optTokens :: Bool,

optTree :: Bool,

optLState :: Bool,

optAll :: Bool,

optBloat :: Bool,

optPP :: Bool,

optTest :: Bool,

optST :: Bool,

optSAT :: Bool,

optSA :: Bool

}

deriving Show

defaultOptions :: Options

defaultOptions =

Options {

optHelp = False,

optVer = False,

optInput = False,

optTokens = False,

optTree = False,

optLState = False,

optAll = False,

optBloat = False,

optPP = False,

optTest = False,

optST = False,

optSAT = False,

optSA = False

}

version :: String

version = "1.0"

prompt :: String

prompt = "Please enter some JavaScript here (Terminate with Ctrl+D in UNIX, or Ctrl+Z in Windows, followed by Return. Must be on a new line.):"

------------------------------------------------------------------------

-- Main

------------------------------------------------------------------------

main :: IO()

main = do

hSetEncoding stdout utf8

hSetEncoding stdin utf8

hSetEncoding stderr utf8

startTime <- getCPUTime

(opts, mbArgs) <- parseCmdLine

if optHelp opts then

printHelp

else if optVer opts then

printVersion

else if optTest opts then

runTests mbArgs

else if optST opts then

showResult $ read $ head $ fromJust mbArgs

else if optSAT opts then

showPastResults

else if optSA opts then

showAverages

else do

-- Get input (from file if given)

input <-

case mbArgs of

Nothing -> putStrLn prompt >> getContents

Just [file] -> readFile file

Just (f:fs) -> readFile f -- Potential support for

-- multiple files?

execute opts input

execTime startTime

------------------------------------------------------------------------

-- Compressor

------------------------------------------------------------------------

execute :: Options -> String -> IO()

execute opts input = do

-- Parse

let parseOutput = parseJS input

case parseOutput of

Left error -> do

putStrLn $ show error

Right (tree, state) -> do

-- Display input

if optInput opts || optAll opts then do

putStrLn "\n\nINPUT:"

putStrLn input

else

putStr "" -- Do nothing

-- Display tokens

if optTokens opts || optAll opts then do

putStrLn "\n\nTOKENS:"

nonMLexer input

else

putStr "" -- Do nothing

-- Display Parse Tree

if optTree opts || optAll opts then do

putStrLn "\n\nPARSE TREE:"

putStrLn $ ppTree tree (optBloat opts)

return()

else

putStr "" -- Do nothing

-- Display Lexer State

if optLState opts || optAll opts then do

putStrLn "\n\nSTATE:"

putStrLn $ show state

else

putStr "" -- Do nothing

-- Display output

putStrLn "\n\nOUTPUT:"

let output = genJS tree

if optPP opts

then putStrLn $ ppOutput output ""

else putStrLn output

-- Display stats

putStrLn "\n\nSTATS:"

putStrLn $ showRatio input output

------------------------------------------------------------------------

-- Parse command line arguments

------------------------------------------------------------------------

parseCmdLine :: IO (Options, Maybe [String])

parseCmdLine = do

args <- getArgs

(opts, mbFiles) <- processOptions defaultOptions args

return (opts, mbFiles)

processOptions :: Options -> [String] -> IO (Options, Maybe [String])

processOptions opts args = do

(opts', mbArgs) <- posAux opts args

return (opts', mbArgs)

where

posAux :: Options -> [String] -> IO (Options, Maybe [String])

-- No arguments left

posAux opts [] = return (opts, Nothing)

posAux opts arguments@(arg:args)

-- No options, just other args

| take 2 arg /= "--" = return (opts, Just arguments)

-- Options

| otherwise = do

-- Process option (dropping the --)

opts' <- poAux opts (drop 2 arg)

-- Move on to next option

posAux opts' args

poAux :: Options -> String -> IO Options

poAux opts o

| o == "help" || o == "h" =

return (opts {optHelp = True})

| o == "version" || o == "ver" || o == "v" =

return (opts {optVer = True})

| o == "input" || o == "i" =

return (opts {optInput = True})

| o == "tokens" || o == "tok" =

return (opts {optTokens = True})

| o == "tree" =

return (opts {optTree = True})

| o == "lstate" =

return (opts {optLState = True})

| o == "all" =

return (opts {optAll = True})

| o == "rembloat" =

return (opts {optBloat = True})

| o == "prettyprint" || o == "pp" =

return (opts {optPP = True})

| o == "test" || o == "t" =

return (opts {optTest = True})

| o == "showTest" =

return (opts {optST = True})

| o == "showAllTests" =

return (opts {optSAT = True})

| o == "showAverages" =

return (opts {optSA = True})

| otherwise = do

putStrLn ("Unknown option \"--" ++ o ++ "\"")

return opts

------------------------------------------------------------------------

-- Miscellaneous output

------------------------------------------------------------------------

printHelp :: IO()

printHelp = putStr

"USAGE:\n\

\ jshop [options] file.js Compress \"file.js\"\n\

\ jshop [options] Read input from standard input.\n\

\ (Terminate with Ctrl+D, Enter in UNIX, or Ctrl+Z,\n\

\ Enter in Windows. Must be on a new line.)\n\

\\n\

\DEFAULT OUTPUT:\n\

\ Compressed JS and ratio of input to output.\n\

\\n\

\OPTIONS:\n\

\ Output\n\

\ --help, --h\n\

\ Print help message and stop.\n\n\

\ --version, --ver, --v\n\

\ Print JSHOP version and stop.\n\n\

\ --input, --i\n\

\ Print input.\n\n\

\ --tokens, --tok\n\

\ Print list of tokens.\n\n\

\ --tree\n\

\ Print the Parse tree.\n\n\

\ --lstate\n\

\ Print the Lexer state.\n\n\

\ --all\n\

\ Print input, tokens, Parse tree, Lexer state, output and ratio.\n\n\

\ --rembloat\n\

\ Experimental. Removes bloat from the Parse tree. Can make it unreadable.\n\

\ Only works with --tree or --all.\n\n\

\ --prettyprint, --pp\n\

\ Experimental. Pretty prints the output for ease of reading.\n\n\

\ Testing\n\

\ --test --t [\"message\"]\n\

\ Run full test suite and stop. Message is optional. If added, test\n\

\ results will be saved to file.\n\n\

\ --showAverages\n\

\ Displays the average compression ratios for all past tests.\n\n\

\ --showTest NUM\n\

\ Displays a past test of index NUM.\n\n\

\ --showAllTests\n\

\ Displays all past tests.\n\n\

\"

printVersion :: IO()

printVersion = do

putStrLn "\nJavaScript Haskell Optimiser (JSHOP)"

putStrLn $ "Version " ++ version

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: Token \*

\* Purpose: Representation of tokens (lexical symbols) \*

\* Authors: Nick Brunt, Henrik Nilsson \*

\* \*

\* Based on the HMTC equivalent \*

\* Copyright (c) Henrik Nilsson, 2006 - 2011 \*

\* <http://www.cs.nott.ac.uk/~nhn/> \*

\* \*

\* Revisions for JavaScript \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Representation of tokens (lexical symbols).

module Token where

-- | Token type.

data Token

= WS -- ^ Whitespace

| SLCom -- ^ Single line comment (necessary for line counting - discarded after lexer)

| LitInt Integer -- ^ Integer literals

| LitFloat Double -- ^ Float literals (Using Haskell's Double for safety)

| LitStr String -- ^ String literals

| Id String -- ^ Identifiers

| Regex String -- ^ Regular expressions

| ResId String -- ^ Reserved identifier

| ResOp String -- ^ Reserved operator

| Other String -- ^ Unknown other symbol

| EOF -- ^ End of file (input) marker

deriving (Eq, Show)

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- \* JSHOP \*

-- \* \*

-- \* Module: Lexer \*

-- \* Purpose: JavaScript Lexical Analyser \*

-- \* Author: Nick Brunt \*

-- \* \*

-- \* Copyright (c) Nick Brunt, 2011 - 2012 \*

-- \* \*

-- \* The structure for this file was partially \*

-- \* determined from a Haskell lexer: \*

-- \* http://darcs.haskell.org/alex/examples/haskell.x \*

-- \* \*

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

{

-- | JavaScript Lexical Analyser generated by Alex

**module** Lexer **where**

-- JSHOP module imports

**import** Token

}

%wrapper "basic"

-- Special characters

$whitechar = [ \t\n\r\f\v]

$spacechar = [ \t]

$special = [\(\)\,\;\[\]\`\{\}]

$digit = 0-9

$alpha = [a-zA-Z]

-- Symbols are any of the following characters except (#) some special cases

$symbol = [\!\#\$\%\&\\*\+\.\/\<\=\>\?\@\\\^\|\-\~\,\;] # [$special \**\_**\:\"\']

$graphic = [$alpha $symbol $digit $special \**\_**\:\"\'\,]

$octit = 0-7

$hexit = [0-9 A-F a-f]

$nl = [\n\r]

$charesc = [abfnrtv\\\"\'\&\/]

@escape = \\ ( $charesc | x $hexit+ )

@dString = $graphic # [\"\\] | " " | $nl | @escape

@sString = $graphic # [\'\\] | " " | $nl | @escape

-- As stated in JavaScript Pocket Reference (O'Reilly, David Flanagan, 2nd edition), page 3

@reservedid =

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|

-- reserved words for possible future extensions

abstract|boolean|byte|char|**class**|const|debugger|double|enum|export|extends|final|

float|implements|**import**|int|interface|long|native|package|private|protected|public|

short|static|super|synchronized|throws|transient|volatile|

-- and finally, let's hope not

goto

-- As stated in JavaScript Pocket Reference (O'Reilly, David Flanagan, 2nd edition), pages 10 and 11

@reservedop =

"." | "[" | "]" | "(" | ")" | "++" | "--" | "-" | "+" | "~" | "!" | "\*" | "/" | "%" |

"<<" | ">>" | ">>>" | "<" | "<=" | ">" | ">=" | "==" | "!=" | "===" | "!==" | "&" |

"^" | "|" | "&&" | "||" | "?" | ":" | "=" | "\*=" | "+=" | "-=" | "/=" | "%=" | "<<=" |

">>=" | ">>=" | "&=" | "^=" | "|=" | "," | ";" | "{" | "}"

@decimal = $digit+

@float = $digit\* "." $digit+ ((e|E) ("+"|"-")? $digit+)?

@hex = "0" ("x"|"X") $hexit+

-- Some versions of JavaScript support octals, some do not.

-- http://docstore.mik.ua/orelly/webprog/jscript/ch03\_01.htm#jscript4-CHP-3-SECT-1

@oct = "0" $octit+

-- "Identifiers are composed of any number of letters and digits, and \_ and $ characters. The

-- first character of an identifier must not be a digit, however."

-- From JavaScript Pocket Reference (O'Reilly, David Flanagan, 2nd edition), page 2

$firstLetter = [$alpha \**\_** \$]

@id = $firstLetter [$alpha $digit \**\_** \$]\*

-- This regular expression matches a JavaScript regular expression. There were several

-- problems with matching regexs because you cannot specify context in the lexer. I have

-- worked around this by specifying what the character following the regex is allowed to be

-- (see $reFollow). This means I am matching one too many characters, so I have to correct

-- this in the lexer function (found in LexerMonad.hs). This is the only solution available

-- beyond writing a whole lexer dedicated to regular expressions.

@reEscapedChar = \\.

@reCharClass = \[[^\]]\*\]

@reBody = @reEscapedChar | [^\[\/\\] | @reCharClass

@reMods = "g" | "i" | "m"

$reFollow = [\)\,\;\ \.\} $nl]

@regex = \/ @reBody\* \/ @reMods\* $reFollow

-- Equivalent to \/(\\.|[^\[\/\\]|\[[^\]]\*\])\*\/[gim]\*[\)\,\;\ \.\} $nl]

-- Capture any other unknown symbols

@other = $symbol

-- String -> Token

tokens :-

<0> "//" [$spacechar $printable]\* $nl? { \s -> SLCom }

<0> $white+ { \s -> WS }

<0> @reservedid { \s -> ResId s }

<0> @reservedop { \s -> ResOp s }

-- JavaScipt floats do not require zeros before the decimal point. Haskell's floats do.

<0> @float { \s -> LitFloat (read ("0" ++ s) :: Double) }

-- Read hex's and oct's in as integers. Much easier to handle.

<0> @hex { \s -> LitInt (read s :: Integer) }

-- JavaScript notation for octals is 0 followed by [0..7]+, whereas Haskell's notation

-- is 0o followed by [0..7]+.

<0> @oct { \s -> LitInt (read ("0o" ++ tail s) :: Integer) }

<0> @decimal { \s -> LitInt (read s :: Integer) }

<0> @id { \s -> Id s }

<0> \" @dString\* \" { \s -> LitStr s }

<0> \' @sString\* \' { \s -> LitStr s }

<0> @regex { \s -> Regex s }

<0> @other { \s -> Other s }

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: LexerMonad \*

\* Purpose: Monadic wrapper for the Alex lexer \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\* Adapted from <http://www.haskell.org/alex/doc/html/wrappers.html> \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Monadic wrapper for the Alex lexer

module LexerMonad (

lexer, -- String -> Either (Token,String) String

monadicLexer, -- (Token -> P a) -> P a

autoSemiInsert -- autoSemiInsert :: Token -> a -> P a

) where

-- Standard library imports

import Data.Char

import qualified Data.ByteString.Char8 as BS

import Control.Monad.State

import Control.Monad.Error

-- JSHOP module imports

import Token

import ParseMonad

import Lexer

-- | Main lexer function. Uses state monad to store current state of the lexer.

monadicLexer :: (Token -> P a) -> P a

monadicLexer cont = do

chr <- get

put chr { nl = False }

monadicLexer' cont

-- | Catches newlines and multi-line comments. Single-line comments are caught in the lexer.

monadicLexer' :: (Token -> P a) -> P a

monadicLexer' cont = do

chr <- get

case (rest chr) of

('\n':xs) -> do

put chr {rest = xs, lineno = (lineno chr) + 1, nl = True}

monadicLexer' cont

('/':'\*':xs) -> do

put chr {rest = xs}

scanComment cont

\_ -> do

let lexResult = lexer (rest chr)

case lexResult of

-- Specifically check for single line comments. Increment line number then skip.

Left (SLCom, xs) -> do

put chr {rest = xs, lineno = (lineno chr) + 1, nl = True}

monadicLexer' cont

Left (token, xs) -> do

put chr {rest = xs, rest2 = (rest chr), lastToken = Just token}

cont token

Right xs -> do

put chr {rest = xs}

monadicLexer' cont

-- | Scans to the end of a multiline comment. Could have used a regex in

-- the lexer but we want to count lines.

scanComment :: (Token -> P a) -> P a

scanComment cont = do

chr <- get

case (rest chr) of

('\n':xs) -> do

put chr {rest = xs, lineno = (lineno chr) + 1}

scanComment cont

('\*':'/':xs) -> do

put chr {rest = xs}

monadicLexer cont

(\_:xs) -> do

put chr {rest = xs}

scanComment cont

-- | Given a string, returns a tuple containing the token of the first item in the

-- string and the remaining string.

lexer :: String -- ^ The remaining input

-> Either (Token,String) String -- ^ Either the token and the remaining string

-- or just the remaining string if the token is

-- to be skipped

lexer input = go ('\n', input)

where

go inp@(\_,rem) =

case alexScan inp 0 of

AlexEOF -> Left (EOF, [])

AlexError (c,cs) -> error ("Lexical error at " ++ take 50 cs)

AlexSkip inp' len -> go inp'

AlexToken inp'@(x,xs) len act -> case act (take len rem) of

WS -> Right xs -- Skip whitespace

-- Regexs match one char too many (see note in Lexer.x) so this corrects it.

Regex s -> Left (Regex (init s), (last s):xs)

token -> Left (token, xs)

{- |

Handle automatic semicolon insertion. This will get passed the current

lookahead token which will get discard so even if we have found a ';' we

need to put it back onto the stream so that it gets found again.

See ECMA-262 documentation page 21

-}

autoSemiInsert :: Token -> a -> P a

autoSemiInsert token res = do

s <- get

if token == (ResOp ";") -- Next token is a ; anyway, so no need to add another one

|| token == (ResOp "}") -- Next token is a }, so a ; is not necessary, but add one anyway to

-- keep the parser happy (pun intended)

|| token == (ResOp "(") -- Next token is a ), so this must have been an exprStmt, so add a ;

|| token == EOF -- We're at the end of the file, add a ;

|| (nl s)

then do

let r = if token == (ResOp ";") then rest2 s

else (';':(rest2 s))

put s { rest = r }

return res

else throwError $ "Expecting Semi or NL: " ++

"lineno = " ++ show (lineno s) ++

", token = " ++ show token ++

", " ++ take 50 (show s)

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: ParseMonad \*

\* Purpose: Monad for storing Parser state \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Monad for storing Parser state

module ParseMonad where

-- Standard library imports

import Control.Monad.Identity

import Control.Monad.Error

import Control.Monad.State

-- JSHOP module imports

import Token

-- | Lexer state

data LexerState

= LS {

rest :: String, -- ^ The remaining input

lineno :: Int, -- ^ Current line number

nl :: Bool, -- ^ Newline flag

rest2 :: String, -- ^ For use with automatic semicolon insertion

lastToken :: (Maybe Token) -- ^ The token just lexed

}

deriving Show

-- | The initial state of the lexer

startState :: String -> LexerState

startState str

= LS {

rest = str,

lineno = 1,

nl = False,

rest2 = "",

lastToken = Nothing

}

-- | Parser monad incorporating the lexer state

type P = StateT LexerState (ErrorT String Identity)

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: ParseTree \*

\* Purpose: JavaScript Parse Tree \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\* As defined in ECMA-262 \*

\* \*

\* Expressions - Page 40 \*

\* Statements - Page 61 \*

\* Function Definition - Page 71 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | JavaScript Parse Tree. Representation of JavaScript programs after parsing.

module ParseTree where

-- | The Parse Tree is made up of a list of sources.

data Tree

= Tree [Source]

deriving (Show, Eq)

-- | A source element can either be a statement or a function declaration.

data Source

= Statement Statement

| SFuncDecl FuncDecl

deriving (Show, Eq)

-- | A function declation signature can contain a name and a list of args.

-- The body of the function is a list of sources.

data FuncDecl

= FuncDecl (Maybe String) [String] [Source]

deriving (Show, Eq)

-- | A statement does not return a value and usually ends with a semicolon.

data Statement

= EmptyStmt -- ^ A single semicolon

| IfStmt IfStmt -- ^ If statement

| IterativeStmt IterativeStmt -- ^ Iterative statement (while, for etc.)

| ExprStmt Expression -- ^ Expression followed by a semicolon

| Block [Statement] -- ^ List of statements ({...})

| VarStmt [VarDecl] -- ^ Variable declaration (var ...)

| TryStmt TryStmt -- ^ Try catch finally

| ContinueStmt (Maybe String) -- ^ Continue statement

| BreakStmt (Maybe String) -- ^ Break statement

| ReturnStmt (Maybe Expression) -- ^ Return statement

| WithStmt Expression Statement -- ^ With statement

| LabelledStmt String Statement -- ^ Labelled statement

| Switch Switch -- ^ Switch statement

| ThrowExpr Expression -- ^ Throw an exception

deriving (Show, Eq)

-- | An if statement with an optional else branch

data IfStmt

= IfElse Expression Statement Statement

| If Expression Statement

deriving (Show, Eq)

-- | Itertive statements (loops)

data IterativeStmt

= DoWhile Statement Expression

| While Expression Statement

| For (Maybe Expression) (Maybe Expression) (Maybe Expression) Statement

| ForVar [VarDecl] (Maybe Expression) (Maybe Expression) Statement

| ForIn LeftExpr Expression Statement

| ForVarIn [VarDecl] Expression Statement

deriving (Show, Eq)

-- | Try statement

data TryStmt

= TryBC [Statement] [Catch]

| TryBF [Statement] [Statement]

| TryBCF [Statement] [Catch] [Statement]

deriving (Show, Eq)

-- | Catch

data Catch

= Catch String [Statement]

| CatchIf String [Statement] Expression

deriving (Show, Eq)

-- | Switch statement

data Switch

= SSwitch Expression CaseBlock

deriving (Show, Eq)

-- | A block of cases within a switch statement

data CaseBlock

= CaseBlock [CaseClause] [DefaultClause] [CaseClause]

deriving (Show, Eq)

-- | An individual case clause

data CaseClause

= CaseClause Expression [Statement]

deriving (Show, Eq)

-- | The default clause in a switch statement

data DefaultClause

= DefaultClause [Statement]

deriving (Show, Eq)

-- | An expression returns a value

data Expression

= Assignment [Assignment]

deriving (Show, Eq)

-- | A variable declaration is made up of the identifier, and a possible assignment

data VarDecl

= VarDecl String (Maybe Assignment)

deriving (Show, Eq)

-- | An assignment can be an expression or a function

data Assignment

= CondExpr CondExpr

| Assign LeftExpr AssignOp Assignment

| AssignFuncDecl FuncDecl

deriving (Show, Eq)

-- | Left expression

data LeftExpr

= NewExpr NewExpr

| CallExpr CallExpr

deriving (Show, Eq)

-- | Assignment operators

data AssignOp

= AssignNormal

| AssignOpMult

| AssignOpDiv

| AssignOpMod

| AssignOpPlus

| AssignOpMinus

| AssignOpSLeft

| AssignOpSRight

| AssignOpSRight2

| AssignOpAnd

| AssignOpNot

| AssignOpOr

deriving (Show, Eq)

-- | Conditional expression (...?...:...)

data CondExpr

= LogOr LogOr

| CondIf LogOr Assignment Assignment

deriving (Show, Eq)

-- | New expression

data NewExpr

= MemberExpr MemberExpr

| NNewExpr NewExpr

deriving (Show, Eq)

-- | Call expression

data CallExpr

= CallMember MemberExpr [Assignment]

| CallCall CallExpr [Assignment]

| CallSquare CallExpr Expression

| CallDot CallExpr String

deriving (Show, Eq)

-- | Member expression

data MemberExpr

= MemExpression PrimaryExpr

| FuncExpr FuncDecl

| ArrayExpr MemberExpr Expression

| MemberNew MemberExpr [Assignment]

| MemberCall MemberExpr String

deriving (Show, Eq)

-- | Primary expressions

data PrimaryExpr

= ExpLiteral Literal

| ExpId String

| ExpBrackExp Expression

| ExpThis

| ExpRegex String

| ExpArray ArrayLit

| ExpObject [(PropName, Assignment)]

deriving (Show, Eq)

-- | Literals

data Literal

= LNull -- ^ null

| LBool Bool -- ^ true or false

| LInt Integer

| LFloat Double

| LStr String -- ^ \"string\" or \'string\'

deriving (Show, Eq)

-- | Array literals

data ArrayLit

= ArraySimp [Assignment]

deriving (Show, Eq)

-- | Property names

data PropName

= PropNameId String

| PropNameStr String

| PropNameInt Integer

deriving (Show, Eq)

-- | Logical or

data LogOr

= LogAnd LogAnd

| LOLogOr LogOr LogAnd -- ^ ||

deriving (Show, Eq)

-- | Logical and

data LogAnd

= BitOR BitOR

| LALogAnd LogAnd BitOR -- ^ &&

deriving (Show, Eq)

-- | Bitwise or

data BitOR

= BitXOR BitXOR

| BOBitOR BitOR BitXOR -- ^ |

deriving (Show, Eq)

-- | Bitwise xor

data BitXOR

= BitAnd BitAnd

| BXBitXOR BitXOR BitAnd -- ^ ^

deriving (Show, Eq)

-- | Bitwise and

data BitAnd

= EqualExpr EqualExpr

| BABitAnd BitAnd EqualExpr -- ^ &

deriving (Show, Eq)

-- | Equals expressions

data EqualExpr

= RelExpr RelExpr

| Equal EqualExpr RelExpr -- ^ ==

| NotEqual EqualExpr RelExpr -- ^ !=

| EqualTo EqualExpr RelExpr -- ^ ===

| NotEqualTo EqualExpr RelExpr -- ^ !==

deriving (Show, Eq)

-- | Relative expressions

data RelExpr

= ShiftExpr ShiftExpr

| LessThan RelExpr ShiftExpr -- ^ <

| GreaterThan RelExpr ShiftExpr -- ^ \>

| LessEqual RelExpr ShiftExpr -- ^ <=

| GreaterEqual RelExpr ShiftExpr -- ^ \>=

| InstanceOf RelExpr ShiftExpr -- ^ instanceof

| InObject RelExpr ShiftExpr -- ^ in

deriving (Show, Eq)

-- | Shift expressions

data ShiftExpr

= AddExpr AddExpr

| ShiftLeft ShiftExpr AddExpr -- ^ <<

| ShiftRight ShiftExpr AddExpr -- ^ \>\>

| ShiftRight2 ShiftExpr AddExpr -- ^ \>\>\>

deriving (Show, Eq)

-- | Additive expressions

data AddExpr

= MultExpr MultExpr

| Plus AddExpr MultExpr -- ^ +

| Minus AddExpr MultExpr -- ^ \-

deriving (Show, Eq)

-- | Multiplicative expressions

data MultExpr

= UnaryExpr UnaryExpr

| Times MultExpr UnaryExpr -- ^ \\*

| Div MultExpr UnaryExpr -- ^ /

| Mod MultExpr UnaryExpr -- ^ %

deriving (Show, Eq)

-- | Unary expressions

data UnaryExpr

= PostFix PostFix

| Delete UnaryExpr -- ^ delete a

| Void UnaryExpr -- ^ void a

| TypeOf UnaryExpr -- ^ typeof a

| PlusPlus UnaryExpr -- ^ ++a

| MinusMinus UnaryExpr -- ^ \-\-a

| UnaryPlus UnaryExpr -- ^ +a

| UnaryMinus UnaryExpr -- ^ \-a

| Not UnaryExpr -- ^ !a

| BitNot UnaryExpr -- ^ ~a

deriving (Show, Eq)

-- | Post fix operators

data PostFix

= LeftExpr LeftExpr

| PostInc LeftExpr -- ^ a++

| PostDec LeftExpr -- ^ a\-\-

deriving (Show, Eq)

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- \* JSHOP \*

-- \* \*

-- \* Module: Parser \*

-- \* Purpose: JavaScript Parser \*

-- \* Authors: Nick Brunt, Henrik Nilsson \*

-- \* \*

-- \* Based (loosely) on the HMTC equivalent \*

-- \* Copyright (c) Henrik Nilsson, 2006 - 2011 \*

-- \* http://www.cs.nott.ac.uk/~nhn/ \*

-- \* \*

-- \* Revisions for JavaScript \*

-- \* Copyright (c) Nick Brunt, 2011 - 2012 \*

-- \* \*

-- \* Rules derived from ECMA-262 \*

-- \* \*

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

{

-- | JavaScript parser

**module** Parser **where**

-- Standard library imports

**import** Data.Char

**import** Control.Monad.State

**import** Control.Monad.Error

-- JSHOP module imports

**import** Token

**import** Lexer

**import** LexerMonad

**import** ParseTree

**import** ParseMonad

}

%name parse

%monad { P }

%lexer { monadicLexer } { EOF }

%tokentype { Token }

%error { parseError }

%token

LITINT { LitInt $$ }

LITFLOAT { LitFloat $$ }

LITSTR { LitStr $$ }

ID { Id $$ }

REGEX { Regex $$ }

BREAK { ResId "break" }

CASE { ResId "case" }

CATCH { ResId "catch" }

CONTINUE { ResId "continue" }

DEFAULT { ResId "default" }

DELETE { ResId "delete" }

DO { ResId "do" }

ELSE { ResId "else" }

FALSE { ResId "false" }

FINALLY { ResId "finally" }

FOR { ResId "for" }

FUNCTION { ResId "function" }

IF { ResId "if" }

IN { ResId "in" }

INSTANCEOF { ResId "instanceof" }

NEW { ResId "new" }

NULL { ResId "null" }

RETURN { ResId "return" }

SWITCH { ResId "switch" }

THIS { ResId "this" }

THROW { ResId "throw" }

TRUE { ResId "true" }

TRY { ResId "try" }

TYPEOF { ResId "typeof" }

VAR { ResId "var" }

VOID { ResId "void" }

WHILE { ResId "while" }

WITH { ResId "with" }

'.' { ResOp "." }

'[' { ResOp "[" }

']' { ResOp "]" }

'(' { ResOp "(" }

')' { ResOp ")" }

'++' { ResOp "++" }

'--' { ResOp "--" }

'-' { ResOp "-" }

'+' { ResOp "+" }

'~' { ResOp "~" }

'!' { ResOp "!" }

'\*' { ResOp "\*" }

'/' { ResOp "/" }

'%' { ResOp "%" }

'<<' { ResOp "<<" }

'>>' { ResOp ">>" }

'>>>' { ResOp ">>>" }

'<' { ResOp "<" }

'<=' { ResOp "<=" }

'>' { ResOp ">" }

'>=' { ResOp ">=" }

'==' { ResOp "==" }

'!=' { ResOp "!=" }

'===' { ResOp "===" }

'!==' { ResOp "!==" }

'&' { ResOp "&" }

'^' { ResOp "^" }

'|' { ResOp "|" }

'&&' { ResOp "&&" }

'||' { ResOp "||" }

'?' { ResOp "?" }

':' { ResOp ":" }

'=' { ResOp "=" }

'\*=' { ResOp "\*=" }

'+=' { ResOp "+=" }

'-=' { ResOp "-=" }

'/=' { ResOp "/=" }

'%=' { ResOp "%=" }

'<<=' { ResOp "<<=" }

'>>=' { ResOp ">>=" }

'>>>=' { ResOp ">>>=" }

'&=' { ResOp "&=" }

'^=' { ResOp "^=" }

'|=' { ResOp "|=" }

',' { ResOp "," }

';' { ResOp ";" }

'{' { ResOp "{" }

'}' { ResOp "}" }

OTHER { Other $$ }

EOF { EOF }

%%

-- Basic structure of a JS program

program :: { Tree }

: sources { Tree $1 }

sources :: { [Source] }

: {- epsilon -} { [] }

| source { [$1] }

| sources source { $1 ++ [$2] }

-- Possible source elements

source :: { Source }

: statement { Statement $1 }

| funcDecl { SFuncDecl $1 }

-- Function declaration

funcDecl :: { FuncDecl }

: FUNCTION funcDecl2

{ $2 }

| FUNCTION '(' formalParamList ')' '{' funcBody '}'

{ FuncDecl Nothing $3 $6 }

funcDecl2 :: { FuncDecl }

: ID '(' formalParamList ')' '{' funcBody '}'

{ FuncDecl (Just $1) $3 $6 }

formalParamList :: { [String] }

: {- epsilon -} { [] }

| ID { [ $1 ] }

| formalParamList ',' ID

{ $1 ++ [$3] }

funcBody :: { [Source] }

: sources { $1 }

-- Statements

statement :: { Statement }

: ';' { EmptyStmt }

| ifStmt { IfStmt $1}

| iterativeStmt { IterativeStmt $1 }

| block { Block $1 }

| exprStmt { ExprStmt $1 }

| varStmt { VarStmt $1 }

| tryStmt { TryStmt $1 }

| switchStmt { Switch $1 }

| CONTINUE ID ';'

{ ContinueStmt (Just $2) }

| CONTINUE ';'

{ ContinueStmt Nothing }

| BREAK ID ';'

{ BreakStmt (Just $2) }

| BREAK ';'

{ BreakStmt Nothing }

| RETURN exprSemi ';'

{ ReturnStmt (Just $2) }

| RETURN ';'

{ ReturnStmt Nothing }

| WITH '(' expression ')' statement

{ WithStmt $3 $5 }

| ID ':' statement

{ LabelledStmt $1 $3 }

| THROW exprSemi ';'

{ ThrowExpr $2 }

ifStmt :: { IfStmt }

: IF '(' expression ')' statement ELSE statement

{ IfElse $3 $5 $7 }

| IF '(' expression ')' statement

{ If $3 $5 }

iterativeStmt :: { IterativeStmt }

: DO statement WHILE '(' expression ')' ';'

{ DoWhile $2 $5 }

| WHILE '(' expression ')' statement

{ While $3 $5 }

| FOR '(' optExpression ';' optExpression ';' optExpression ')' statement

{ For $3 $5 $7 $9 }

| FOR '(' VAR varDeclList ';' optExpression ';' optExpression ')' statement

{ ForVar $4 $6 $8 $10 }

| FOR '(' VAR varDeclList IN expression ')' statement

{ ForVarIn $4 $6 $8 }

| FOR '(' leftExpr IN expression ')' statement

{ ForIn $3 $5 $7 }

exprStmt :: { Expression }

: exprSemi ';' { $1 }

exprSemi :: { Expression }

: expression {%% \t -> autoSemiInsert t $1 }

block :: { [Statement] }

-- Hack to stop empty blocks being parsed as empty object literals.

-- This is something to do with Haskell's pattern matching.

-- The empty list should really be picked up in stmtList, but it is

-- overridden by objectLit if not defined here.

: '{' {- epsilon -} '}' { [] }

| '{' stmtList '}' { $2 }

stmtList :: { [Statement] }

: {- epsilon -} { [] }

| statement { [$1] }

| stmtList statement { $1 ++ [$2] }

varStmt :: { [VarDecl] }

: VAR varDeclList ';' { $2 }

| VAR varDeclList { $2 }

tryStmt :: { TryStmt }

: TRY block catchList

{ TryBC $2 $3 }

| TRY block finally

{ TryBF $2 $3 }

| TRY block catchList finally

{ TryBCF $2 $3 $4 }

catchList :: { [Catch] }

: catch { [$1] }

| catchList catch { $1 ++ [$2 ] }

catch :: { Catch }

: CATCH '(' ID ')' block

{ Catch $3 $5 }

| CATCH '(' ID IF expression ')' block

{ CatchIf $3 $7 $5}

finally :: { [Statement] }

: FINALLY block { $2 }

switchStmt :: { Switch }

: SWITCH '(' expression ')' caseBlock

{ SSwitch $3 $5 }

caseBlock :: { CaseBlock }

: '{' caseClauses '}'

{ CaseBlock $2 [] [] }

| '{' caseClauses defaultClause caseClauses '}'

{ CaseBlock $2 [$3] $4 }

caseClauses :: { [CaseClause] }

: {- epsilon -} { [] }

| caseClause { [$1] }

| caseClauses caseClause { $1 ++ [$2] }

caseClause :: { CaseClause }

: CASE expression ':' stmtList

{ CaseClause $2 $4 }

defaultClause :: { DefaultClause }

: DEFAULT ':' stmtList

{ DefaultClause $3 }

expression :: { Expression }

: assignmentList { Assignment $1 }

assignmentList :: { [Assignment] }

: assignment { [$1] }

| assignmentList ',' assignment { $1 ++ [$3] }

optExpression :: { Maybe Expression }

: {- epsilon -} { Nothing }

| expression { Just $1 }

varDeclList :: { [VarDecl] }

: varDecl { [$1] }

| varDeclList ',' varDecl

{ $1 ++ [$3] }

varDecl :: { VarDecl }

: ID initialiser { VarDecl $1 $2 }

initialiser :: { Maybe Assignment }

: {- epsilon -} { Nothing }

| '=' assignment { Just $2 }

assignment :: { Assignment }

: leftExpr assignOp assignment

{ Assign $1 $2 $3 }

| condExpr { CondExpr $1 }

| funcDecl { AssignFuncDecl $1 }

leftExpr :: { LeftExpr }

: newExpr { NewExpr $1 }

| callExpr { CallExpr $1 }

assignOp :: { AssignOp }

: '\*=' { AssignOpMult }

| '/=' { AssignOpDiv }

| '%=' { AssignOpMod }

| '+=' { AssignOpPlus }

| '-=' { AssignOpMinus }

| '<<=' { AssignOpSLeft }

| '>>=' { AssignOpSRight }

| '>>>=' { AssignOpSRight2 }

| '&=' { AssignOpAnd }

| '^=' { AssignOpNot }

| '|=' { AssignOpOr }

| '=' { AssignNormal }

condExpr :: { CondExpr }

: logOr { LogOr $1 }

| logOr '?' assignment ':' assignment

{ CondIf $1 $3 $5 }

newExpr :: { NewExpr }

: memberExpr { MemberExpr $1 }

| NEW newExpr { NNewExpr $2 }

callExpr :: { CallExpr }

: memberExpr arguments

{ CallMember $1 $2 }

| callExpr arguments

{ CallCall $1 $2 }

| callExpr '[' expression ']'

{ CallSquare $1 $3 }

| callExpr '.' ID

{ CallDot $1 $3 }

memberExpr :: { MemberExpr }

: primaryExpr

{ MemExpression $1 }

| funcDecl

{ FuncExpr $1 }

| memberExpr '[' expression ']'

{ ArrayExpr $1 $3 }

| memberExpr '.' ID

{ MemberCall $1 $3 }

| NEW memberExpr arguments

{ MemberNew $2 $3 }

arguments :: { [Assignment] }

: '(' ')' { [] }

| '(' argumentList ')'

{ $2 }

argumentList :: { [Assignment] }

: assignment { [$1] }

| argumentList ',' assignment

{ $1 ++ [$3] }

-- Primary expression

primaryExpr :: { PrimaryExpr }

: literal { ExpLiteral $1 }

| ID { ExpId $1 }

| THIS { ExpThis }

| REGEX { ExpRegex $1 }

| arrayLit { ExpArray $1 }

| objectLit { ExpObject $1 }

| '(' expression ')'

{ ExpBrackExp $2 }

literal :: { Literal }

: NULL { LNull }

| TRUE { LBool True }

| FALSE { LBool False }

| LITINT { LInt $1 }

| LITFLOAT { LFloat $1 }

| LITSTR { LStr $1 }

arrayLit :: { ArrayLit }

: '[' elementList ']'

{ ArraySimp $2 }

elementList :: { [Assignment] }

: {- epsilon -} { [] }

| assignment { [$1] }

| elementList ',' assignment

{ $1 ++ [$3]}

objectLit :: { [(PropName, Assignment)] }

: '{' '}' { [] }

| '{' propertyList '}' { $2 }

propertyList :: { [(PropName, Assignment)] }

: property { [$1] }

| propertyList ',' property { $1 ++ [$3] }

property :: { (PropName, Assignment) }

: propertyName ':' assignment

{ ($1, $3) }

propertyName :: { PropName }

: ID { PropNameId $1 }

| LITSTR { PropNameStr $1 }

| LITINT { PropNameInt $1 }

logOr :: { LogOr }

: logAnd { LogAnd $1 }

| logOr '||' logAnd

{ LOLogOr $1 $3 }

logAnd :: { LogAnd }

: bitOR { BitOR $1 }

| logAnd '&&' bitOR

{ LALogAnd $1 $3 }

bitOR :: { BitOR }

: bitXOR { BitXOR $1 }

| bitOR '|' bitXOR

{ BOBitOR $1 $3 }

bitXOR :: { BitXOR }

: bitAnd { BitAnd $1 }

| bitXOR '^' bitAnd

{ BXBitXOR $1 $3 }

bitAnd :: { BitAnd }

: equalExpr { EqualExpr $1 }

| bitAnd '&' equalExpr

{ BABitAnd $1 $3 }

equalExpr :: { EqualExpr }

: relExpr { RelExpr $1 }

| equalExpr '==' relExpr

{ Equal $1 $3 }

| equalExpr '!=' relExpr

{ NotEqual $1 $3 }

| equalExpr '===' relExpr

{ EqualTo $1 $3 }

| equalExpr '!==' relExpr

{ NotEqualTo $1 $3 }

relExpr :: { RelExpr }

: shiftExpr { ShiftExpr $1 }

| relExpr '<' shiftExpr

{ LessThan $1 $3 }

| relExpr '>' shiftExpr

{ GreaterThan $1 $3 }

| relExpr '<=' shiftExpr

{ LessEqual $1 $3 }

| relExpr '>=' shiftExpr

{ GreaterEqual $1 $3 }

| relExpr INSTANCEOF shiftExpr

{ InstanceOf $1 $3 }

| relExpr IN shiftExpr

{ InObject $1 $3 }

shiftExpr :: { ShiftExpr }

: addExpr { AddExpr $1 }

| shiftExpr '<<' addExpr

{ ShiftLeft $1 $3 }

| shiftExpr '>>' addExpr

{ ShiftRight $1 $3 }

| shiftExpr '>>>' addExpr

{ ShiftRight2 $1 $3 }

addExpr :: { AddExpr }

: multExpr { MultExpr $1 }

| addExpr '+' multExpr

{ Plus $1 $3 }

| addExpr '-' multExpr

{ Minus $1 $3 }

multExpr :: { MultExpr }

: unaryExpr { UnaryExpr $1 }

| multExpr '\*' unaryExpr

{ Times $1 $3 }

| multExpr '/' unaryExpr

{Div $1 $3 }

| multExpr '%' unaryExpr

{ Mod $1 $3 }

unaryExpr :: { UnaryExpr }

: postFix { PostFix $1 }

| DELETE unaryExpr { Delete $2 }

| VOID unaryExpr { Void $2 }

| TYPEOF unaryExpr { TypeOf $2 }

| '++' unaryExpr { PlusPlus $2 }

| '--' unaryExpr { MinusMinus $2 }

| '+' unaryExpr { UnaryPlus $2 }

| '-' unaryExpr { UnaryMinus $2 }

| '!' unaryExpr { Not $2 }

| '~' unaryExpr { BitNot $2 }

postFix :: { PostFix }

: leftExpr { LeftExpr $1 }

| leftExpr '++' { PostInc $1 }

| leftExpr '--' { PostDec $1 }

{

-- | Handle errors thrown by the parser

parseError :: Token -> P a

parseError tok = **do**

s <- get

throwError ("Parse error:" ++

" lineno = " ++ show (lineno s) ++

", token = " ++ show tok ++

-- Only show the next 50 chars. Keeps error messages more tidy.

", rest = < " ++ take 50 (rest s) ++ "... >"

)

}

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: Diagnostics \*

\* Purpose: Diagnostic messages and computations (monad) \*

\* Authors: Nick Brunt, Henrik Nilsson \*

\* \*

\* Based on the HMTC equivalent \*

\* Copyright (c) Henrik Nilsson, 2006 - 2011 \*

\* <http://www.cs.nott.ac.uk/~nhn/> \*

\* \*

\* Revisions for JavaScript \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Diagnostic messages and computations (monad)

module Diagnostics where

-- | Diagnostic computation. A computation with diagnostic output can

-- succeed or fail, and additionally yields a list of diagnostic messages.

newtype D a = D ([String] -> (Maybe a, [String]))

unD :: D a -> ([String] -> (Maybe a, [String]))

unD (D x) = x

instance Monad D where

return a = D (\dms -> (Just a, dms))

d >>= f = D (\dms ->

case unD d dms of

(Nothing, dms') -> (Nothing, dms')

(Just a, dms') -> unD (f a) dms')

-- | Runs a diagnostic computation. Returns:

--

-- (1) Result of the computation, if any.

--

-- (2) Sorted list of diagnostic messages.

runD :: D a -> (Maybe a, [String])

runD d = (ma, dms)

where

(ma, dms) = unD d []

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: CodeCompMonad \*

\* Purpose: Code Compression Monad \*

\* Authors: Nick Brunt, Henrik Nilsson \*

\* \*

\* Based (loosely) on the HMTC equivalent \*

\* Copyright (c) Henrik Nilsson, 2006 - 2011 \*

\* <http://www.cs.nott.ac.uk/~nhn/> \*

\* \*

\* Revisions for JavaScript \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Code compression monad. This module provides an abstraction for

-- code compression computations with support for generation of distinct

-- names, e.g. for variables and functions.

module CodeCompMonad (

-- \* Code generation computation

CC, -- Abstract. Instances: Monad.

emit, -- i -> CC i x ()

runCC, -- CC i x a -> (a, [i], [x])

-- \*\* Compression functions

pop, -- CC String () String

showState, -- CC String () ()

regID, -- String -> CC String () ()

incScope, -- CC String () ()

decScope, -- Bool -> CC String () ()

resetScope, -- CC String () ()

emitID -- String -> CC String () ()

) where

-- Standard library imports

import Data.List

------------------------------------------------------------------------------

-- Code generator state

------------------------------------------------------------------------------

data CCState i x

= CCS {

identifiers :: [(Identifier, (Int,Int))],

-- ^ List of identifiers and their scope and level

currentScope :: Int, -- ^ Increment this every time we enter a new block

currentLevel :: Int, -- ^ Increment this every time we enter a new block

-- Decrement it every time we leave a block

divs :: [[i]], -- ^ Diversions

sect :: [i], -- ^ Current section

aux :: [x] -- ^ Auxiliary stream

}

deriving Show

data Identifier

= Variable {

origName :: String,

compName :: String

}

deriving Show

------------------------------------------------------------------------------

-- Code generator computation

------------------------------------------------------------------------------

-- | Code generation computation. Parameterised on the type of instructions

-- and additional auxiliary information. One use of the auxiliary information

-- is for additional separate output sections by instantiating with suitable

-- disjoint union type.

-- For example, Either can be used to implement prefix and suffix sections:

-- emitPfx i = emitAux (Left i), emitSfx = emitAux (Right i)

newtype CC i x a = CC (CCState i x -> (a, CCState i x))

unCC :: CC i x a -> (CCState i x -> (a, CCState i x))

unCC (CC f) = f

instance Monad (CC i x) where

return a = CC $ \ccs -> (a, ccs)

cc >>= f = CC $ \ccs ->

let (a, ccs') = unCC cc ccs

in unCC (f a) ccs'

-- | Return current state

get :: CC i x (CCState i x)

get = CC $ \ccs -> (ccs, ccs)

-- | Store given state

put :: (CCState i x) -> CC i x ()

put ccs = CC $ \ccs' -> ((), ccs)

-- | Pops the last item off the stack and returns it

pop :: CC String () String

pop = do

ccs@(CCS {sect=(s:ss)}) <- get

put $ ccs{sect=ss}

return s

-- | Emit instruction

emit :: i -> CC i x ()

emit i = CC $ \ccs -> ((), ccs {sect = i : sect ccs})

-- | Increment the current scope and level

incScope :: CC String () ()

incScope = do

ccs@(CCS {currentScope = cs, currentLevel = cl}) <- get

put $ ccs {currentScope = cs+1, currentLevel = cl+1}

-- | Decrement the current scope. Depending on the forget flag, forget all identifiers

-- from the previous scope.

decScope :: Bool -> CC String () ()

decScope forget = do

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

let ids' = if forget then forgetScopes ids cs cl

else ids

put $ ccs {identifiers = ids', currentScope = cs, currentLevel = cl-1}

where

forgetScopes :: [(Identifier, (Int,Int))] -> Int -> Int -> [(Identifier, (Int,Int))]

forgetScopes [] \_ \_ = []

forgetScopes (x:xs) s l = if fst (snd x) <= s && snd (snd x) == l then

forgetScopes xs s l

else x:(forgetScopes xs s l)

-- | Reset the current scope and level to 0

resetScope :: CC String () ()

resetScope = CC $ \ccs -> ((), ccs {currentScope = 0, currentLevel = 0})

-- | Show the current code compression state

showState :: CC String () ()

showState = do

emit "\n\n"

CC $ \ccs -> ((), ccs {sect = (ppCCS ccs) : sect ccs})

emit "\n\n"

-- | Register the given identifier as a known id

regID :: String -> CC String () ()

regID id = do

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

put $ ccs {

identifiers = ((Variable {

origName = id,

compName = ""

}),(cs,cl)):ids

}

-- | Given an id, scope and level, generate a compressed id

newID :: String -- ^ Original ID

-> Int -- ^ Scope of original ID

-> Int -- ^ Level of original ID

-> CC String () String -- ^ Return the compressed ID

newID origID s l = do

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

let usedIDs = [compName i | (i,(s',l')) <- ids, s' <= cs, l' <= cl] -- only from same scope or lower

let compID = if l == 0 then -- 0 = global

origID

else

genID usedIDs genIDList

let ids' = updateCompID origID s l compID ids

put $ ccs {identifiers = ids'}

return compID

where

updateCompID :: String -> Int -> Int -> String -> [(Identifier, (Int,Int))] -> [(Identifier, (Int,Int))]

updateCompID \_ \_ \_ \_ [] = []

updateCompID origID s l compID ((id,(s',l')):ids)

= if origName id == origID && s' == s && l' == l then

((Variable {origName = origID, compName = compID}), (s,l)):ids

else

(id,(s',l')):(updateCompID origID s l compID ids)

genID :: [String] -> [String] -> String

genID [] idList = head idList

genID usedIDs (id:ids) = if elem id usedIDs then genID usedIDs ids

else id

{- |

Given the original ID, looks up the compressed ID.

If there is no compressed ID, an attempt is made to create one.

If the original ID is a library function or a global, the original is emitted.

-}

emitID :: String -> CC String () String

emitID origID = do

ccs@(CCS {identifiers = ids, currentScope = cs, currentLevel = cl}) <- get

let mbCompID = findCompID origID ids cs cl

case mbCompID of

Just (compID, (s,l)) -> if compID /= "" then

return compID

else do

-- Generate new name

compID' <- newID origID s l

return compID'

Nothing -> do

-- Must be either a library function, or global var

put $ ccs {

identifiers = ((Variable {

origName = origID,

compName = origID

}),(0,0)):ids

}

return origID

where

findCompID :: String -> [(Identifier, (Int,Int))] -> Int -> Int -> Maybe (String, (Int,Int))

findCompID origID [] \_ \_ = Nothing

findCompID origID ((ids, (s,l)):xs) cs cl =

if s <= cs && l <= cl && origID == (origName ids) then

Just (compName ids, (s,l))

else

findCompID origID xs cs cl

-- | Run a code generation computation

runCC :: CC i x a -> (a, [i], [x])

runCC cc =

let

(a, ccs') = unCC cc ccs0

in

(a, joinSects (sect ccs' : divs ccs'), reverse (aux ccs'))

where

ccs0 = CCS {

identifiers = [],

currentScope = 0,

currentLevel = 0,

divs = [],

sect = [],

aux = []

}

joinSects :: [[i]] -> [i]

joinSects [] = []

joinSects (s:ss) = jsAux (joinSects ss) s

where

jsAux is [] = is

jsAux is (i:ris) = jsAux (i:is) ris

-- | Generates an infinite list of all possible identifier names

genIDList :: [String]

genIDList = [c:s | s <- "":allStrings, c <- firstChar] \\ reservedIDs

where

firstChar = ['a'..'z']++['A'..'Z']++['\_'] -- Technically the first char can be a '$' but this is

-- monopolised by jQuery so I'll leave it out for simplicity

-- Note: To make all var names unique, I could use a special character... Maybe 'λ'?

-- Unfortunately this is not valid ASCII so it makes reading and writing files very difficult!

alph = ['a'..'z']++['A'..'Z']++['0'..'9']++['$','\_']

allStrings = [c:s | s <- "":allStrings, c <- alph]

reservedIDs = ["if","in","do","int","for","new","try","var"]

-- There are obviously more, but none shorter than 4 letters. A longer list would

-- decrease performance unneccesarily. Variables longer than 3 letters should never

-- arise.

-- > length $ takeWhile (\xs -> length xs < 4) genIDList

-- > 220525

-- | Pretty print the Code Compression state

ppCCS :: (CCState String ()) -> String

ppCCS (CCS {identifiers = ids, currentScope = cs, currentLevel = cl, divs = ds, sect = ss, aux = as})

= "\n----------- CC State -----------\n\n"

++ "Current Scope: " ++ show cs ++ "\n\n"

++ "Current Level: " ++ show cl ++ "\n\n"

++ "Identifiers:\n" ++ ppIds ids ++ "\n\n"

++ "Divs: " ++ show ds ++ "\n\n"

++ "Sect: " ++ show ss ++ "\n\n"

++ "Aux: " ++ show as ++ "\n\n"

++ "---------------------------------\n"

-- | Pretty print the identifier list

ppIds :: [(Identifier, (Int,Int))] -> String

ppIds [] = ""

ppIds ((id,(s,l)):ids) = " S" ++ show s ++ " L" ++ show l ++ ": " ++ show id ++ "\n" ++ ppIds ids

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: Analyser \*

\* Purpose: Analyses the Parse Tree before code compression to detect \*

\* variable declarations. \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Analyses the Parse Tree before code compression to detect variable declarations.

module Analyser where

-- JSHOP module imports

import Diagnostics

import ParseTree

import CodeCompMonad

import CompUtils

anMaybe :: (a -> JSCC()) -> (Maybe a) -> JSCC()

anMaybe anFunc mbExpr

= case mbExpr of

Just expr -> anFunc expr

Nothing -> return ()

-- | Source

anSrc :: Source -> JSCC()

anSrc (Statement stmt) = anStmt stmt

anSrc (SFuncDecl funcDecl) = anFuncDecl funcDecl

anSrcSeq :: [Source] -> JSCC()

anSrcSeq x = mapM\_ anSrc x

-- | Function declaration

anFuncDecl :: FuncDecl -> JSCC()

anFuncDecl (FuncDecl (Just id) formalParamList sources) = do

regID id

incScope

anFormalParamList formalParamList

anSrcSeq sources

decScope False

anFuncDecl (FuncDecl Nothing formalParamList sources) = do

incScope

anFormalParamList formalParamList

anSrcSeq sources

decScope False

-- | Formal param list

anFormalParamList :: [String] -> JSCC()

anFormalParamList ids = mapM\_ regID ids

-- | Statement

anStmt :: Statement -> JSCC()

anStmt (IfStmt ifStmt) = anIfStmt ifStmt

anStmt (IterativeStmt itStmt) = anItStmt itStmt

anStmt (ExprStmt expr) = anExpr expr

anStmt (TryStmt tryStmt) = anTryStmt tryStmt

anStmt (Switch switch) = anSwitch switch

anStmt (VarStmt varDecls) = anVarDeclList varDecls

anStmt (Block stmts) = anStmtSeq stmts

anStmt (ReturnStmt (Just expr)) = anExpr expr

anStmt (WithStmt expr stmt) = anExpr expr >> anStmt stmt

anStmt (LabelledStmt id stmt) = regID id >> anStmt stmt

anStmt (ThrowExpr expr) = anExpr expr

anStmt \_ = return ()

anStmtSeq :: [Statement] -> JSCC()

anStmtSeq s = mapM\_ anStmt s

-- | If statement

anIfStmt :: IfStmt -> JSCC()

anIfStmt (IfElse expr stmtTrue stmtFalse) = do

anExpr expr

anStmt stmtTrue

anStmt stmtFalse

anIfStmt (If expr stmt) = anExpr expr >> anStmt stmt

-- | Iterative statement

anItStmt :: IterativeStmt -> JSCC()

anItStmt (DoWhile stmt expr) = anStmt stmt >> anExpr expr

anItStmt (While expr stmt) = anExpr expr >> anStmt stmt

anItStmt (For mbExpr mbExpr2 mbExpr3 stmt) = do

anMaybe anExpr mbExpr

anMaybe anExpr mbExpr2

anMaybe anExpr mbExpr3

anStmt stmt

anItStmt (ForVar varDecls mbExpr2 mbExpr3 stmt) = do

anVarDeclList varDecls

anMaybe anExpr mbExpr2

anMaybe anExpr mbExpr3

anStmt stmt

anItStmt (ForIn leftExpr expr stmt) = do

anLeftExpr leftExpr

anExpr expr

anStmt stmt

anItStmt (ForVarIn varDecls expr stmt) = do

anVarDeclList varDecls

anExpr expr

anStmt stmt

-- | Try statement

anTryStmt :: TryStmt -> JSCC()

anTryStmt (TryBC stmts catchs) = anStmtSeq stmts >> anCatchSeq catchs

anTryStmt (TryBF stmts stmts2) = anStmtSeq stmts >> anStmtSeq stmts2

anTryStmt (TryBCF stmts catchs stmts2) = do

anStmtSeq stmts

anCatchSeq catchs

anStmtSeq stmts2

-- | Catch

anCatch :: Catch -> JSCC()

anCatch (Catch id stmts) = regID id >> anStmtSeq stmts

anCatch (CatchIf id stmts expr) = do

regID id

anExpr expr

anStmtSeq stmts

anCatchSeq :: [Catch] -> JSCC()

anCatchSeq c = mapM\_ anCatch c

-- | Switch

anSwitch :: Switch -> JSCC()

anSwitch (SSwitch expr caseBlock) = anExpr expr >> anCaseBlock caseBlock

-- | Case block

anCaseBlock :: CaseBlock -> JSCC()

anCaseBlock (CaseBlock caseClauses defaultClauses caseClauses2) = do

anCaseClauseSeq caseClauses

anDefaultClauseSeq defaultClauses

anCaseClauseSeq caseClauses2

-- | Case clause

anCaseClause :: CaseClause -> JSCC()

anCaseClause (CaseClause expr stmts) = anExpr expr >> anStmtSeq stmts

anCaseClauseSeq :: [CaseClause] -> JSCC()

anCaseClauseSeq cc = mapM\_ anCaseClause cc

-- | Default clause

anDefaultClause :: DefaultClause -> JSCC()

anDefaultClause (DefaultClause stmts) = anStmtSeq stmts

anDefaultClauseSeq :: [DefaultClause] -> JSCC()

anDefaultClauseSeq dc = mapM\_ anDefaultClause dc

-- | Expression

anExpr :: Expression -> JSCC()

anExpr (Assignment assigns) = anAssignList assigns

-- | Var declaration

anVarDecl :: VarDecl -> JSCC()

anVarDecl (VarDecl id (Just assign)) = regID id >> anAssign assign

anVarDecl (VarDecl id Nothing) = regID id

anVarDeclList :: [VarDecl] -> JSCC()

anVarDeclList vd = mapM\_ anVarDecl vd

-- | Assignment

anAssign :: Assignment -> JSCC()

anAssign (CondExpr condExpr) = anCondExpr condExpr

anAssign (Assign leftExpr assignOp assign) = do

anLeftExpr leftExpr

anAssignOp assignOp

anAssign assign

anAssign (AssignFuncDecl funcDecl) = anFuncDecl funcDecl

anAssignList :: [Assignment] -> JSCC()

anAssignList a = mapM\_ anAssign a

-- | Left expression

anLeftExpr :: LeftExpr -> JSCC()

anLeftExpr (NewExpr newExpr) = anNewExpr newExpr

anLeftExpr (CallExpr callExpr) = anCallExpr callExpr

-- | Assignment operator

anAssignOp :: AssignOp -> JSCC()

anAssignOp \_ = return ()

-- | Conditional expression

anCondExpr :: CondExpr -> JSCC()

anCondExpr (LogOr logOr) = anLogOr logOr

anCondExpr (CondIf logOr assignTrue assignFalse) = do

anLogOr logOr

anAssign assignTrue

anAssign assignFalse

-- | New expression

anNewExpr :: NewExpr -> JSCC()

anNewExpr (MemberExpr memberExpr) = anMemberExpr memberExpr

anNewExpr (NNewExpr newExpr) = anNewExpr newExpr

-- | Call expression

anCallExpr :: CallExpr -> JSCC()

anCallExpr (CallMember memberExpr assigns) = anMemberExpr memberExpr >> anAssignList assigns

anCallExpr (CallCall callExpr assigns) = anCallExpr callExpr >> anAssignList assigns

anCallExpr (CallSquare callExpr expr) = anCallExpr callExpr >> anExpr expr

anCallExpr (CallDot callExpr id) = anCallExpr callExpr

-- | Member expression

anMemberExpr :: MemberExpr -> JSCC()

anMemberExpr (MemExpression primExpr) = anPrimExpr primExpr

anMemberExpr (FuncExpr funcDecl) = anFuncDecl funcDecl

anMemberExpr (ArrayExpr memberExpr expr) = anMemberExpr memberExpr >> anExpr expr

anMemberExpr (MemberNew memberExpr assigns) = anMemberExpr memberExpr >> anAssignList assigns

anMemberExpr (MemberCall memberExpr id) = anMemberExpr memberExpr

-- | Primary expressions

anPrimExpr :: PrimaryExpr -> JSCC()

anPrimExpr (ExpArray arrayLit) = anArrayLit arrayLit

anPrimExpr (ExpObject objLit) = anObjLit objLit

anPrimExpr (ExpBrackExp expr) = anExpr expr

anPrimExpr \_ = return ()

-- | Array literal

anArrayLit :: ArrayLit -> JSCC()

anArrayLit (ArraySimp elementList) = anElementList elementList

anElementList :: [Assignment] -> JSCC()

anElementList a = mapM\_ anAssign a

-- | Object literal

anObjLit :: [(PropName, Assignment)] -> JSCC()

anObjLit [] = return ()

anObjLit propList = anPropList propList

anPropList :: [(PropName, Assignment)] -> JSCC()

anPropList p = mapM\_ anProp p

anProp :: (PropName, Assignment) -> JSCC()

anProp (propName, assign) = anPropName propName >> anAssign assign

-- | Property name

anPropName :: PropName -> JSCC()

anPropName \_ = return ()

-- | Logical or

anLogOr :: LogOr -> JSCC()

anLogOr (LogAnd logAnd) = anLogAnd logAnd

anLogOr (LOLogOr logOr logAnd) = anLogOr logOr >> anLogAnd logAnd

-- | Logical and

anLogAnd :: LogAnd -> JSCC()

anLogAnd (BitOR bitOr) = anBitOr bitOr

anLogAnd (LALogAnd logAnd bitOr) = anLogAnd logAnd >> anBitOr bitOr

-- | Bitwise or

anBitOr :: BitOR -> JSCC()

anBitOr (BitXOR bitXor) = anBitXor bitXor

anBitOr (BOBitOR bitOr bitXor) = anBitOr bitOr >> anBitXor bitXor

-- | Bitwise xor

anBitXor :: BitXOR -> JSCC()

anBitXor (BitAnd bitAnd) = anBitAnd bitAnd

anBitXor (BXBitXOR bitXor bitAnd) = anBitXor bitXor >> anBitAnd bitAnd

-- | Bitwise and

anBitAnd :: BitAnd -> JSCC()

anBitAnd (EqualExpr equalExpr) = anEqualExpr equalExpr

anBitAnd (BABitAnd bitAnd equalExpr) = anBitAnd bitAnd >> anEqualExpr equalExpr

-- | Equality operators

anEqualExpr :: EqualExpr -> JSCC()

anEqualExpr (RelExpr relExpr) = anRelExpr relExpr

anEqualExpr (Equal equalExpr relExpr) = anEqualExpr equalExpr >> anRelExpr relExpr

anEqualExpr (NotEqual equalExpr relExpr) = anEqualExpr equalExpr >> anRelExpr relExpr

anEqualExpr (EqualTo equalExpr relExpr) = anEqualExpr equalExpr >> anRelExpr relExpr

anEqualExpr (NotEqualTo equalExpr relExpr) = anEqualExpr equalExpr >> anRelExpr relExpr

-- | Relational operators

anRelExpr :: RelExpr -> JSCC()

anRelExpr (ShiftExpr shiftExpr) = anShiftExpr shiftExpr

anRelExpr (LessThan relExpr shiftExpr) = anRelExpr relExpr >> anShiftExpr shiftExpr

anRelExpr (GreaterThan relExpr shiftExpr) = anRelExpr relExpr >> anShiftExpr shiftExpr

anRelExpr (LessEqual relExpr shiftExpr) = anRelExpr relExpr >> anShiftExpr shiftExpr

anRelExpr (GreaterEqual relExpr shiftExpr) = anRelExpr relExpr >> anShiftExpr shiftExpr

anRelExpr (InstanceOf relExpr shiftExpr) = anRelExpr relExpr >> anShiftExpr shiftExpr

anRelExpr (InObject relExpr shiftExpr) = anRelExpr relExpr >> anShiftExpr shiftExpr

-- | Shift operators

anShiftExpr :: ShiftExpr -> JSCC()

anShiftExpr (AddExpr addExpr) = anAddExpr addExpr

anShiftExpr (ShiftLeft shiftExpr addExpr) = anShiftExpr shiftExpr >> anAddExpr addExpr

anShiftExpr (ShiftRight shiftExpr addExpr) = anShiftExpr shiftExpr >> anAddExpr addExpr

anShiftExpr (ShiftRight2 shiftExpr addExpr) = anShiftExpr shiftExpr >> anAddExpr addExpr

-- | Additive operators

anAddExpr :: AddExpr -> JSCC()

anAddExpr (MultExpr multExpr) = anMultExpr multExpr

anAddExpr (Plus addExpr multExpr) = anAddExpr addExpr >> anMultExpr multExpr

anAddExpr (Minus addExpr multExpr) = anAddExpr addExpr >> anMultExpr multExpr

-- | Multiplicative operators

anMultExpr :: MultExpr -> JSCC()

anMultExpr (UnaryExpr unaryExpr) = anUnaryExpr unaryExpr

anMultExpr (Times multExpr unaryExpr) = anMultExpr multExpr >> anUnaryExpr unaryExpr

anMultExpr (Div multExpr unaryExpr) = anMultExpr multExpr >> anUnaryExpr unaryExpr

anMultExpr (Mod multExpr unaryExpr) = anMultExpr multExpr >> anUnaryExpr unaryExpr

-- | Unary operators

anUnaryExpr :: UnaryExpr -> JSCC()

anUnaryExpr (PostFix postFix) = anPostFix postFix

anUnaryExpr (Delete unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (Void unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (TypeOf unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (PlusPlus unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (MinusMinus unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (UnaryPlus unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (UnaryMinus unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (Not unaryExpr) = anUnaryExpr unaryExpr

anUnaryExpr (BitNot unaryExpr) = anUnaryExpr unaryExpr

-- | Post fix

anPostFix :: PostFix -> JSCC()

anPostFix (LeftExpr leftExpr) = anLeftExpr leftExpr

anPostFix (PostInc leftExpr) = anLeftExpr leftExpr

anPostFix (PostDec leftExpr) = anLeftExpr leftExpr

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: CodeCompressor \*

\* Purpose: Generate and compress JavaScript from the Parse Tree \*

\* Authors: Nick Brunt, Henrik Nilsson \*

\* \*

\* Based on the HMTC equivalent \*

\* Copyright (c) Henrik Nilsson, 2006 - 2011 \*

\* <http://www.cs.nott.ac.uk/~nhn/> \*

\* \*

\* Revisions for JavaScript \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\* As defined in ECMA-262 \*

\* \*

\* Expressions - Page 40 \*

\* Statements - Page 61 \*

\* Function Definition - Page 71 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Generate and compress JavaScript from the Parse Tree

module CodeCompressor where

-- Standard library imports

import Monad (when)

import Char (isDigit)

import Array

import Maybe

-- JSHOP module imports

import Diagnostics

import ParseTree

import CodeCompMonad

import Analyser

import CompUtils

------------------------------------------------------------------------------

-- Code generation functions

------------------------------------------------------------------------------

-- | Generates a JavaScript program from the Parse Tree

genCode :: Tree -> D [String]

genCode tree = do

let (\_, code, \_) = runCC (run tree)

return code

-- | Generate code to run a complete program

run :: Tree -> JSCC ()

run (Tree sources) = do

anSrcSeq sources

resetScope

genSrcSeq sources

-- | Source

genSrc :: Source -> JSCC()

genSrc (Statement stmt) = genStmt stmt

genSrc (SFuncDecl funcDecl) = genFuncDecl funcDecl

genSrcSeq :: [Source] -> JSCC()

genSrcSeq s = mapM\_ genSrc s

-- | Function declaration

genFuncDecl :: FuncDecl -> JSCC()

genFuncDecl (FuncDecl (Just id) formalParamList sources) = do

emit "function "

id' <- emitID id

emit id'

incScope

emit "("

genFormalParamList formalParamList

emit "){"

genSrcSeq sources

emit "}"

-- showState

decScope True

genFuncDecl (FuncDecl Nothing formalParamList sources) = do

emit $ "function("

incScope

genFormalParamList formalParamList

emit "){"

genSrcSeq sources

emit "}"

-- showState

decScope True

-- | Formal param list

genFormalParamList :: [String] -> JSCC()

genFormalParamList [] = emit ""

genFormalParamList [id] = do

id' <- emitID id

emit id'

genFormalParamList ids = do

genFormalParamList $ init ids

emit ","

ids' <- emitID $ last ids

emit ids'

-- | Statement

genStmt :: Statement -> JSCC()

genStmt EmptyStmt = emit ";"

genStmt (IfStmt ifStmt) = genIfStmt ifStmt

genStmt (IterativeStmt itStmt) = genItStmt itStmt

genStmt (ExprStmt expr) = do

genExpr expr

emit ";"

genStmt (TryStmt tryStmt) = genTryStmt tryStmt

genStmt (Switch switch) = genSwitch switch

genStmt (VarStmt varDecls) = do

emit "var "

genVarDeclList varDecls

emit ";"

genStmt (Block stmts) = do

emit "{"

genStmtSeq stmts

emit "}"

genStmt (ContinueStmt (Just id)) = do

emit $ "continue "

id' <- emitID id

emit id'

emit ";"

genStmt (ContinueStmt Nothing)

= emit "continue;"

genStmt (BreakStmt (Just id)) = do

emit "break "

id' <- emitID id

emit id'

emit ";"

genStmt (BreakStmt Nothing)

= emit "break;"

genStmt (ReturnStmt (Just expr)) = do

emit "return "

genExpr expr

emit ";"

genStmt (ReturnStmt Nothing) = emit "return;"

genStmt (WithStmt expr stmt) = do

emit "with("

genExpr expr

emit ")"

genStmt stmt

genStmt (LabelledStmt id stmt) = do

id' <- emitID id

emit id'

emit ":"

genStmt stmt

genStmt (ThrowExpr expr) = do

emit "throw "

genExpr expr

emit ";"

genStmtSeq :: [Statement] -> JSCC()

genStmtSeq [] = return ()

genStmtSeq (x:xs) = do

genStmt x

genStmtSeq xs

-- | If statement

genIfStmt :: IfStmt -> JSCC()

genIfStmt ifStmt@(IfElse expr

(ExprStmt (Assignment [Assign leftExprTrue assignOpTrue assignTrue]))

(ExprStmt (Assignment [Assign leftExprFalse assignOpFalse assignFalse])))

= if leftExprTrue == leftExprFalse then

genTernaryCond expr leftExprTrue assignOpTrue assignTrue assignFalse

else

genIfElse ifStmt

genIfStmt ifStmt@(IfElse expr

(Block [ExprStmt (Assignment [Assign leftExprTrue assignOpTrue assignTrue])])

(Block [ExprStmt (Assignment [Assign leftExprFalse assignOpFalse assignFalse])]))

= if leftExprTrue == leftExprFalse then

genTernaryCond expr leftExprTrue assignOpTrue assignTrue assignFalse

else

genIfElse ifStmt

genIfStmt ifStmt@(IfElse \_ \_ \_) = genIfElse ifStmt

genIfStmt (If expr stmt) = do

emit "if("

genExpr expr

emit ")"

genStmt stmt

-- | Optimisation note:

-- If the true or false statements are blocks (surrounded by braces), no

-- spaces are needed around the else keyword.

genIfElse :: IfStmt -> JSCC()

genIfElse (IfElse expr stmtTrue stmtFalse) = do

emit "if("

genExpr expr

emit ")"

genStmt stmtTrue

blockSpace stmtTrue

emit "else"

blockSpace stmtFalse

genStmt stmtFalse

where

blockSpace :: Statement -> JSCC()

blockSpace (Block \_) = return ()

blockSpace \_ = emit " "

-- | Ternary conditional

genTernaryCond :: Expression -- ^ Condition

-> LeftExpr -- ^ Left side of assignment

-> AssignOp -- ^ Assignment operator

-> Assignment -- ^ Assignment if true

-> Assignment -- ^ Assignment if false

-> JSCC()

genTernaryCond expr leftExpr assignOp assignTrue assignFalse = do

leftExpr' <- genLeftExpr leftExpr

genMaybe genPrimExpr leftExpr'

genAssignOp assignOp

genExpr expr

emit "?"

genAssign assignTrue

emit ":"

genAssign assignFalse

emit ";"

-- | Iterative statement

genItStmt :: IterativeStmt -> JSCC()

genItStmt (DoWhile stmt expr) = do

emit "do "

genStmt stmt

emit " while("

genExpr expr

emit ");"

genItStmt (While expr stmt) = do

emit "while("

genExpr expr

emit ")"

genStmt stmt

genItStmt (For mbExpr mbExpr2 mbExpr3 stmt) = do

emit "for("

genMaybe genExpr mbExpr

emit ";"

genMaybe genExpr mbExpr2

emit ";"

genMaybe genExpr mbExpr3

emit ")"

genStmt stmt

genItStmt (ForVar varDecls mbExpr2 mbExpr3 stmt) = do

emit "for(var "

genVarDeclList varDecls

emit ";"

genMaybe genExpr mbExpr2

emit ";"

genMaybe genExpr mbExpr3

emit ")"

genStmt stmt

genItStmt (ForIn leftExpr expr stmt) = do

emit "for("

leftExpr' <- genLeftExpr leftExpr

genMaybe genPrimExpr leftExpr'

emit " in "

genExpr expr

emit ")"

genStmt stmt

genItStmt (ForVarIn varDecls expr stmt) = do

emit "for(var "

genVarDeclList varDecls

emit " in "

genExpr expr

emit ")"

genStmt stmt

-- | Try statement

genTryStmt :: TryStmt -> JSCC()

genTryStmt (TryBC stmts catchs) = do

emit "try{"

genStmtSeq stmts

emit "}"

genCatchSeq catchs

genTryStmt (TryBF stmts stmts2) = do

emit "try{"

genStmtSeq stmts

emit "}finally{"

genStmtSeq stmts2

emit "}"

genTryStmt (TryBCF stmts catchs stmts2) = do

emit "try{"

genStmtSeq stmts

emit "}"

genCatchSeq catchs

emit "finally{"

genStmtSeq stmts2

emit "}"

-- | Catch

genCatch :: Catch -> JSCC()

genCatch (Catch id stmts) = do

emit "catch("

id' <- emitID id

emit id'

emit "){"

genStmtSeq stmts

emit "}"

-- Not in ECMA-262

-- <http://code.google.com/p/jslibs/wiki/JavascriptTips#Exceptions_Handling_/_conditional_catch_(try_catch_if)>

genCatch (CatchIf id stmts expr) = do

emit "catch("

id' <- emitID id

emit id'

emit " if "

genExpr expr

emit "){"

genStmtSeq stmts

emit "}"

genCatchSeq :: [Catch] -> JSCC()

genCatchSeq c = mapM\_ genCatch c

-- | Switch

genSwitch :: Switch -> JSCC()

genSwitch (SSwitch expr caseBlock) = do

emit "switch("

genExpr expr

emit ")"

genCaseBlock caseBlock

-- | Case block

genCaseBlock :: CaseBlock -> JSCC()

genCaseBlock (CaseBlock caseClauses defaultClauses caseClauses2) = do

emit "{"

genCaseClauseSeq caseClauses

genDefaultClauseSeq defaultClauses

genCaseClauseSeq caseClauses2

emit "}"

-- | Case clause

genCaseClause :: CaseClause -> JSCC()

genCaseClause (CaseClause expr stmts) = do

emit "case "

genExpr expr

emit ":"

genStmtSeq stmts

genCaseClauseSeq :: [CaseClause] -> JSCC()

genCaseClauseSeq cc = mapM\_ genCaseClause cc

-- | Default clause

genDefaultClause :: DefaultClause -> JSCC()

genDefaultClause (DefaultClause stmts) = do

emit "default:"

genStmtSeq stmts

genDefaultClauseSeq :: [DefaultClause] -> JSCC()

genDefaultClauseSeq dc = mapM\_ genDefaultClause dc

-- | Expression

genExpr :: Expression -> JSCC()

genExpr (Assignment assigns) = genAssignList assigns

-- | Var declaration

genVarDecl :: VarDecl -> JSCC()

genVarDecl (VarDecl id (Just assign)) = do

id' <- emitID id

emit id'

emit "="

genAssign assign

genVarDecl (VarDecl id Nothing) = do

id' <- emitID id

emit id'

genVarDeclList :: [VarDecl] -> JSCC()

genVarDeclList [] = emit ""

genVarDeclList [varDecl] = genVarDecl varDecl

genVarDeclList varDecls = do

genVarDeclList $ init varDecls

emit ","

genVarDecl $ last varDecls

-- | Assignment

genAssign :: Assignment -> JSCC()

genAssign (CondExpr condExpr) = do

condExpr' <- genCondExpr condExpr

genMaybe genPrimExpr condExpr'

genAssign (Assign leftExpr assignOp assign) = do

leftExpr' <- genLeftExpr leftExpr

genMaybe genPrimExpr leftExpr'

genAssignOp assignOp

genAssign assign

genAssign (AssignFuncDecl funcDecl) = do

genFuncDecl funcDecl

genAssignList :: [Assignment] -> JSCC()

genAssignList [] = return ()

genAssignList [x] = genAssign x

genAssignList (x:xs) = do

genAssign x

emit ","

genAssignList xs

-- | Left expression

genLeftExpr :: LeftExpr -> JSCC (Maybe PrimaryExpr)

genLeftExpr (NewExpr newExpr) = do

newExpr' <- genNewExpr newExpr

return newExpr'

genLeftExpr (CallExpr callExpr) = do

genCallExpr callExpr

return Nothing

-- | Assignment operator

genAssignOp :: AssignOp -> JSCC()

genAssignOp AssignNormal = emit "="

genAssignOp AssignOpMult = emit "\*="

genAssignOp AssignOpDiv = emit "/="

genAssignOp AssignOpMod = emit "%="

genAssignOp AssignOpPlus = emit "+="

genAssignOp AssignOpMinus = emit "-="

genAssignOp AssignOpSLeft = emit "<<="

genAssignOp AssignOpSRight = emit ">>="

genAssignOp AssignOpSRight2 = emit ">>>="

genAssignOp AssignOpAnd = emit "&="

genAssignOp AssignOpNot = emit "^="

genAssignOp AssignOpOr = emit "|="

-- | Conditional expression

genCondExpr :: CondExpr -> JSCC (Maybe PrimaryExpr)

genCondExpr (LogOr logOr) = do

logOr' <- genLogOr logOr

return logOr'

genCondExpr (CondIf logOr assignTrue assignFalse) = do

logOr' <- genLogOr logOr

genMaybe genPrimExpr logOr'

emit "?"

genAssign assignTrue

emit ":"

genAssign assignFalse

return Nothing

-- | New expression

genNewExpr :: NewExpr -> JSCC (Maybe PrimaryExpr)

genNewExpr (MemberExpr memberExpr) = do

memberExpr' <- genMemberExpr memberExpr

return memberExpr'

genNewExpr (NNewExpr newExpr) = do

emit "new "

newExpr' <- genNewExpr newExpr

genMaybe genPrimExpr newExpr'

return Nothing

-- | Call expression

genCallExpr :: CallExpr -> JSCC()

genCallExpr (CallMember memberExpr assigns) = do

memberExpr' <- genMemberExpr memberExpr

genMaybe genPrimExpr memberExpr'

emit "("

genAssignList assigns

emit ")"

genCallExpr (CallCall callExpr assigns) = do

genCallExpr callExpr

emit "("

genAssignList assigns

emit ")"

genCallExpr (CallSquare callExpr expr) = do

genCallExpr callExpr

emit "["

genExpr expr

emit "]"

genCallExpr (CallDot callExpr id) = do

genCallExpr callExpr

emit "."

id' <- emitID id

emit id'

-- | Member expression

genMemberExpr :: MemberExpr -> JSCC (Maybe PrimaryExpr)

genMemberExpr (MemExpression primExpr) = do

primExpr' <- retPrimExpr primExpr

return primExpr'

genMemberExpr (FuncExpr funcDecl) = do

genFuncDecl funcDecl

return Nothing

genMemberExpr (ArrayExpr memberExpr expr) = do

memberExpr' <- genMemberExpr memberExpr

genMaybe genPrimExpr memberExpr'

emit "["

genExpr expr

emit "]"

return Nothing

genMemberExpr (MemberNew memberExpr assigns) = do

emit "new "

memberExpr' <- genMemberExpr memberExpr

genMaybe genPrimExpr memberExpr'

emit "("

genAssignList assigns

emit ")"

return Nothing

genMemberExpr (MemberCall memberExpr id) = do

memberExpr' <- genMemberExpr memberExpr

genMaybe genPrimExpr memberExpr'

emit "."

-- Do not shorten this. Could be referring to an external object.

emit id

return Nothing

-- | Primary expressions

genPrimExpr :: PrimaryExpr -> JSCC()

genPrimExpr (ExpLiteral lit) = genLiteral lit

genPrimExpr (ExpId id) = do

id' <- emitID id

emit id'

genPrimExpr ExpThis = emit "this"

genPrimExpr (ExpRegex regex) = emit regex

genPrimExpr (ExpArray arrayLit) = genArrayLit arrayLit

genPrimExpr (ExpObject objLit) = genObjLit objLit

genPrimExpr (ExpBrackExp (Assignment [CondExpr (LogOr (LogAnd (BitOR (BitXOR (BitAnd (EqualExpr (RelExpr (ShiftExpr (AddExpr (MultExpr (UnaryExpr (PostFix (LeftExpr (NewExpr (MemberExpr (MemExpression primExpr))))))))))))))))]))

= genPrimExpr primExpr -- Only one primary expression, don't emit brackets.

-- Yes, it's convoluted, but it's the simplest way.

genPrimExpr (ExpBrackExp expr) = do

emit "("

genExpr expr

emit ")"

retPrimExpr :: PrimaryExpr -> JSCC (Maybe PrimaryExpr)

retPrimExpr (ExpLiteral lit) = do

lit' <- retLiteral lit

return $ Just $ ExpLiteral lit'

retPrimExpr primExpr = return $ Just primExpr

retLiteral :: Literal -> JSCC Literal

retLiteral lit = return lit

-- | Array literal

genArrayLit :: ArrayLit -> JSCC()

genArrayLit (ArraySimp elementList) = do

emit "["

genElementList elementList

emit "]"

genElementList :: [Assignment] -> JSCC()

genElementList [] = emit ""

genElementList [assign] = genAssign assign

genElementList assigns = do

genElementList $ init assigns

emit ","

genAssign $ last assigns

-- | Object literal

genObjLit :: [(PropName, Assignment)] -> JSCC()

genObjLit [] = emit "{}"

genObjLit propList = do

emit "{"

genPropList propList

emit "}"

genPropList :: [(PropName, Assignment)] -> JSCC()

genPropList [prop] = genProp prop

genPropList props = do

genPropList $ init props

emit ","

genProp $ last props

genProp :: (PropName, Assignment) -> JSCC()

genProp (propName, assign) = do

genPropName propName

emit ":"

genAssign assign

-- | Property name

genPropName :: PropName -> JSCC()

genPropName (PropNameId id) = emit id

genPropName (PropNameStr str) = emit str

genPropName (PropNameInt int) = emit $ show int

-- | Logical or

genLogOr :: LogOr -> JSCC (Maybe PrimaryExpr)

genLogOr (LogAnd logAnd) = do

logAnd' <- genLogAnd logAnd

return logAnd'

genLogOr (LOLogOr logOr logAnd) = do

logOr' <- genLogOr logOr

genMaybe genPrimExpr logOr'

emit "||"

logAnd' <- genLogAnd logAnd

genMaybe genPrimExpr logAnd'

return Nothing

-- | Logical and

genLogAnd :: LogAnd -> JSCC (Maybe PrimaryExpr)

genLogAnd (BitOR bitOr) = do

bitOr' <- genBitOr bitOr

return bitOr'

genLogAnd (LALogAnd logAnd bitOr) = do

logAnd' <- genLogAnd logAnd

genMaybe genPrimExpr logAnd'

emit "&&"

bitOr' <- genBitOr bitOr

genMaybe genPrimExpr bitOr'

return Nothing

-- | Bitwise or

genBitOr :: BitOR -> JSCC (Maybe PrimaryExpr)

genBitOr (BitXOR bitXor) = do

bitXor' <- genBitXor bitXor

return bitXor'

genBitOr (BOBitOR bitOr bitXor) = do

bitOr' <- genBitOr bitOr

genMaybe genPrimExpr bitOr'

emit "|"

bitXor' <- genBitXor bitXor

genMaybe genPrimExpr bitXor'

return Nothing

-- | Bitwise xor

genBitXor :: BitXOR -> JSCC (Maybe PrimaryExpr)

genBitXor (BitAnd bitAnd) = do

bitAnd' <- genBitAnd bitAnd

return bitAnd'

genBitXor (BXBitXOR bitXor bitAnd) = do

bitXor' <- genBitXor bitXor

genMaybe genPrimExpr bitXor'

emit "^"

bitAnd' <- genBitAnd bitAnd

genMaybe genPrimExpr bitAnd'

return Nothing

-- | Bitwise and

genBitAnd :: BitAnd -> JSCC (Maybe PrimaryExpr)

genBitAnd (EqualExpr equalExpr) = do

equalExpr' <- genEqualExpr equalExpr

return equalExpr'

genBitAnd (BABitAnd bitAnd equalExpr) = do

bitAnd' <- genBitAnd bitAnd

genMaybe genPrimExpr bitAnd'

emit "&"

equalExpr' <- genEqualExpr equalExpr

genMaybe genPrimExpr equalExpr'

return Nothing

-- | Equality operators

genEqualExpr :: EqualExpr -> JSCC (Maybe PrimaryExpr)

genEqualExpr (RelExpr relExpr) = do

relExpr' <- genRelExpr relExpr

return relExpr'

genEqualExpr (Equal equalExpr relExpr) = do

equalExpr' <- genEqualExpr equalExpr

genMaybe genPrimExpr equalExpr'

emit "=="

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

return Nothing

genEqualExpr (NotEqual equalExpr relExpr) = do

equalExpr' <- genEqualExpr equalExpr

genMaybe genPrimExpr equalExpr'

emit "!="

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

return Nothing

genEqualExpr (EqualTo equalExpr relExpr) = do

equalExpr' <- genEqualExpr equalExpr

genMaybe genPrimExpr equalExpr'

emit "==="

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

return Nothing

genEqualExpr (NotEqualTo equalExpr relExpr) = do

equalExpr' <- genEqualExpr equalExpr

genMaybe genPrimExpr equalExpr'

emit "!=="

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

return Nothing

-- | Relational operators

genRelExpr :: RelExpr -> JSCC (Maybe PrimaryExpr)

genRelExpr (ShiftExpr shiftExpr) = do

shiftExpr' <- genShiftExpr shiftExpr

return shiftExpr'

genRelExpr (LessThan relExpr shiftExpr) = do

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

emit "<"

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

return Nothing

genRelExpr (GreaterThan relExpr shiftExpr) = do

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

emit ">"

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

return Nothing

genRelExpr (LessEqual relExpr shiftExpr) = do

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

emit "<="

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

return Nothing

genRelExpr (GreaterEqual relExpr shiftExpr) = do

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

emit ">="

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

return Nothing

genRelExpr (InstanceOf relExpr shiftExpr) = do

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

emit " instanceof "

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

return Nothing

genRelExpr (InObject relExpr shiftExpr) = do

relExpr' <- genRelExpr relExpr

genMaybe genPrimExpr relExpr'

emit " in "

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

return Nothing

-- | Shift operators

genShiftExpr :: ShiftExpr -> JSCC (Maybe PrimaryExpr)

genShiftExpr (AddExpr addExpr) = do

addExpr' <- genAddExpr addExpr

return addExpr'

genShiftExpr (ShiftLeft shiftExpr addExpr) = do

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

emit "<<"

addExpr' <- genAddExpr addExpr

genMaybe genPrimExpr addExpr'

return Nothing

genShiftExpr (ShiftRight shiftExpr addExpr) = do

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

emit ">>"

addExpr' <- genAddExpr addExpr

genMaybe genPrimExpr addExpr'

return Nothing

genShiftExpr (ShiftRight2 shiftExpr addExpr) = do

shiftExpr' <- genShiftExpr shiftExpr

genMaybe genPrimExpr shiftExpr'

emit ">>>"

addExpr' <- genAddExpr addExpr

genMaybe genPrimExpr addExpr'

return Nothing

-- | Additive operators

genAddExpr :: AddExpr -> JSCC (Maybe PrimaryExpr)

genAddExpr (MultExpr multExpr) = do

multExpr' <- genMultExpr multExpr

return multExpr'

genAddExpr (Plus addExpr multExpr) = do

a <- genAddExpr addExpr

if isJust a then genPrimExpr $ fromJust a else emit ""

emit "+"

b <- genMultExpr multExpr

res <- peSimpCalc genPrimExpr a b '+'

return res

genAddExpr (Minus addExpr multExpr) = do

a <- genAddExpr addExpr

if isJust a then genPrimExpr $ fromJust a else emit ""

emit "-"

b <- genMultExpr multExpr

res <- peSimpCalc genPrimExpr a b '-'

return res

-- | Multiplicative operators

genMultExpr :: MultExpr -> JSCC (Maybe PrimaryExpr)

genMultExpr (UnaryExpr unaryExpr) = do

unaryExpr' <- genUnaryExpr unaryExpr

return unaryExpr'

genMultExpr (Times multExpr unaryExpr) = do

a <- genMultExpr multExpr

if isJust a then genPrimExpr $ fromJust a else emit ""

emit "\*"

b <- genUnaryExpr unaryExpr

res <- peSimpCalc genPrimExpr a b '\*'

return res

genMultExpr (Div multExpr unaryExpr) = do

a <- genMultExpr multExpr

if isJust a then genPrimExpr $ fromJust a else emit ""

emit "/"

b <- genUnaryExpr unaryExpr

res <- peSimpCalc genPrimExpr a b '/'

return res

genMultExpr (Mod multExpr unaryExpr) = do

a <- genMultExpr multExpr

if isJust a then genPrimExpr $ fromJust a else emit ""

emit "%"

b <- genUnaryExpr unaryExpr

res <- peSimpCalc genPrimExpr a b '%'

return res

-- | Unary operators

--

-- Spaces are removed in cleanup function along with semicolons

genUnaryExpr :: UnaryExpr -> JSCC (Maybe PrimaryExpr)

genUnaryExpr (PostFix postFix) = do

postFix' <- genPostFix postFix

return postFix'

genUnaryExpr (Delete unaryExpr) = do

emit "delete "

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (Void unaryExpr) = do

emit "void "

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (TypeOf unaryExpr) = do

emit "typeof "

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (PlusPlus unaryExpr) = do

emit " ++"

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (MinusMinus unaryExpr) = do

emit " --"

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (UnaryPlus unaryExpr) = do

emit "+"

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (UnaryMinus unaryExpr) = do

emit "-"

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (Not unaryExpr) = do

emit "!"

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

genUnaryExpr (BitNot unaryExpr) = do

emit "~"

unaryExpr' <- genUnaryExpr unaryExpr

genMaybe genPrimExpr unaryExpr'

return Nothing

-- | Post fix

genPostFix :: PostFix -> JSCC (Maybe PrimaryExpr)

genPostFix (LeftExpr leftExpr) = do

leftExpr' <- genLeftExpr leftExpr

return leftExpr'

genPostFix (PostInc leftExpr) = do

leftExpr' <- genLeftExpr leftExpr

genMaybe genPrimExpr leftExpr'

emit "++ "

return Nothing

genPostFix (PostDec leftExpr) = do

leftExpr' <- genLeftExpr leftExpr

genMaybe genPrimExpr leftExpr'

emit "-- "

return Nothing

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: CompUtils \*

\* Purpose: Compression utilities. Various methods are employed including \*

\* variable shrinking, string and number optimisation, and \*

\* partial evaluation. \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | Compression utilities. Various methods are employed including variable shrinking,

-- string and number optimisation, and partial evaluation.

module CompUtils where

-- Standard library imports

import Numeric (showHex)

import Maybe

-- JSHOP module imports

import ParseTree

import CodeCompMonad

-- | Type synonum for the JS code compression monad

type JSCC a

= CC String () a

-- | Helper function to handle Maybe types

genMaybe :: (a -> JSCC()) -> (Maybe a) -> JSCC()

genMaybe genFunc mbExpr

= case mbExpr of

Just expr -> genFunc expr

Nothing -> return ()

{- | Optimisation note:

Literal numbers can be expressed in hexadecimal notation in JavaScript.

Hex numbers begin with \"0x\" so are not always shorter. To find the point

at which it becomes shorter to express a number in hex, I wrote this function:

> findPoint :: Integer -> IO ()

> findPoint n = do

> let hex = "0x" ++ (showHex n "")

> if length (show n) > length hex

> then putStrLn $ hex ++ " (" ++ (show (length hex)) ++ ") is shorter than "

> ++ show n ++ " (" ++ (show (length (show n))) ++ ")"

> else do

> putStrLn $ hex ++ " (" ++ (show (length hex)) ++ "), " ++ (show n)

> ++ " (" ++ (show (length (show n))) ++ ")"

> findPoint (n+1)

I ran it in ghci and this was the relevant result:

> \*Main> findPoint 999999999995

> 0xe8d4a50ffb (12), 999999999995 (12)

> 0xe8d4a50ffc (12), 999999999996 (12)

> 0xe8d4a50ffd (12), 999999999997 (12)

> 0xe8d4a50ffe (12), 999999999998 (12)

> 0xe8d4a50fff (12), 999999999999 (12)

> 0xe8d4a51000 (12) is shorter than 1000000000000 (13)

The chances of there ever being a literal number this large expressed in a

script is very low, but at least I've got it covered! As far as I know, no

other compressor does this.

The maximum integer in JavaScript is 9007199254740992 so it is possible for this

compression technique to take effect.

-}

compInt :: Integer -> String

compInt n = if n < 1000000000000 then show n

else "0x" ++ showHex n ""

{- | Optimisation note:

JavaScript strings can be wrapped in single or double quotes. Escaping quotes

requires an extra character (backslash), so it is sometimes possible to switch

the type of quote used to wrap the string so that it is no longer necessary to

escape.

This function optimises strings with escaped quotes in them.

For example:

> 'te\'st' -> "te'st"

> "te\"st" -> 'te"st'

> "te'st\"in\"g" -> 'tes\'st"in"g'

-}

compStr :: Maybe Char -> String -> String

compStr mbQ str =

case mbQ of

Nothing -> if length fullStr < length str then fullStr else str

Just q -> q:(genStrBody (init(tail str)) q) ++ [q]

where

origType = head str

altType = if origType == '\'' then '"' else '\''

qList = getEscQ str

qType = if countElem origType qList > countElem altType qList

then altType

else origType

strBody = genStrBody (init(tail str)) qType

fullStr = qType:(strBody ++ [qType])

genStrBody :: String -> Char -> String

genStrBody [] \_ = []

genStrBody ('\\':'\'':xs) '\'' = '\\':'\'':(genStrBody xs '\'')

genStrBody ('\\':'\'':xs) qType = '\'':(genStrBody xs qType)

genStrBody ('\'':xs) '\'' = '\\':'\'':(genStrBody xs '\'')

genStrBody ('\'':xs) qType = '\'':(genStrBody xs qType)

genStrBody ('\\':'"':xs) '"' = '\\':'"':(genStrBody xs '"')

genStrBody ('\\':'"':xs) qType = '"':(genStrBody xs qType)

genStrBody ('"':xs) '"' = '\\':'"':(genStrBody xs '"')

genStrBody ('"':xs) qType = '"':(genStrBody xs qType)

genStrBody (x:xs) qType = x:(genStrBody xs qType)

getEscQ :: String -> [Char]

getEscQ [] = []

getEscQ ('\\':'\'':xs) = '\'':(getEscQ xs)

getEscQ ('\\':'"':xs) = '"':(getEscQ xs)

getEscQ (x:xs) = getEscQ xs

countElem :: Eq a => a -> [a] -> Int

countElem i = length . filter (i==)

{- | Optimisation note:

This function partially evaluates literal sums.

For example:

> 1 + 2 becomes 3

> -34.3 + 23.784 becomes -10.516

> 34 \* 12 / a becomes 408/a

KNOWN BUG: Double arithmetic sometimes throws up silly results due to

precision errors.

> E.g. 0.2 + 0.1 gives 0.30000000000000004

In these cases, the original sum will be displayed (if it is shorter).

This is not ideal as 0.3 would be shorter still. This is a documented bug:

<http://hackage.haskell.org/trac/ghc/ticket/5856>

-}

peSimpCalc :: (PrimaryExpr -> JSCC()) -- ^ genPrimExpr function

-> Maybe PrimaryExpr -- ^ First operand

-> Maybe PrimaryExpr -- ^ Second operand

-> Char -- ^ Operator

-> JSCC (Maybe PrimaryExpr)

peSimpCalc gen (Just (ExpLiteral a)) (Just (ExpLiteral b)) op = do

pop -- operator

pop -- first operand

let (x, mbY) = simpCalcLit a b op

if litLength x > origLength then do

genLiteral a

emit [op]

genLiteral b

return Nothing

else if isJust mbY then do

genLiteral x

emit [op]

genLiteral $ fromJust mbY

return Nothing

else return $ Just (ExpLiteral x)

where

origLength = litLength a + litLength b + 1 -- The 1 is the op

litLength :: Literal -> Int

litLength (LNull) = 4 -- null

litLength (LBool \_) = 2 -- !0 or !1

litLength (LInt x) = length $ show x

litLength (LFloat x) = length $ dropPrefix $ show $ roundIfInt x

litLength (LStr s) = length s

peSimpCalc gen Nothing (Just (ExpLiteral b)) \_

= genLiteral b >> return Nothing

peSimpCalc gen \_ (Just b) \_

= gen b >> return Nothing

peSimpCalc \_ \_ \_ \_

= return Nothing

simpCalcLit :: Literal -> Literal -> Char -> (Literal, Maybe Literal)

simpCalcLit (LInt a) (LInt b) op

= (LFloat (calcInt a b op), Nothing)

simpCalcLit (LInt a) (LFloat b) op

= (LFloat (calcDouble (fromInteger a) b op), Nothing)

simpCalcLit (LFloat a) (LInt b) op

= (LFloat (calcDouble a (fromInteger b) op), Nothing)

simpCalcLit (LFloat a) (LFloat b) op

= (LFloat (calcDouble a b op), Nothing)

simpCalcLit (LStr a) (LStr b) '+'

= (LStr (concatStr a b), Nothing)

simpCalcLit a b op

= (a, Just b)

calcInt :: Integer -> Integer -> Char -> Double

calcInt x y '%' = fromInteger $ x `mod` y

calcInt x y op = calcDouble (fromInteger x) (fromInteger y) op

calcDouble :: Double -> Double -> Char -> Double

calcDouble x y op = case op of

'+' -> x + y

'-' -> x - y

'\*' -> x \* y

'/' -> x / y

\_ -> 0

-- | Two literal strings can be concatenated, however their quote types may

-- be different. This function sorts it all out!

concatStr :: String -> String -> String

concatStr x@('\'':xs) ('\'':ys) = init x ++ ys

concatStr x@('"':xs) ('"':ys) = init x ++ ys

concatStr x@('"':xs) y@('\'':ys) = init x ++ (tail $ compStr (Just '"') y)

concatStr x@('\'':xs) y@('"':ys) = init x ++ (tail $ compStr (Just '\'') y)

concatStr x y = x ++ y

-- | Ints written as Doubles have \".0\" at the end, which is a waste of

-- chars, hence this function.

roundIfInt :: (RealFrac a, Integral b) => a -> Either a b

roundIfInt n = if isInt n then Right (round n) else Left n

where

isInt :: RealFrac a => a -> Bool

isInt x = x == fromInteger (round x)

-- | Removes the first word from a string. Useful for removing Eithers

-- or Maybes e.g. \"Left \", \"Right \", \"Just\", \"LInt\" etc.

dropPrefix :: String -> String

dropPrefix str = tail $ dropWhile (/=' ') str

{- | Literal

Optimisation note (literal bools):

True can be represented as 1 in JavaScript, however if you simply put 1, it will

be assumed to be an integer. Adding ! evaluates it as a boolean expression.

Consequently, true can be expressed as \"not false\" and false can be

expressed as \"not true\".

-}

genLiteral :: Literal -> JSCC()

genLiteral (LNull) = emit "null"

genLiteral (LBool True) = emit "!0"

genLiteral (LBool False) = emit "!1"

genLiteral (LInt int) = emit $ compInt int

genLiteral (LFloat float) = emit $ dropPrefix $ show $ roundIfInt float

genLiteral (LStr str) = emit $ compStr Nothing str

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: Utilities \*

\* Purpose: A set of helper functions used by Main and TestSuite \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

-- | A set of helper functions used by Main and TestSuite

module Utilities where

-- Standard library imports

import Control.Monad.Identity

import Control.Monad.Error

import Control.Monad.State

import System.CPUTime

import Text.Printf

import Data.List

-- JSHOP module imports

import Token

import Lexer

import LexerMonad

import ParseTree

import ParseMonad hiding (nl) -- Causes ambiguity with nl function (PP)

import Parser

import Diagnostics

import CodeCompMonad

import CodeCompressor

-- | Parses a string of JavaScript, returning either an error string or a Parse Tree

-- and the lexer state

parseJS :: String -> Either String (Tree, LexerState)

parseJS str

= runIdentity $ runErrorT $ runStateT parse (startState str)

-- | Generates compresses JavaScript from a Parse Tree

genJS :: Tree -> String

genJS tree = result

where

(mbCode, msgs) = runD (genCode tree)

result = case mbCode of

Nothing -> "No code generated"

Just code -> cleanup $ concat code

-- | Saves a given string to a given filename

saveFile :: String -> String -> IO()

saveFile file output = do

writeFile file output

-- | Shows the ratio of input to output

showRatio :: [a] -> [a] -> String

showRatio inp out = "Reduced by " ++ show reduced ++ " chars, \t"

++ take 5 (show percent) ++ "% of original."

where

reduced = (length inp) - (length out)

percent = intToFloat (length out) / intToFloat (length inp) \* 100

-- | Calulates the ratio of input to output

calcRatio :: [a] -> [a] -> Float

calcRatio inp out = percent

where

reduced = (length inp) - (length out)

percent = intToFloat (length out) / intToFloat (length inp) \* 100

intToFloat :: Int -> Float

intToFloat n = fromInteger (toInteger n)

mean :: (Real a, Fractional b) => [a] -> b

mean xs = realToFrac (sum xs) / (fromIntegral $ length xs)

-- | Calculates the total execution time based on the given start time

execTime :: Integer -> IO()

execTime startTime = do

endTime <- getCPUTime

let execTime = calcTime startTime endTime

printf "\nTotal execution time: %0.3f secs" execTime

calcTime :: Integer -> Integer -> Double

calcTime start end = (fromIntegral (end - start)) / (10^12)

-- | Final clean to remove things which cannot easily be removed when in Parse Tree format.

--

-- What about if we're in a regex? :S

cleanup :: String -> String

cleanup [] = []

-- Semi-colons

cleanup (';':'}':xs) = '}':(cleanup xs)

-- Post fix and unary spaces

cleanup ('+':' ':'+':xs) = '+':' ':'+':(cleanup xs)

cleanup ('+':' ':x:xs) = '+':x:(cleanup xs)

cleanup ('-':' ':'-':xs) = '-':' ':'-':(cleanup xs)

cleanup ('-':' ':x:xs) = '-':x:(cleanup xs)

-- New Object and Array declarations

cleanup str

| Just xs <- stripPrefix "new Object()" str = '{':'}':(cleanup xs)

| Just xs <- stripPrefix "new Object;" str = '{':'}':';':(cleanup xs)

| Just xs <- stripPrefix "new Array()" str = '[':']':(cleanup xs)

| Just xs <- stripPrefix "new Array;" str = '[':']':';':(cleanup xs)

-- Return undefined

| Just xs <- stripPrefix "return undefined;" str = "return;" ++ (cleanup xs)

-- Leave everything else alone

cleanup (x:xs) = x:(cleanup xs)

-- | Pretty print the ParseTree

--

-- So far this just puts a blank line between each source (function or statement) and does some

-- simple indentations for each branch. Suffice to say I'm not putting much effort into making

-- the Parse Tree readable as it's not that important. It just needs to be debuggable for my sake.

ppTree :: Tree -> Bool -> String

ppTree (Tree sources) remBloat

= if remBloat then

ppRemBloat tree

else

tree

where

tree = ppIndent $ concat [show x ++ "\n\n" | x <- sources]

-- | Make newline and indent every time a branch terminates.

ppIndent :: String -> String

ppIndent [] = []

ppIndent (')':' ':xs) = ')':'\n':'\t':(ppIndent xs)

ppIndent (')':',':xs) = ')':',':'\n':'\t':(ppIndent xs)

ppIndent (']':' ':xs) = ']':'\n':'\t':(ppIndent xs)

ppIndent (x:xs) = x:(ppIndent xs)

-- | HIGHLY EXPERIMENTAL. This is designed to remove a lot of the bloat in the Parse Tree by

-- taking out all the logOr, relExpr, ShiftExpr type stuff. NOTE, sometimes this stuff is

-- necessary to understand the structure, so it's best to leave it in. However, for browsing,

-- it can be easier to remove it.

ppRemBloat :: String -> String

ppRemBloat [] = []

ppRemBloat str

| Just xs <- stripPrefix "CondExpr" str = ppRemBloat $ shrink xs

| Just xs <- stripPrefix "ShiftExpr" str = ppRemBloat $ shrink xs

| Just xs <- stripPrefix "AddExpr" str = ppRemBloat $ shrink xs

ppRemBloat (x:xs) = x:(ppRemBloat xs)

shrink :: String -> String

shrink xs = "..." ++ '(':(leaf branch) ++ ')':(dropWhile (/= ')') xs)

where

branch = takeWhile (/= ')') xs

leaf = reverse . takeWhile (/= '(') . reverse

-- | EXPERIMENTAL. Rudimentaty JS pretty printer.

ppOutput :: String -> ShowS

ppOutput str = ppOutput' str 0

where

ppOutput' :: String -> Int -> ShowS

-- End of file

ppOutput' [] \_ = showString ""

-- End of statement, begin new line

ppOutput' (';':xs) n = showChar ';' . nl . indent n . ppOutput' xs n

-- Begining of block statement, indent + 1

ppOutput' ('{':xs) n = showChar '{' . nl . indent (n+1) . ppOutput' xs (n+1)

-- End of block statement, indent - 1

ppOutput' ('}':xs) n = nl . indent (n-1) . showChar '}'

. nl . indent (n-1) . ppOutput' xs (n-1)

-- Any other char, continue

ppOutput' (x:xs) n = showChar x . ppOutput' xs n

-- Pretty printing utils

indent :: Int -> ShowS

indent n = showString (take (2 \* n) (repeat ' '))

nl :: ShowS

nl = showChar '\n'

spc :: ShowS

spc = showChar ' '

ppSeq :: Int -> (Int -> a -> ShowS) -> [a] -> ShowS

ppSeq \_ \_ [] = id

ppSeq n pp (x:xs) = pp n x . ppSeq n pp xs

-- | Given a JavaScript string, writes a list of tokens (1 per line)

--

-- Non monad version of the lexer

nonMLexer :: String -> IO()

nonMLexer str = do

let ts = tokens str

putStrLn $ concat [show t ++ "\n"| t <- ts]

nonMLexFile :: String -> IO()

nonMLexFile file = do

input <- readFile file

nonMLexer input

-- | Converts a JavaScript string into a list of tokens

tokens :: String -> [Token]

tokens [] = []

tokens str' =

case str' of

('\n':xs) -> tokens xs

('/':'\*':xs) -> nonMScanComment xs

\_ -> ts

where

lexResult = lexer str'

ts = case lexResult of

Left (t, rest) -> (t:tokens rest)

Right rest -> tokens rest

-- | Scans multi-line comments

nonMScanComment :: String -> [Token]

nonMScanComment str = do

case str of

('\n':xs) -> nonMScanComment xs

('\*':'/':xs) -> (Other "COMMENT":tokens xs)

(\_:xs) -> nonMScanComment xs

{-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JSHOP \*

\* \*

\* Module: TestSuite \*

\* Purpose: A set of tests to run on a selection of inputs \*

\* Author: Nick Brunt \*

\* \*

\* Copyright (c) Nick Brunt, 2011 - 2012 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-}

module TestSuite where

-- Standard library imports

import System.Directory

import System.CPUTime

import Maybe

import Control.Monad

-- JSHOP module imports

import Utilities

-- Test result data structures

data TestResults =

TestResults {

testNum :: Int,

message :: String,

strucTests :: [Test],

libTests :: [Test],

time :: Double,

average :: Float

}

deriving (Read, Show)

data Test =

Test {

name :: String,

result :: Bool, -- True = pass, False = fail

errorMsg :: String,

inputSize :: Int,

outputSize :: Int,

reduction :: Int,

percentage :: Float

}

deriving (Read, Show)

defaultTest :: Test

defaultTest =

Test {

name = "",

result = False,

errorMsg = "",

inputSize = 0,

outputSize = 0,

reduction = 0,

percentage = 0

}

testResultsFile :: String

testResultsFile = "tests/testResults.log"

-- Structure tests

funcFile :: String

funcFile = "tests/structure/functions.js"

exprFile :: String

exprFile = "tests/structure/expressions.js"

statFile :: String

statFile = "tests/structure/statements.js"

runTests :: Maybe [String] -> IO()

runTests mbArgs = do

startTime <- getCPUTime

putStrLn "Starting test suite"

putStrLn "-------------------\n"

let msg = head $ fromMaybe ["No message"] mbArgs

-- Structure tests run every possible JavaScript control structure

-- through the program to test that they can be fully parsed.

funcTest <- runParseTest (defaultTest {name="Functions"}) funcFile

exprTest <- runParseTest (defaultTest {name="Expressions"}) exprFile

statTest <- runParseTest (defaultTest {name="Statements"}) statFile

-- Library tests run a set of JavaScript libraries through the program

-- to test real world code and also to check compression ratios.

-- Get list of files in libraries directory

files <- getDirectoryContents "tests/libraries"

-- Filter out ".." and "." and add path

let names = filter (\x -> head x /= '.') files

let libs = ["tests/libraries/" ++ f | f <- names]

let libTests = [defaultTest {name=libName} | libName <- names]

libTests' <- zipWithM runParseTest libTests libs

nextTestNum <- getNextTestNum

endTime <- getCPUTime

let testResults = TestResults {

testNum = nextTestNum,

message = msg,

strucTests = funcTest:exprTest:[statTest],

libTests = libTests',

time = calcTime startTime endTime,

average = mean $ map percentage libTests'

}

-- Pretty print results

putStrLn $ ppTestResults testResults ""

-- Write results to file

if msg /= "No message" then

if nextTestNum == 0 then

writeFile testResultsFile (show testResults)

else

appendFile testResultsFile ('\n':(show testResults))

else

putStr ""

runParseTest :: Test -> String -> IO Test

runParseTest test file = do

input <- readFile file

let parseOutput = parseJS input

case parseOutput of

Left error -> do

return (test {

result = False,

errorMsg = error,

inputSize = length input

})

Right (tree, state) -> do

let output = genJS tree

let outFile = outTestFile file

saveFile outFile output

-- Write to file

return (test {

result = True,

inputSize = length input,

outputSize = length output,

reduction = (length input) - (length output),

percentage = calcRatio input output

})

where

outTestFile :: String -> String

outTestFile inFile = "tests/outputLibraries/" ++ minFile

where

file = reverse $ takeWhile (/='/') $ reverse inFile

minFile = reverse $ takeWhile (/='.') (reverse file)

++ ".nim" ++ dropWhile (/='.') (reverse file)

showPastResults :: IO()

showPastResults = do

f <- readFile testResultsFile

let results = [ppTestResults tr "" | tr <- map read (lines f)]

mapM\_ putStrLn results

showLastResult :: IO()

showLastResult = do

f <- readFile testResultsFile

putStrLn $ ppTestResults (read $ last (lines f)) ""

showResult :: Int -> IO()

showResult n = do

f <- readFile testResultsFile

putStrLn $ ppTestResults (read $ (lines f) !! n) ""

showAverages :: IO()

showAverages = do

putStrLn "Percentage of output to input:\n"

f <- readFile testResultsFile

let tests = [tr | tr <- map read (lines f)]

let strings = ["Test " ++ (show $ testNum t) ++

" average:\t" ++ (show $ average t) ++

" \t" ++ (message t) | t <- tests]

mapM\_ putStrLn strings

getNextTestNum :: IO Int

getNextTestNum = do

f <- readFile testResultsFile

return $ length $ lines f

ppTestResults :: TestResults -> ShowS

ppTestResults (TestResults {testNum = n, message = m,

strucTests = sts, libTests = lts,

time = t, average = a}) =

showString "Test number" . spc . showString (show n) . nl

. indent 1 . showString "Message:" . spc . showString m . nl . nl

. indent 1 . showString "STRUCTURE TESTS" . nl

. ppSeq 1 ppTest sts . nl

. indent 1 . showString "LIBRARY TESTS" . nl

. ppSeq 1 ppTest lts . nl

. showString "Completed in" . spc . showString (take 5 (show t))

. spc . showString "seconds" . nl

. showString "Average compression:" . spc

. showString (take 5 (show a)) . showChar '%' . nl

ppTest :: Int -> Test -> ShowS

ppTest idnt (Test {name = n, result = r, errorMsg = e,

inputSize = i, outputSize = o,

reduction = d, percentage = p}) =

indent idnt . showString n . nl

. indent (idnt+1) . showString "Result:" . spc . ppResult r . nl

. ppErrorMsg (idnt+1) e

. indent (idnt+1) . showString "Input size:" . spc . showString (show i) . nl

. indent (idnt+1) . showString "Output size:" . spc . showString (show o) . nl

. indent (idnt+1) . showString "Reduced by:" . spc . showString (show d) . nl

. indent (idnt+1) . showString "Percentage of original:" . spc

. showString (take 5 (show p)) . showChar '%' . nl

ppResult :: Bool -> ShowS

ppResult True = showString "PASS"

ppResult False = showString "FAIL"

ppErrorMsg :: Int -> String -> ShowS

ppErrorMsg \_ "" = showString ""

ppErrorMsg n msg = indent n . showString "Error message:"

. spc . showString msg . nl

##############################################################################

# #

# Makefile for JSHOP #

# Copyright (c) Nick Brunt, 2011-2012 #

# #

# Loosely based on the HMTC makefile #

# Copyright (c) Henrik Nilsson, 2006 #

# http://www.cs.nott.ac.uk/~nhn/ #

# #

##############################################################################

# Default operation (when you only call "make")

all**:** jshop

#-----------------------------------------------------------------------------

# Source files:

#-----------------------------------------------------------------------------

# Alex (lexer) source files

alexFiles **=** \

Lexer.x

# Happy (parser) source files

happyFiles **=** \

Parser.y

# Haskell source files

haskellFiles **=** \

Token.hs \

Lexer.hs \

LexerMonad.hs \

ParseTree.hs \

ParseMonad.hs \

Parser.hs \

Diagnostics.hs \

CodeCompMonad.hs \

CompUtils.hs \

Analyser.hs \

CodeCompressor.hs \

Utilities.hs \

TestSuite.hs \

Main.hs

# Definition of .hi and .o files for cleaning purposes

hs\_interfaces **:=** $(haskellFiles:.hs=.hi)

hs\_objects **:=** $(haskellFiles:.hs=.o)

#-----------------------------------------------------------------------------

# Compile lexer:

#-----------------------------------------------------------------------------

# Options:

# g - optimise for ghc

# i - generate info file

%.hs**:** %.x

#

# Compiling lexer...

#

alex -gi $(alexFiles)

#-----------------------------------------------------------------------------

# Compile parser:

#-----------------------------------------------------------------------------

# Options:

# g - optimise for ghc

# i - generate info file

# a - use array-based shift reduce parser. Much faster when combined with -g

# d - prints debug info at runtime

%.hs**:** %.y

#

# Compiling parser...

#

happy -agi $(happyFiles)

#-----------------------------------------------------------------------------

# Compile JSHOP:

#-----------------------------------------------------------------------------

# Options:

# make - compile to exe

# O2 - optimise (level 2)

# w - hide warnings

# o - specify output filename

jshop**:** $(haskellFiles)

#

# Compiling JSHOP...

#

ghc --make -O2 -w Main -o jshop.exe

#-----------------------------------------------------------------------------

# Generate Documentation:

#-----------------------------------------------------------------------------

# Options:

# HsColour:

# css - output in HTML 4.01 with CSS

# anchor - adds an anchor to every entity for use with Haddock

#

# haddock:

# odir - output directory

# html - generate documentation in HTML format

# source-base - adds link to source code directory

# source-module - adds link to each individual module

# source-entity - adds link to each individual entity

# title - title to appear at the top of each page

# w - supress warnings

doc**:** clean-doc $(haskellFiles)

#

# Generating syntax highlighted HTML source files...

#

for file in $(haskellFiles) ; do \

HsColour -css -anchor $$file > doc/src/`basename $$file .hs`.html ; \

done

#

# Generating documentation...

#

haddock --odir=doc --html --source-base=src/ --source-module=src/%M.html --source-entity=src/%M.html#%N --title="JSHOP" $(haskellFiles) -w

#-----------------------------------------------------------------------------

# Cleaning:

#-----------------------------------------------------------------------------

# Remove all .hi and .o files

clean**:**

#

# Removing Haskell interfaces and objects...

#

-$(RM) $(hs\_interfaces) $(hs\_objects)

# Remove all documentation files (except css)

clean-doc**:**

#

# Cleaning documentation...

#

-rm doc/\*.\*

-rm doc/src/\*.html

# Remove ALL unnecessary files leaving only absolute source

really-clean**:** clean clean-doc

#

# Removing extraneous files...

#

-$(RM) Parser.hs

-$(RM) Lexer.hs

-$(RM) error.log

-$(RM) Lexer.info

-$(RM) Parser.info

-$(RM) jshop

-- PROTOTYPE

**module** JSHOP **where** -- JavaScript Haskell OPtimiser

**import** Char

**import** Text.Regex

**import** Text.Regex.PCRE

**import** Text.Regex.Base.RegexLike

**import** System

**import** ModRegex

**type** Code = [Char]

**type** Regexp = [Char]

**type** Matches = [(Int,Int)]

-- \*\* Regular Expressions \*\*

-- Single line comments e.g. // comment

sComments = "//[^\n]\*"

-- Multi line comments e.g. /\* comment \*/

mComments = "/\\\*[^\*]\*\\\*+([^/][^\*]\*\\\*+)\*/"

-- Conditional comments e.g. /\*@ comment @\*/

cComments = "/\\\*@[^\*]\*\\\*+([^/][^\*]\*(@\\\*)+)\*/"

-- http://wordaligned.org/articles/string-literals-and-regular-expressions

-- Single quote strings e.g. 'a string'

sString = "'([^'\\\\]|\\\\.)\*'"

-- Double quote strings e.g. "a string"

dString = "\"([^\"\\\\]|\\\\.)\*\""

-- Regular expression e.g. /pattern/modifiers

jsRegex = "[^\*/]/(\\\\[/\\\\]|[^\*/])(\\\\.|[^/\n\\\\])\*/[gim]\*"

-- Whitespace e.g. spaces, tabs, newlines etc. equivalent to "[ \t\r\n\v\f]+"

space = "\\s+"

-- Incrementors or decrementors e.g. a++ +b

plusplus = "([+-])"++space++"([+-])"

-- Word boundaries e.g. var a

wordBound = "\\b\\s+\\b"

-- Dollar variable names e.g. var $ in

dollarVar = "\\b\\s+\\$\\s+\\b"

-- Variables starting with dollar e.g. $this

startDollar = "\\$\\s+\\b"

-- Variables ending with dollar e.g. something$

endDollar = "\\b\\s+\\$"

-- Fixed decimal number corner case

{-

This is a strange corner case. In JavaScript you can call certain functions on numbers in two ways. There's obvious bracketed way: (12).toFixed(2) which is fine, or there's the unconventional shorthand way: 12 .toFixed(2) with a space instead of brackets. Without the following regex to detect it, the space would be removed by this program.

-}

spaceShorthand = "(\\d)\\s+(\\.\\s\*[a-z\\$\_\\[(])"

-- Empty for loop expression e.g. for(;;)

emptyFor = "for\\(;\\)"

-- Semicolons before closing braces

semiBracket = ";+\\s\*([};])"

-- Returns a list of the offset and length of each match to pat in cs

getMatches :: Regexp -> Code -> Matches

getMatches pat cs = getAllMatches (match regex cs) :: [(MatchOffset,MatchLength)]

**where**

regex = makeRegexOpts (defaultCompOpt + compCaseless) defaultExecOpt pat

noConflict :: Matches -> (Int,Int) -> Bool

noConflict [] **\_** = True

noConflict ((ro,rl):rs) (o,l)

= not (o >= ro && o < (ro + rl)) && noConflict rs (o,l)

{-

Recursively replace each match. Note that this is done in reverse to prevent the offsets getting out of line when each match is removed. This could be compensated for using the length of the match just removed, but reversing the list is the simplest solution.

-}

remMatches :: Code -> Matches -> Code

remMatches cs [] = cs

remMatches [] **\_** = []

remMatches cs ms = remMatches (take offset cs) (reverse (tail rms)) ++ drop (offset + length) cs

**where**

rms = reverse ms

(offset, length) = head rms

remove :: Regexp -> (Code -> Matches) -> Code -> Code

remove pat fExcep cs = remMatches cs validMatches

**where**

validMatches = filter (noConflict exceptions) matches

matches = getMatches pat cs

exceptions = fExcep cs

comExceptions :: Code -> Matches

comExceptions cs = getMatches dString cs ++

getMatches sString cs ++

getMatches cComments cs ++

getMatches jsRegex cs

spaceExceptions :: Code -> Matches

spaceExceptions cs = getMatches dString cs ++

getMatches sString cs ++

getMatches cComments cs ++

getMatches jsRegex cs ++

getMatches plusplus cs ++

getMatches wordBound cs ++

getMatches dollarVar cs ++

getMatches startDollar cs ++

getMatches endDollar cs ++

getMatches spaceShorthand cs

{-

Remove comments. Note that multi line comments must be removed first otherwise the closing \*/ would get removed accidentally in a situation like this: /\* comment // comment \*/

-}

remComments :: Code -> Code

remComments = remove sComments comExceptions .

remove mComments comExceptions

cleanup :: Code -> Code

cleanup = pcreSubRegex emptyFor "for(;;)" . -- Fixes empty for expressions: for(;) -> for(;;)

pcreSubRegex semiBracket "\\1" . -- Mucks up empty for expressions: for(;;) -> for(;)

pcreSubRegex wordBound " " .

pcreSubRegex endDollar " $" .

pcreSubRegex startDollar "$ " .

pcreSubRegex dollarVar " $ " .

pcreSubRegex plusplus "\\1 \\2" .

pcreSubRegex spaceShorthand "\\1 \\2"

remWhitespace :: Code -> Code

remWhitespace = cleanup .

remove space spaceExceptions

compress :: Code -> Code

compress = remWhitespace . remComments

compFile :: String -> IO()

compFile file = **do**

input <- readFile file

**let** output = compress input

putStrLn ("\n" ++ msg ++ "\n" ++ underline ++ "\n")

putStrLn output

putStrLn ("\n" ++ underline ++ "\n")

putStrLn ("Info\n----\n")

putStrLn ("Input size: " ++ show (length input) ++ " bytes")

putStrLn ("Output size: " ++ show (length output) ++ " bytes")

putStrLn ("Compressed by: " ++ show (length input - length output) ++ " bytes")

**where**

msg = ("Compressed " ++ file ++ " follows")

underline = (replicate (length msg) '-')

main :: IO()

main = **do**

args <- getArgs

**case** args **of**

[] -> **do**

-- No specified file or code, ask for user input

putStr "Please enter a filename: "

inFile <- getLine

compFile inFile

(inFile:**\_**) -> **do**

-- Return file with original filename plus ".min" e.g. test.min.js

input <- readFile inFile

**let** outFile = takeWhile (/='.') inFile ++ ".min" ++ dropWhile (/='.') inFile

writeFile outFile (compress input)

-- Some testing functions

findPat :: [Char] -> Code -> IO()

findPat pattern code = **do**

putStrLn ("\nMatches:\n--------\n\nOffset, Length: Code extract\n")

putStrLn (concat [show offset ++ ", "

++ show length ++ ": "

++ take length (drop offset code)

++ "\n" | (offset,length) <- matches])

**where**

matches = getMatches pattern code

findPatFile :: String -> String -> IO()

findPatFile file pattern = **do**

input <- readFile file

findPat pattern input

1. <http://www.haskell.org/alex/> [↑](#footnote-ref-1)
2. <http://www.haskell.org/happy/> [↑](#footnote-ref-2)
3. A higher-order function is a function which accepts a function as an argument and returns another function as the result. [↑](#footnote-ref-3)
4. <http://www.haskell.org/haskellwiki/Parsec> [↑](#footnote-ref-4)
5. Adapted from the definition at <http://www.haskell.org/haskellwiki/Monad> [↑](#footnote-ref-5)
6. http://www.jslint.com/ [↑](#footnote-ref-6)
7. <http://www.haskell.org/haddock/> [↑](#footnote-ref-7)
8. <http://www.cs.york.ac.uk/fp/darcs/hscolour/> [↑](#footnote-ref-8)