



UNIVERSITY OF GOTHENBURG

JSHC

- JavaScript Haskell Compiler

Bachelor's thesis in Computer Science

STAFFAN BJÖRNESJÖ

PETER HOLM

University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sverige 2011
Kandidatarbete/rapport nr 2011:032

JSHC
JavaScript Haskell Compiler

© Staffan Björnesjö, Peter Holm
June 2011
Department of Computer Science and Engineering
University of Gothenburg
SE-412 96 Göteborg
Sweden

Abstract

Development of large web based applications using scripting languages such as PHP or JavaScript can be difficult given the lack of static type checking and abstraction through module systems in many such languages. We argue that static type checking and module systems can help programmers avoid common errors when developing large web applications. The literature indicates that adding an extra layer of abstraction on top of the scripting languages that provides these features can be a solution to this problem.

We suggest a solution where the statically typed functional language Haskell is compiled to JavaScript code, enabling programmers to create web applications in a modular and type safe environment.

An implementation of a compiler, supporting compilation of a subset Haskell to JavaScript, *JSHC*, is presented as a proof of concept. The compiler has a graphical interface in the form of an Emacs-like text editor extended to contain an interpreter terminal, to allow execution of arbitrary expressions. To allow for client-side compilation, the compiler is written entirely in JavaScript.

A comparison between *JSHC* and another JavaScript based Haskell implementation is presented, as well as some unexpected issues with the Haskell specification.

Sammanfattning

Utveckling av stora webbaserade applikationer med hjälp av skriptspråk såsom PHP och JavaScript kan vara svårt i och med avsaknaden av statisk typkontroll och abstraktion genom modulsystem i många sådana språk. Vi argumenterar för att statisk typkontroll och modulsystem kan hjälpa programmerare att undvika vanliga fel vid utveckling av stora webapplikationer. Literaturen pekar på att ett extra lager med abstraktion ovanpå skriptspråken, som erbjuder dessa möjligheter kan vara en lösning på problemet.

Vi föreslår en lösning där det statiskt typade funktionella språket Haskell kompileras till JavaScript-kod. Detta gör det möjligt att skapa webapplikationer i en modulär och typsäker miljö.

En implementation av en kompilator med stöd för kompilering av en delmängd av Haskell till JavaScript, *JSHC*, presenteras som prototyplösning på problemet. Kompilatorn har ett grafiskt gränssnitt i form av en Emacs-lik texteditor som utökats med en kommandotolk, för att möjliggöra exekvering av godtyckliga uttryck. För att kompileringen ska kunna köras på klientsidan är kompilatorn skriven helt i JavaScript.

En jämförelse mellan *JSHC* och en annan JavaScript-baserad Haskellimplementation presenteras, tillsammans med en redogörelse för vissa oväntade komplikationer rörande Haskellspecifikationen.

Contents

1	Introduction	1
1.1	Related work	1
1.2	Purpose	1
1.3	Delimitations	2
2	Theoretical background	3
2.1	Introduction to Haskell	3
2.1.1	Functions	3
2.1.2	Types	4
2.1.2.1	Type declarations	4
2.1.3	Pattern matching	4
2.1.4	Expressions	5
2.1.5	Module system	5
2.2	Introduction to Compilation	5
2.2.1	Lexical analysis	5
2.2.2	Parsing	6
2.2.3	Semantic analysis	6
2.2.4	Intermediate representation	6
2.2.5	Back-end	6
2.3	Compiling Haskell	7
2.3.1	Advanced features of the type system	7
2.3.1.1	Quantification of types	7
2.3.1.2	Kinds	7
2.3.1.3	Instantiation of types	7
2.3.1.4	Type inference	7
2.3.2	Dependency groups	8
3	Method	10
3.1	PEG.js	10
3.2	Jison	11
3.3	Ymacs	11
4	Result	12
4.1	Supported language features	12
4.2	User Interface	14
4.3	Implementation	15
4.3.1	User Interface	16
4.3.1.1	The Haskell mode	16
4.3.1.2	The Interpreter terminal	16
4.3.1.3	Setting modes automatically	16
4.3.1.4	Interpreter	16
4.3.2	Loading modules	16
4.3.2.1	Lexical analysis	16
4.3.2.2	Parsing	17

4.3.3	Checking module groups	17
4.3.3.1	Finding and traversing dependency groups	18
4.3.3.2	Export check	18
4.3.3.3	Name check	18
4.3.3.4	Fixity Resolution	19
4.3.3.5	Type checking	19
4.3.4	Translation to the intermediate representation	20
4.3.5	Compiler back-end	21
4.3.5.1	Internal libraries	21
4.3.5.2	Code generation	21
4.3.6	Standard libraries	23
5	Discussion	25
5.1	Implementation details	25
5.1.1	Export checking	25
5.1.2	Dependency checking	25
5.2	Comparison with other implementations	25
5.2.1	Other possible solutions to client-side Haskell	25
5.2.2	HIJi (Haskell in JavaScript)	26
5.3	Use of JSHC for web programming	26
5.4	Future work on JSHC	26
5.4.1	Additional support of the Haskell specification	26
5.4.1.1	FFI: Foreign Function Interface	26
5.4.1.2	Type classes	27
5.4.1.3	Input / output	27
5.4.2	Possible additions to the compiler	27
5.4.2.1	Loading files asynchronously	27
5.4.2.2	Pretty printer	27
5.4.2.3	Coloring Haskell code	27
5.4.2.4	Concurrency	27
5.4.2.5	Code optimisation	27
5.5	Issues	28
5.5.1	Project size	28
5.5.2	The Haskell grammar	28
5.5.3	Interpreter: accessing URLs	28
	Bibliography	29
A	Syntax reference	31
A.1	Context-free syntax	31
A.2	Intermediate representation	34
B	Declaration of contributions	36

Chapter 1

Introduction

Modern web applications are becoming more and more complex, with users expecting more and more automation of menial tasks such as spelling correction and auto completion of text. This leads to web applications becoming larger and more complex, which in turn means that they are becoming more error prone. One large source of errors in web applications is the absence of type safety in the form of static type checking in most scripting languages aimed at web development, such as JavaScript or PHP [19, 30, 25]. Another problem with the growing size of web applications is the lack of abstraction in the form of a module system [25]. One solution to this problem is providing an extra layer of abstraction on top of these scripts, that in turn supports a module system, and can be subject to static type checking. For example another, already existing, programming language that has these features can be used [19, 28, 24].

Haskell[23] is one such language, with well developed type- and module systems. By compiling Haskell directly to JavaScript, we use its powerful type system to guard against some of the errors that are hard to avoid when writing large and complex applications with a dynamically typed language such as JavaScript, while still leveraging the wide range of compatible runtime environments for JavaScript code.

1.1 Related work

Using Haskell for type safe web-programming is not a new concept. Several different approaches have been suggested; the use of Haskell to model HTML[28] and using server-side Haskell applications to generate CGI[24] are two examples. Other languages similar to Haskell have also been suggested for type safe web-applications, such as the functional logic language Curry[19].

There are several implementations of compilers from Haskell to JavaScript; although none of them has reached a full release stage as of the time of writing. There are three implementations of JavaScript back-ends for existing Haskell compilers; the Utrecht Haskell Compiler has one under development [12] which is planned for inclusion in an upcoming release, the now defunct [26] York Haskell Compiler had one, coupled with a toolkit for web content creation [18], and there is one in development as a back-end for the Glasgow Haskell Compiler which is in its alpha stage at the time of writing [1].

There are also at least one other attempt to implement Haskell directly in JavaScript, called Haskell in JavaScript, which is an online interpreter [10].

1.2 Purpose

The purpose of this project is to implement a Haskell compiler that generates JavaScript code, JSHC (JavaScript Haskell Compiler). The compiler is to be written entirely in JavaScript,

to ensure compatibility with most major modern browsers, and relieve users of the burden of having to download and install a compiler to their system. A JavaScript based graphical user interface with a text-editor and a code interpreter will also be implemented by modifying an existing JavaScript based text editor. The code interpreter will enable users to run their programs, and execute Haskell expressions directly in the editor, in the vein of the Glasgow Haskell Compiler's GHCi, or the Hugs interpreter. The compiler is to serve as a proof of concept for developing web applications in Haskell in an online environment.

1.3 Delimitations

Due to the size and complexity of the Haskell programming language, the JSHC implementation is limited to a small subset of Haskell containing the features described as beneficial to web programming above, i.e. static type checking, module system and lazy evaluation.

This differs from the initial intention with the project, which was to implement the full Haskell 2010 specification[23]. The reasons for this delimitation is discussed more in [5.5.1](#).

Chapter 2

Theoretical background

2.1 Introduction to Haskell

This section contains a quick introduction to Haskell for readers unfamiliar with or new to Haskell and/or functional languages. It consists of a very general overview, followed by separate sub chapters with details about specific features.

Haskell is a **lazy, purely functional** programming language with **strong static typing** and **type inference**. A program in Haskell consists of one or more modules, interconnected by their import and export specifications. The contents of a module is a set of top-level declarations, which can be, among others, function declarations, datatype declarations and typeclass declarations. Functions consist of a name, a set of arguments and an expression, that expresses the result of the function when applied to its arguments.

Being lazy means that an expression is only evaluated when its result is needed. Any expression whose result is not used is simply ignored. This enables lazy languages to feature some things that are otherwise impossible, such as infinite data structures. Since the data structure is lazy, only the part that is needed is calculated. For example, a list of infinite length can be described in Haskell like so:

```
x = [1..]
```

This declaration assigns to the variable `x` an expression that calculates an infinite list containing all integers > 0 in rising order. While an expression calculating this in a strict language would never terminate as the set of integers > 0 is infinite, a lazy language only calculates the needed parts, e.g:

```
take 5 x
```

where `take` is a function that returns a list consisting of the first `n` elements in a list, only needs the first 5 elements of `x`, so no more than 5 elements are computed.

2.1.1 Functions

Being a purely functional language, Haskell's functions do not have side effects, i.e. they only compute values given their arguments. There is no update of variables.

Another feature of purely functional languages is that functions are first-class objects. This means that functions can be passed around as arguments to other functions, be returned as the result of a function or created at runtime. This enables the creation of **higher order functions**, generalized functions that use other functions as arguments. For example, the function `filter` from the Haskell standard libraries takes a function, which takes one argument and returns a boolean, and a list. The function then applies the supplied function to each element in the list and returns a new list with all elements for which the supplied function returned `True`.

A Haskell function can either be named, meaning it is declared with a name somewhere in the program, either as a top-level definition or as a local function inside a `let` or `where` expression, or anonymous as lambda expressions. Function declarations take the form `name arg1 ... argn = expression`, where `name` is any identifier starting with a lower case letter, `arg1` through `n` is a series of zero or more arguments and `expression` is the expression that denotes the result of the function. It is possible to define functions as infix operators in the same way as ordinary functions, complete with custom precedence levels.

It is also possible to create anonymous functions, which are not bound to any declarations through **lambda expressions**. Lambda expressions have the form `\arg1 ... argn -> expression` where `arg1` through `n` is a series of arguments and `expression` is the expression that denotes the result of the function.

2.1.2 Types

All values in Haskell have a type. The type `Int -> Int` represents a function which takes a value of type `Int` and produces a result of type `Int`.

It is possible to write a function that takes values that may be of more than one type. To achieve that, one uses **type variables**, which are written with lower-case letters.

A function `id :: a -> a` (where `::` is read as "has type") is therefore a function that takes a value of any type and produces a result of the same type.

This is called a **polymorphic type**, as `id` can have a different type depending on how it is used, as the type variable `a` can be replaced with different types.

Types can have **type parameters**, which is necessary for collections if one wants to be able to use them where all values have the same type, but be able to choose that type.

A list of integers is written `[Int]`, and a function that computes the length of a list as an `Int` would have type `[a] -> Int` as it would work regardless of the type of the values in the list. Note however that all the values must have the same type.

The list type is built-in. More generally, a type can be applied to any number of types as in `Array Int Double`, which is an `Array` with indices of type `Int` and stores values of type `Double`.

2.1.2.1 Type declarations

New types can be created.

`data Tree a = Leaf | Node a (Tree a) (Tree a)` declares a **type constructor** `Tree` which can be applied to any type that one wants the values in the tree to have. It also declares the **data constructor** `Leaf` which represents an empty tree, and the data constructor `Node` which is used to construct a new tree given a value to store in the root node of the tree, and the left and right sub-tree.

The built-in lists have the special type constructor `[]`, and the data constructors `[] :: ∀a. [a]` (the empty list) and `(:) :: ∀a. a -> [a] -> [a]` (creating a new list given a first element and the rest of the elements).

2.1.3 Pattern matching

Pattern matching is a conditional construct that matches an expression `e` to a set of pre-defined patterns paired with expressions, `rhs`, in a fixed order. Matching can be done on data constructors of any arity, primitive values such as integers, and variables. Matching an expression to a variable will always succeed and, in addition, bind the value matched to the variable for use in the `rhs`. The first pattern to match `e` has its corresponding `rhs` evaluated with any variable bindings defined in the pattern. Pattern matching is used in several places in Haskell, most notably function definitions, where they allow specifying different function behaviour depending on the arguments given (dividing the function into **clauses**), and case expressions.

2.1.4 Expressions

Expressions in Haskell include `let` and `where` expressions which allow for local declarations inside expressions and functions, respectively.

Conditional expressions of the if-then-else form are supported, as well as pattern matching inside expressions using case expressions.

2.1.5 Module system

Haskell's module system is its main method of encapsulation. Apart from local variables in a certain function, it is the only way to determine whether something is known (i.e. in scope) or not. Each module has a name, a set of imports, a set of exports and a body containing top-level declarations. The specification of these is optional, with defaults being the module name "Main", the empty import set and the export set containing all top-level declarations.

By specifying exports, nothing defined in the module that is not in the export space can be accessed by an importing module. It is also allowed to specify the export of imported modules or their respective exports. This way, if module A imports module B and module C imports module A, it is possible for module C to also have imported some definitions defined in module B (or even module B's entire export set), depending on what module A's export set is.

It is also possible for a module to specify what it will import from another module, either as an inclusive specification or as an exclusive specification (using the `hiding` keyword).

One important feature of Haskell's module system is that unlike, for example, Java, it is permitted for a group of modules to import each other.

2.2 Introduction to Compilation

This chapter is intended as an introduction to the general concepts of compilation for readers who have little or no experience with the inner workings of compilers. The approach to compilation described here is not the only one, and only describes the compilation steps used in JSHC. For a more general and in-depth view of compilers and compilation, [8] is recommended.

A compiler is a program that, given program code, produces new code in another language, the **target language**. Common examples of compilers' target languages are machine code in the case of the GNU C Compiler, bytecode to be run in a virtual machine in the case of Java or another high level programming language in the case of the Glasgow Haskell Compiler, which can compile to C.

A compiler consists of two main parts; the **front-end** and the **back-end**. The front-end takes the program code and creates an abstract representation of it, and the back-end takes that representation and converts it into the target language.

2.2.1 Lexical analysis

Lexical analysis is the first part of the front-end, and is the process by which the compiler takes the program code in the form of an input string, and parses it into a list of separate tokens, **lexemes**, which are the smallest significant components in a programming language [8], chapter 1.2.

For example, given a program in a simple language with Haskell-like syntax with a source code that looks like this:

```
fun x y = x + y
```

a lexer could generate a list of tokens in this vein:

```
[fun,x,y,=,x,+,y]
```

2.2.2 Parsing

Parsing is the second part of the front-end and takes the list of lexemes and interprets it according to a grammar, creating a data structure containing an abstract representation of the code, called the **abstract syntax tree**. The abstract syntax tree is a tree (or tree-like) graph with nodes representing the application of a certain rule in the grammar.

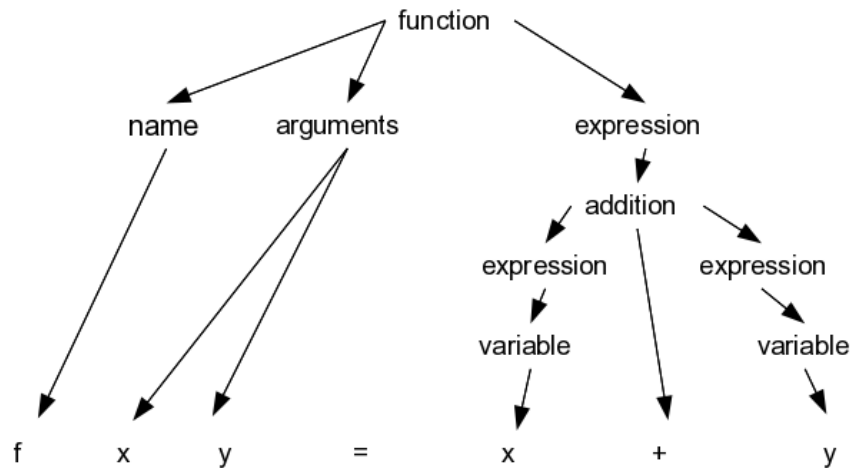


Figure 2.1: Graphic representation of an abstract syntax tree of a function definition in a simple language with Haskell-like syntax.

2.2.3 Semantic analysis

Semantic analysis gathers various semantic properties of the source program, to check for errors and to use later on in the compilation process. One common part of semantic analysis is **type checking**, where the compiler searches the abstract syntax tree for type inconsistencies. For example, in many languages, adding a boolean to an integer is not a valid expression, an error which would be caught by the type checker. Another common part of the semantic analysis is **name checking**, where the compiler checks whether there are any ambiguities in variable names used, and whether used variables are in scope where they are used.

Other operations on the syntax tree, such as reduction of complex constructs into simpler, more general constructs can also be done as part of the semantic analysis. This is done to simplify the code generation process. Some code optimizations that are independent of the target language used can also be performed at this stage.

2.2.4 Intermediate representation

After passing through the compiler front-end, the source program has been converted into an **intermediate representation**, which will later be used as input for the back-end. The intermediate representation can take many forms, for example an abstract syntax tree, code in another high-level programming language, or some low-level code (for example three-address code)[8], chapter 1.2.

2.2.5 Back-end

The back-end is the final part of a compiler. The main function of the back-end is code generation, which is the process of taking the intermediate representation and producing a semantically equivalent program in the target language[8], chapter 8. The back-end

can also fill other functions such as performing code optimizations that are specific to the target language.

2.3 Compiling Haskell

In this section, we present a more detailed introduction to concepts that are required to understand the various steps in the compilation of Haskell.

2.3.1 Advanced features of the type system

2.3.1.1 Quantification of types

Type variables, just as ordinary variables must be declared and used variables must be in scope of a declaration.

The declaration of variables is called **quantification**. A variable that is not declared is **free**, and a variable that is declared is **bound** (by the quantification).

When writing a type signature in Haskell, the quantification is implicit.

This means that a type signature `[a] -> [a]` is actually $\forall a. [a] \rightarrow [a]$, where \forall binds the type variable `a`.

The implicit quantification of type signatures binds all type variables in the signature.

All type signatures written in a program and inferred by the compiler are **rank-1 polymorphic types**, which means that the type given to a value begins with a quantification of all the type variables that occur in it, followed by the rest of the type which does not contain any quantifications. This makes the type system simpler.

2.3.1.2 Kinds

Given the datatype declared in 2.1.2.1, the function type `Tree -> Tree` is invalid as the type `Tree` has a type parameter.

To detect these errors, all types and type constructors have a **kind**.

The kind of all types is `*`.

A function type is valid if both the argument and result type have kind `*`.

The kind of the type constructors `Tree` and `[]` is `* -> *`, which means that they take a type and produce a type. They must therefore be applied to a type which is not missing any arguments (i.e have kind `*`), and will result in a type of kind `*`.

2.3.1.3 Instantiation of types

When using a data constructor such as `[] :: $\forall a. [a]$` , one would want to use it with lists with different element types in different locations.

This is achieved by replacing all bound type variables in the type with new type variables wherever it is used.

These type variables are now free, but will be bound after the type of the expression has been inferred unless they are eliminated by the **type inference**.

2.3.1.4 Type inference

If one makes a declaration `y = x : []` where `x :: Int`, one will have:

- `(:)` :: "a -> [a] -> [a]"
- `[]` :: [b]
- `x` :: Int

where a and b are free type variables.

To replace the type variables with types, one uses type inference which creates type constraint based upon how the types are used.

Since x is the 1st argument to $(:)$, the constraint $a = \text{Int}$ is created.

Since $[]$ is the 2nd argument to $(:)$, the constraint $[b] = [a]$ is created, which gives the constraint $a = b$.

The type inference algorithm must assign a single type to each of the free type variables such that all the constraints are satisfied.

If there is more than one possible assignment, then the type is called ambiguous, and is an error, unless a default [23], chapter 4.3.4, type can be chosen.

If there is no possible assignment, then an error will arise from the constraints.

In this case, there is only a single possible assignment, and the inferred types are:

- $(:) :: \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]$
- $[] :: [\text{Int}]$
- $y :: [\text{Int}]$

For more details see Types and Programming Languages [29], chapter 22.

2.3.2 Dependency groups

Since modules can mutually depend on each other, there can be a cycle in the dependencies between modules.

If a module A and a module B depends on each other by importing functions from the other module, one can not analyze one before the other, as the imported functions will be missing.

To solve this, one must analyze A and B together.

More generally, one creates a dependency group for any set of modules that creates a dependency cycle such as A depends on B , B depends on C , and C depends on A .

The dependency groups are the strongly connected components in the directed graph of dependencies between single modules.

After they have been computed, the resulting graph is a directed acyclic graph (DAG) where nodes are dependency groups, and the edges are the dependencies between the groups.

As there are no cycles in the dependencies between the groups, it is possible to traverse the groups in dependency order (a reverse topological ordering).

The Haskell standard [23] mentions dependency groups in Chapters 4.5.1, 4.5.2, and 4.6, related to different topics.

This concept is used in several places of the compiler. It is needed to describe sets of modules that depend on each other, sets of top-level declarations that depend on each other (even between different modules), and declarations in `let/where` expressions.

An example of dependencies between local definitions is:

```
f x = let
  even 0 = True
  even n = odd (n-1)
  odd 0 = False
  odd n = even (n-1)
in odd x
```

there is a cyclic dependency between the declaration of `even` the declaration of `odd`. The same example can be applied to the case where `even` and `odd` are top-level declarations. Since they mutually depend on each other, type checking needs to be performed on both at the same time.

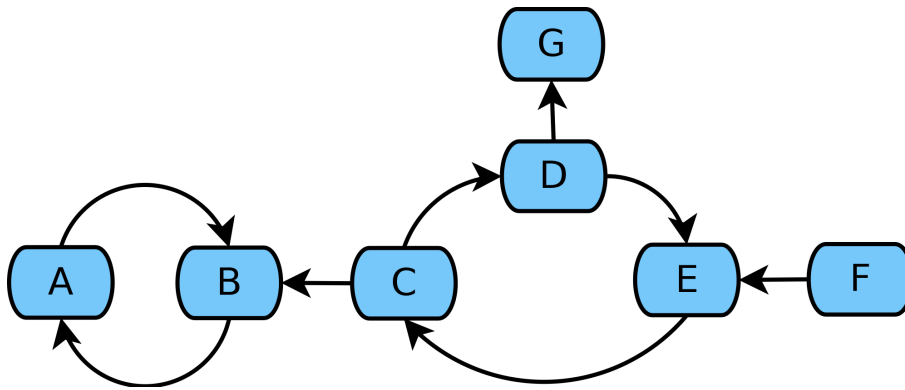


Figure 2.2: Example module dependency graph. The arrows mean *imports from*. In this example, the dependency groups are {A,B}, {C,D,E}, {G} and {F}. On compilation, {A,B} and {G} need to be checked first, then {C,D,E} and {F} last.

Chapter 3

Method

The main part of this project was the development of the compiler itself. The first stage of the project was to identify and isolate subgoals and -problems that could be reasoned about on their own. Before proceeding with any actual implementation, we attempted to identify a theoretical solution to each of these subgoals and -problems, to facilitate subsequent implementation. Examples of such subproblems are type inference and dependency resolution. The next stage was to identify larger components of the project that could largely be implemented independently. Five such components were identified; lexical analyser, parser, semantic analyser, code generator and graphical interface. Responsibility for these parts was then split between group members.

Implementation work was largely carried out individually, with regular meetings to synchronize efforts and plan ahead. To keep work synchronized between meetings, the git[2] version control system was used.

To allow for compatibility with as many web browsers and JavaScript engines as possible, all code in JSHC is compliant with the ECMAScript standard[15].

Early on in the project phase, it was determined that several external tools would be needed to facilitate the development of JSHC.

3.1 PEG.js

PEG.js[22] is a JavaScript implementation of parsing expression grammars, which is an alternative to context-free grammars and regular expressions for defining formal languages[17]. The definition of a parsing expression grammar is similar in appearance to that of a context-free grammar, with a few key differences.

The **unordered choice** operator `|` is replaced by an **ordered choice** operator `/`. This means that, instead of accepting a match on any rule in a production, a parsing expression grammar always accepts the first rule matched. This means that parsing expression grammars are inherently unambiguous. If there are two possible matches to a rule, simply match the one that occurred first.

Parsing expression grammars also have several operators similar to those used in regular expressions. There are the repetition operators `*` which matches zero or more repetitions of a rule, and `+` which matches one or more repetitions of a rule. There is also the `?` operator which either matches a rule and returns it, or does not match it and returns an empty match. Just as with the ordered choice, these operators take a greedy approach, i.e. they always consume the maximum amount of input possible.

Lastly, parsing expression grammars feature the syntactic predicates `&` and `!`. These match their paired rule, and either fail or succeed without advancing the parser position. `&` returns an empty match if the match is successful and fails otherwise, while `!` returns an empty match if the match fails and succeeds otherwise.

PEG.js generates a Packrat[16] parser, i.e. a top-down recursive descent parser with backtracking. To avoid a worst-case scenario of exponential runtime, Packrat parsers store intermediate results in a table, sacrificing storage space for guaranteed linear-time complexity.

PEG.js was chosen for this project due to the suitability of parsing expression grammars for writing lexical analysers [17] in a concise and intuitive way, as well as for the efficiency of Packrat parsers.

3.2 Jison

Jison[11] is a parser generator for generating JavaScript parsers. It takes a context-free grammar in the same format as the input files to Bison, apart for parser actions, which are defined in the implementation language, which is JavaScript for Jison and C for Bison. Just like Bison, the parser generated by Jison is LALR(1) by default[11, 13], which is the option used for the parser in JSHC.

Jison was chosen mainly because of the familiarity and extensive documentation of the Bison notation and functionality, as well as for its powerful constructs for handling user-defined error handling, something which is essential to a Haskell parser [23], chapter 10.3. The speed and compact size of LALR parsers were also taken into account, but not critical for the choice.

3.3 Ymacs

As a graphical interface to JSHC, a JavaScript-based Emacs-like text editor named Ymacs[9] was chosen. The main reasons for this choice were that Ymacs is easily extendable and is fully compatible with three of the most widely used web browsers (Mozilla Firefox, Google Chrome and Apple Safari[7]). Also, Emacs is a popular coding environments among Haskell programmers[32].

The Ymacs interface design centers around the two central concepts of **buffers** and **modes**. The word buffer refers to two different entities in the Ymacs program; the data structure which stores entered text coupled with the functions managing its manipulation, and the text field which is presented a user. These two denotations will be used interchangeably, as it should be clear from context which denotation is intended. While the default Ymacs view contains a single buffer, buffers can be split, by creating a new buffer sharing the same screen space. Each buffer has a mode, which determines the behaviour of the buffer. Modes determine what hot key commands are available, how text is displayed etc. Due to the transparent nature of JavaScript programs (i.e. all code inside a JavaScript application shares the same global namespace), modes can be used to completely redefine the behavior of a buffer without changing any of the Ymacs source code.

Chapter 4

Result

The result of this project is a Haskell compiler written in JavaScript, producing code in JavaScript. An interface with a text editor and interpreter terminal has been implemented by integrating an existing editor with the compiler.

The project web page is <http://jshc.insella.se/>, where one can try out the JSHC interface and also download source code (which is distributed under the permissive MIT license[5]). The source code contains a copy of Ymacs which is distributed under the permissive BSD-3 license[4].

4.1 Supported language features

This section describes informally the subset of Haskell features implemented in JSHC. Figure 4.1 contains a simplified syntax reference for the subset of Haskell supported. Note that some accuracy has been sacrificed in favour of simplicity in that figure. For brevity, module system syntax has been omitted. For a full formal syntax reference, see Appendix A.1.

JSHC supports a subset of the Haskell module system, with exception to qualifying and renaming imports. It also lacks support for many cases of mutually recursive modules.

Polymorphic type inference, without overloading through type classes, is supported, as well as lazy evaluation of expressions.

Top level declarations supported are non-strict abstract datatypes, fixity declarations, top-level function declarations (including user-defined operators) and type signatures. Local declarations are supported in `let` and `where` expressions.

Expressions supported are lambda abstractions, conditional expressions, case expressions, tuples, lists, integer literals and function, operator and data constructor applications.

Pattern matching in case-expressions and function definitions, using patterns for data constructors, tuples, integer literals, variable bindings and wildcards is supported. Patterns can be arbitrarily nested.

```

top-decl → data <con> <var>* [ = <constructor-decl>+ (separator: "|") ]
        | <declaration>
constructor-decl → <con> <type>+
type → <con> | <var> | <type> -> <type> | ( <type> )
declaration → <function> | <type-signature> | <fixity-decl>

type-signature → <var>+ (separator: ",") :: <type>
fixity-decl → <fixity> <integer> <op>+ (separator: ",")
fixity → infixl | infixr | infix
function → <fun-lhs> = <exp> [where { <declaration>+ (separator: ";") } ]
fun-lhs → <var> <pattern>* | <pattern> <op> <pattern>

exp → \ <pattern>+ -> <exp>
    | let { <declaration>+ (separator: ";") } in <exp>
    | if <exp> [;] then <exp> [;] else <exp>
    | case <exp> of { <alternative>+ (separator: ";") }
    | <exp>+
    | <integer>
    | [ <exp>+ (separator: ",") ]
    | ( <exp>+ (separator: ",") )
    | <exp> <op> <exp>
    | <var>
    | <con>

alternative → <pattern> -> <exp>
            [where { <declaration>+ (separator: ";") } ]
pattern → <con> <pattern>*
        | <integer>
        | _
        | <var>
        | ( <pattern>+ (separator: ",") )

var → <qvarid> | ( <op> )
op → <qvarsym> | `<var>`
con → <qconid>

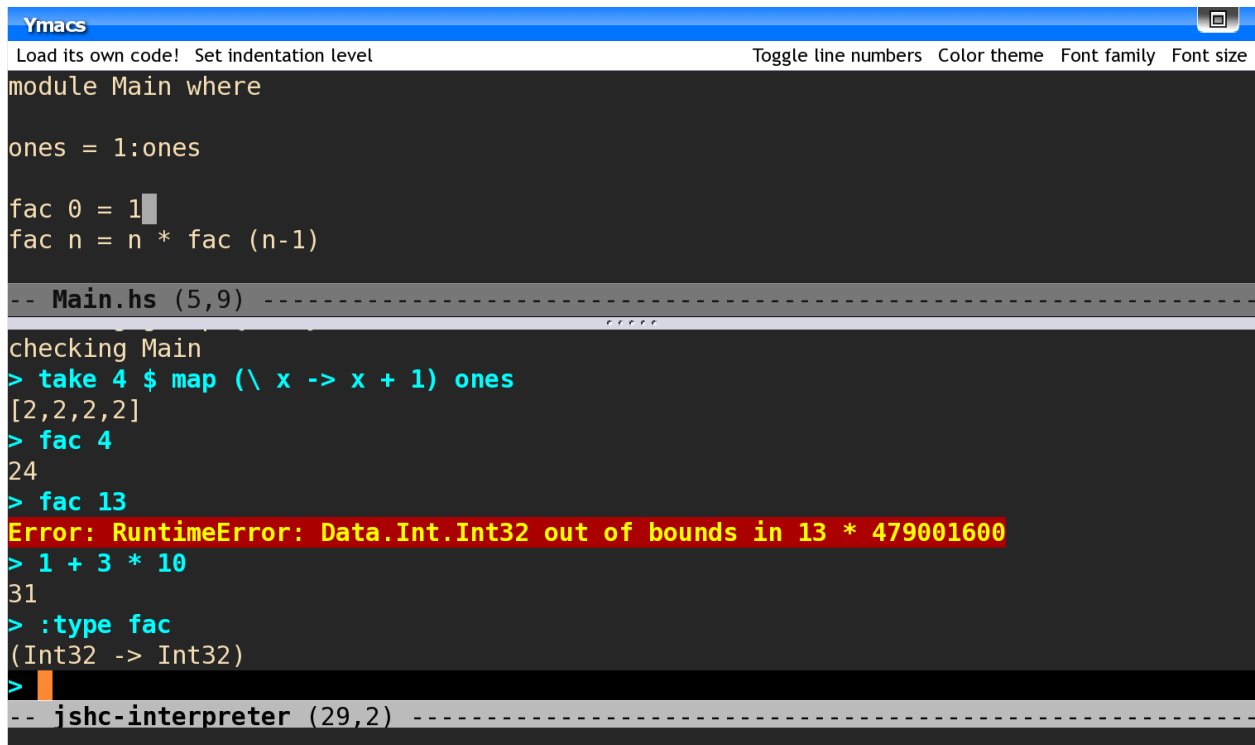
```

Figure 4.1: Simplified syntax reference for the subset of Haskell supported by JSHC. Arrows denote the start of a rule, | denotes alternative productions of a rule. Items surrounded by angular brackets are references to a rule in the grammar and items contained in square brackets are optional. A * after an item represents a list of zero or more repetitions of the rule, while a + represents one or more repetitions. The separator notation after some lists of items indicates that the items in the specified list need to be separated by the given separator token. The nonterminals <qvarid>, <qvarsym> and <qconid> represent the tokens with the same name in the Haskell lexical syntax ([23], 10.3). <integer> represents decimal integer literals. All other items represent strings.

4.2 User Interface

The interface is a modified Ymacs environment.

To modify the behaviour of Ymacs to make it suitable as a development environment for JSHC, several modifications has been done to the default environment.



```
Ymacs
Load its own code! Set indentation level Toggle line numbers Color theme Font family Font size
module Main where

ones = 1:ones

fac 0 = 1
fac n = n * fac (n-1)

-- Main.hs (5,9) -----

checking Main
> take 4 $ map (\ x -> x + 1) ones
[2,2,2,2]
> fac 4
24
> fac 13
Error: RuntimeError: Data.Int.Int32 out of bounds in 13 * 479001600
> 1 + 3 * 10
31
> :type fac
(Int32 -> Int32)
>

-- jshc-interpreter (29,2) -----
```

Figure 4.2: Screen capture of the Ymacs interface and JSHC interpreter as viewed in a web browser.

An interface for using the compiler has been created. It uses Ymacs to allow a user to write modules and then load them in an interpreter that uses the compiler.

The interface provides two initial buffers: a buffer named `keybindings.txt` containing the general keybindings which are available regardless of mode, and a buffer called `Main.hs`, which is empty. The `Main.hs` buffer, and any new buffers that has a name ending with `.hs` will have the command `C-x i` available to show the interpreter terminal in a new frame by splitting the current frame (unless already visible, in which case the focus is just changed).

There will also be an attempt to load the Haskell file with the same name, and if found in any of the paths provided, the file contents will be written into the buffer.

The interpreter terminal supports a subset of the GHCi commands (table 4.1), allows evaluation of Haskell expressions, and also includes a command for evaluation of JavaScript expressions.

It only allows insertion/deletion of a character/selection on the user input on the last line.

There is auto-completion of commands using `TAB`.

command	description
:help	Prints available commands
:kind	Takes any number of type constructors and prints the kind of each
:load	Takes a number of module names of modules to load. Will find all dependencies and recompile modules as necessary.
:show	Prints the possible arguments to :show
:type	Takes a Haskell expression which will be compiled. If it has no errors, the type of the expression will be printed.
:js	Takes a JavaScript expression and evaluates it. If it has no errors, the result will be printed.
:show path	Prints the URLs which will be used when loading Haskell modules. The default paths are <code>hslib/</code> and <code>hsusr/</code> from the root of the source distribution.
:show code	Prints the generated JavaScript code of all loaded Haskell modules.
:show modules	Prints the names of the currently loaded modules.

Table 4.1: Interpreter commands

4.3 Implementation

Figure 4.3 shows how the interface (see 4.3.1) uses the compiler.

Given a set of modules to load (the **target** set), the compiler will load (see 4.3.2) them resulting in the **loaded** set, which is set of all required modules unless there was an error.

The compiler will then compute the dependency groups of the loaded set, and traverse the groups in dependency order (see 4.3.3).

The compiler will then for each module, translate the abstract syntax tree (see 4.3.4) to the intermediate representation, and then generate (see 4.3.5) JavaScript code.

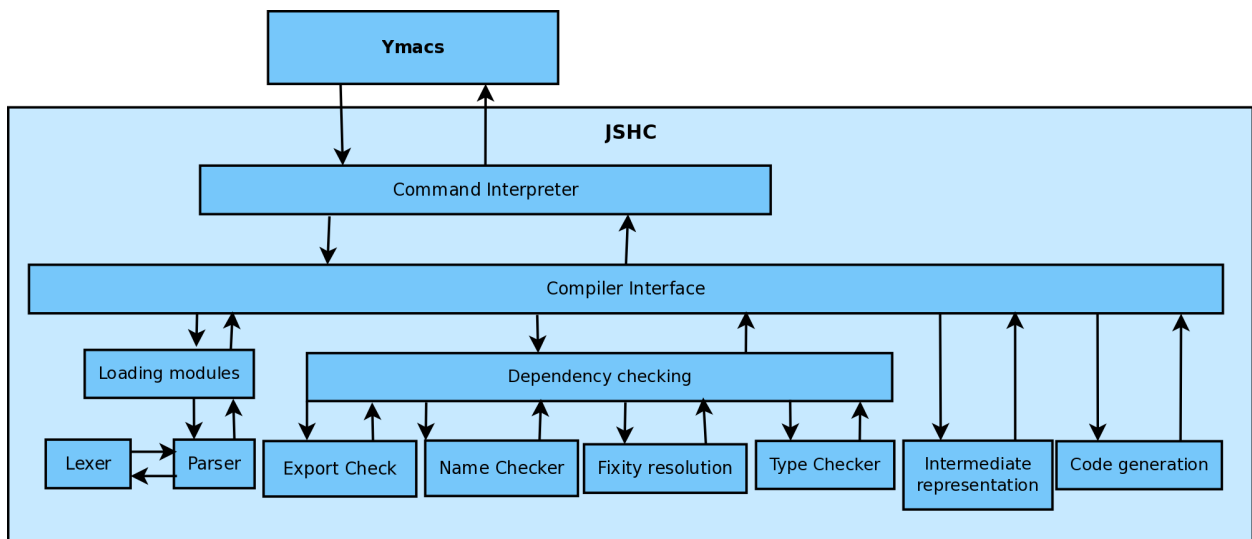


Figure 4.3: Flow of information in the implementation

4.3.1 User Interface

Commands in Ymacs are added using modes. A mode contains a key map to be checked before the default key map, and a Ymacs tokenizer that handles coloring.

4.3.1.1 The Haskell mode

A Haskell mode has been created for buffers containing Haskell code.

The command `C-x i` calls a global function that searches for the interpreter buffer. There can only be a single interpreter buffer, so a new interpreter buffer is only created if one does not already exist.

4.3.1.2 The Interpreter terminal

The terminal is an ordinary buffer modified in several ways, together with a mode that traps input from the `ENTER` and `TAB` keys and assigns custom behaviour to them.

Functions to insert and delete text are wrapped into new functions that will be called instead, to prevent inserting text anywhere in the buffer.

The buffer contains a reference to the interpreter object, and can therefore call the interpreter methods to execute commands. The buffer also adds a callback to the interpreter so that error messages from the commands are not return values from the interpreter, but instead calls to functions that add output to the buffer. This allows output to be written to the buffer while the interpreter is working, and not all at once at the end.

4.3.1.3 Setting modes automatically

Since no mode is automatically set for new buffers, the method that handles creation of new Ymacs buffers has been wrapped in a new method that sets a mode for the buffer based upon what the extension of the name of the buffer is. The implementation is similar for opening an existing module.

4.3.1.4 Interpreter

The interpreter contains the implementation of all commands.

It contains an instance of the compiler which it uses to load code, look up names, and compile and run expressions.

Compiler output is sent to the interpreter via callbacks, so the compiler does not know about the interpreter.

4.3.2 Loading modules

The loading begins with the module names of a set of modules to load, and will search for them in a virtual file system (mapping from module names to unparsed strings), a list of URLs (which can refer to files using the `file://` or `http://` protocols). Each found module must be parsed in order to find out which modules it depends upon. Those modules (unless already loaded) must then be found and parsed as well. This process continues until no more modules are needed, and the minimal set of modules that satisfies all the module dependencies has been found. In case a module can not be found or there is a parse error, this process will still continue and try to satisfy more dependencies and see if there are more missing modules or parse errors.

4.3.2.1 Lexical analysis

The lexical analysis in JSHC is performed in several steps.

While lexical analysis is often done by having the parser request a token from the lexical analyzer, match that token against a rule, request another and so on[8], chapter 3.1, the

layout-dependent syntax of Haskell requires a different approach. First, the input (i.e. the Haskell source code, including whitespace and comments) is parsed into an array of tokens in one go. This is done by using a parser built using PEG.js that was fed a translation (into PEG.js-readable form) of the lexical program structure described in [23], chapter 2.2. Second, the generated array of tokens is run through a function that adds indentation information to the array in the form of special indentation tokens, and annotates tokens with position information. Whitespace and comments are discarded after being used to calculate position information, as they are no longer needed. Lastly, this array is fed into an object which interfaces with the parser according to the Jison/Bison style API[8], chapter 3.1, using the indentation information to generate tokens to convert Haskell's layout-dependent syntax into a context-free one.

As an example, the code:

```
f = let
  x = 1
  y = 2
  in x + y
```

generates, after the first and second stages this array of tokens (token representation rewritten for readability):

```
[{1}, ID, '=', 'let', {3}, ID, '=', INT, <3>, ID, '=', INT, <3>, 'in', ID, OP, ID]
```

where the tokens {*n*} designate the start of a block after a newline and <*n*> representing a newline where a new block should not begin. *n* represents the indentation of the starting column of the first token on the new line. Given this array as input, the last stage of the lexer will deliver to the parser tokens representing a context-free version of the code which, converted back into Haskell source code, would look like this:

```
{f = let {x = 1;y = 2} in x + y}
```

4.3.2.2 Parsing

The context-free syntax is parsed by a parser implemented using the JISON parser generator [11]. The parser and the last step of the lexical analysis work in concert, meaning that the lexical analyzer does not process and return all tokens in one go, but only sends the parser tokens when asked to. To handle certain features of the layout dependent syntax of Haskell, that cannot be handled by the lexical analyzer alone, the parser will inform the lexical analyzer of any such parse errors, so that the lexer can react and send new tokens that solve the problem (c.f. the description of the `parse-error` function in [23], chapter 10.3).

4.3.3 Checking module groups

For each dependency group of modules checked in dependency order, the compiler does the following:

- If some modules are already compiled, they will only be compiled if not all of the modules within the same group have previously been compiled.

- For each module, compute the **tspace** (the set of declared top-level names in the module), and adding fixity information to this name so that it can easily be found later on.

- Resolve the exports (see 4.3.3.2), which results in the **espace** (the set of exported names) being known for each module in the group.

- For each module, check usage of names (see 4.3.3.3).

- For each module, resolve fixity of operators (see 4.3.3.4).

Then dependency groups are produced for the top-level declarations of all modules in the group. The groups are then type checked in dependency order.

- Export check: Computes the export space of each module
- Name check: Checks if names are in scope, and qualifies names referring to top-level declarations
- Fixity resolution: Replace all lists of infix applications with operator application
- Type check: Compute types and kinds.

4.3.3.1 Finding and traversing dependency groups

The implementation for handling dependency groups takes a set of entries, where each entry provides a set of names, depends on a set of names, and contains a set of items, such as modules or declarations.

These entries will have to be created from the declarations or modules that one will want to create dependency groups for.

The entries are nodes in a graph where the minimal number of entries are merged such that there are no longer any cycles in the graph.

4.3.3.1.1 Condensing the graph Given a directed graph where the edges represents dependencies, it merges the minimal amount of nodes to create a directed acyclic graph. This is done by finding the connected components in the graph and merging all the nodes in each connected component into a single node.

The algorithm is a depth-first search through the graph which keeps track of the current path and merges all the nodes in a cycle whenever one is found.

4.3.3.1.2 Dependency order traversal To traverse the graph in dependency order, an opposite edge is added for each edge so that each group not only knows what it depends upon, but also which groups depends upon it. Each group keeps track of the number of remaining groups that it depends on, and for each group that is checked, this number is decreased for all groups that depends on it (which are exactly the set of groups accessible via the added edges). The traversal keeps track of a ready set and waiting set of groups. Whenever the number of remaining unchecked dependencies reaches 0, the group is moved to the ready set.

Unless there are missing dependencies (or there are groups which are currently being checked), the ready set will never be empty while the waiting set is not.

A group is checked by removing it from the ready set, check it, and then inform the traversal algorithm that the group is checked.

When both sets are empty, the traversal is complete.

Concurrency is possible, since any number of groups could be taken out from the ready set to be checked at the same time.

4.3.3.2 Export check

The implementation gives an error if modules in the same group have a list of exports, as this is currently not supported. If the modules do not have a list of exports, all top-level declarations are exported, which is easy to handle, and is only what is currently supported. If there is only a single module in a group, then the export list may exist and the exported names are resolved by looking at all top-level declarations and imports.

4.3.3.3 Name check

This is applied separately on each module in a group.

Errors is given for missing and ambiguous names. Otherwise the name will be qualified with the qualification of what it refers to so that the definition can easily be found later

on when it is non-local. The definition needs to be found later on when fixity and type information is required.

4.3.3.4 Fixity Resolution

Fixity resolution is applied separately on each module in a group. It is the procedure which, given information on each operator's fixity and precedence, calculates in which order the operators in an infix expression are to be applied. This information is then used to convert the infix expressions into ordinary prefix function applications.

For example, the expression $2 + 3 * 10$ could evaluate to 32 or 50, depending on the fixity and precedence of the operators $+$ and $*$. The algorithm looks up the fixity information of the operators, and finds out that they are both left fix operators with precedences of 6 and 7 respectively. Since $*$ has the higher precedence, it should be applied before $+$, resulting in the application expression $(+) 2 ((*) 3 10)$, which evaluates to 32, as expected by anyone familiar with the operators' precedence in ordinary arithmetics.

4.3.3.5 Type checking

The monomorphism restriction ([23], chapter 4.5.5) is not implemented.

Since type classes are not implemented, the type system only implements the parametric polymorphism.

Type checking is applied on dependency groups over all declarations from all modules in the group, since one can not check the top-level declarations of a single module in dependency order as they may depend on types from top-declarations in other modules within the same group.

While type checking, the algorithm keeps track of local names and type variables, and also the constraints using a map from the type variables to types to which they are constrained.

4.3.3.5.1 The constraint map

The constraint map maps type variables (LHS) to types (RHS).

It simply represents the types that type variables are constrained to.

Type variables are only given a type if they are constrained, so all new type variables will not be in the map until a constraint is placed upon them.

Type variables on the LHS may not occur in their corresponding RHS as this would create an infinite type, and may only occur in the RHS of other variables if the other RHSs are such that there is no infinite type. This invariant is maintained when new constrained are inserted.

The reason for using a constraint map is that type errors are discovered as soon as possible, so a type error in a sub-expression will be given for the sub-expression instead of the expression as a whole if the constraints were just gathered and then checked after computing all of them, which is the usual description of solving the constraints (see chapter 22 in [29]).

4.3.3.5.2 Adding constraints

Function constraints of the form $a \rightarrow b = c \rightarrow d$ will be split into two separate constraints $a = c$ and $b = d$, which will each cause elimination of a type variable as the constraint specifies that two type variables are equal.

4.3.3.5.3 Checking a group of declared names

Each declared name in a group is given a type variable. As the declarations are checked, the type variables for the names will be constrained depending on what the declaration contains. After all declarations in the group have been checked, all free type variables occurring in the types bound to the type variables of each declared name are quantified since they only occur inside the declarations.

4.3.3.5.4 Checking declarations It is necessary to use dependency groups and check in dependency group order to give the correct type, instead of checking all of the declarations as a single group, because of the monomorphism restriction. In a declaration such as $f = (+)$, f is supposed to be non-polymorphic, which means that the polymorphic type must be simplified before checking other declarations that depend upon it.

However, the monomorphism restriction is currently not implemented.

4.3.4 Translation to the intermediate representation

This step translates the abstract syntax tree to the intermediate representation used by the back-end, see Appendix A.2 for details. Several constructs are reduced to other, simpler and more general, constructs to create a kernel, which is a small subset of Haskell. The constructs that can be reduced in this way can be seen as syntactic sugar of the kernel[23], chapter 1.2.

Conditional expressions are simplified by converting

$\text{if } e1 \text{ then } e2 \text{ else } e3$

to a case expression of the form

$\text{case } e1 \text{ of } \{\text{True} \rightarrow e2; \text{False} \rightarrow e3\}$.

let and where expressions both bind expressions to local variables, something which is possible to express with case expressions. Thus, expressions of the form

$\text{let } decls \text{ in } exp$ and $exp \text{ where } decls$

where $decls$ is a set of declarations of the form

$p_1 = e_1 \dots p_n = e_n$

are converted to expressions of the form

$\text{case } (e_1, \dots, e_n) \text{ of } \{(p_1, \dots, p_n) \rightarrow exp\}$

Pattern matching in function definitions is converted by merging function clauses of the form

$id \ pats_1 = exp_1 \dots id \ pats_n = exp_n$

where id is the function identifier, $pats$ is any argument patterns and exp is the right hand side of each clause, into one single clause of the form

$id \ a_1 \dots a_n = \text{case } (a_1, \dots, a_n) \text{ of } \{pats_1 \rightarrow exp_1; \dots; pats_n \rightarrow exp_n\}$

where a_1 through a_n are variable pattern matches created to preserve function arity.

Function declarations with arguments are converted into a binding from the function name to a sequence of lambda expressions corresponding to the arity of the function. Functions of the form:

$id \ a_1 \dots a_n = rhs$

where id is the function identifier, a_1 through a_n are function arguments and rhs is the right hand side of the function, are converted into functions of the form

$id = \lambda a_1 \rightarrow \dots \lambda a_n \rightarrow rhs$.

The syntactic sugar for lists, $[e_1, \dots, e_n]$, where e_1 through e_n are arbitrary expressions, is converted into applications of the list constructor (cons), resulting in expressions of the form $((\text{:}) e_1 (\text{::} (\text{:}) e_n []))$.

Note that the cons operator is written in prefix notation, as an ordinary data constructor, instead of as an infix constructor, as is normally done when writing source code. This is because at this stage operator applications have already been removed from the language by the fixity resolution algorithm described in 4.3.3.4.

4.3.5 Compiler back-end

JSHC's back-end takes an intermediate representation in the form of an abstract syntax tree and compiles it to JavaScript code.

4.3.5.1 Internal libraries

To implement some basic functionality of the generated code which cannot be efficiently defined in Haskell (or is needed for our implementation to operate at all), an internal JavaScript library was developed. This library contains functions for handling integer arithmetics, arithmetic comparisons, lazy evaluation and pattern matching.

Since JavaScript does not have integers built in, the arithmetics for the Haskell integer types requires internal libraries that manage integer behaviour, such as overflow.

Laziness is implemented by creating a JavaScript object called `Thunk`. When a lazy expression is created, the strict JavaScript expression is wrapped in a JavaScript function (without parameters), which is subsequently wrapped in a `Thunk` object. When the result is needed, a method of the `Thunk` object is called, which calls the function and assigns the value to a variable. Whenever the result is needed again, the value of the variable is returned.

Pattern matching is handled by an internal function, `match`, that is sent an expression, e , and an array of pairs $\{pattern, rhs\}$, where $pattern$ is the pattern to match the result of e to, and rhs is a function that takes any bindings made in $pattern$ as arguments, and returns the right hand side expression of that particular pattern match alternative. The pairs are then matched sequentially, one at a time until a match is found or there are no more alternatives to match. When a match is found, `match` evaluates rhs with its bindings as arguments.

Matching of a pattern against an expression is done lazily and recursively.

Constructor patterns first evaluate e , then match the name and arity of the constructor in e , then recursively match all the constructor's arguments from left to right. Tuple patterns work in the same way as constructor patterns, but without the comparison of constructor names, as there are none.

Integer patterns first evaluate e and then return a positive match if the value of e is equal to the pattern.

Wildcard and variable patterns both match e without evaluating it, with the difference that variable patterns also add it to the set of bindings.

When there are no sub-patterns left to match, if the match has succeeded and rhs can be executed, if the match fails the matching continues on the next alternative.

4.3.5.2 Code generation

The code generation step generates JavaScript code as a JavaScript string representing the compiled modules. This code can then be accessed and executed by any JavaScript interpreter. For the generated code to be as easy to use in an external application as possible, the code generation preserves the names of as many identifiers as possible. This also allows for easier debugging of the generated code. Below follows a walkthrough of how components of the intermediate representation (Appendix A.2) are translated to JavaScript. If not defined in the text, names in italics refer to the meta variable of an item name in the description of the corresponding node in the intermediate representation (A.2).

4.3.5.2.1 Modules and declarations Modules are represented as JavaScript objects of the form $prefix.modid$, where $prefix$ is a module prefix specified by the user. Both $prefix$ and $modid$ can include `.`, which will create a nested object. E.g. substituting $prefix$ for `JSHC.mod` and $modid$ for `Data.List` will create the JavaScript object `JSHC.mod.Data.List`.

Following paragraphs will use *modid* to refer to the concatenation *prefix.modid*

Function declarations have the form $f = exp$, where f is an arbitrary variable identifier and exp is an arbitrary expression, and are compiled to JavaScript assignment statements of the form $f = exp'$ where f' is the compilation of the identifier f and exp' the compilation of the expression exp .

Data declarations contain constructors of the form $c t_1 \dots t_n$ which are all compiled into curried functions in the same way as lambda expressions, as described in 4.3.5.2.2. The type constructor is used to create a JavaScript object, which is used as the JavaScript constructor called by its data constructors when they have been applied to all arguments. The type variables and data constructor type arguments are not used.

4.3.5.2.2 Expressions Lambda expressions are compiled from a form $\backslash p_1 \dots p_n \rightarrow rhs$, where p represents arbitrary patterns, into a JavaScript function representing its curried [20] form:

```
function ( p1 ) { return function ( p2 ) { return ... function ( pn ) { return rhs } ... } }.
```

Tuple expressions are compiled into JavaScript objects containing an array with all expressions. Expressions of the form (e_1, \dots, e_n) are compiled into JavaScript constructor calls with the argument $[e_1, \dots, e_n]$.

Application of data constructors and functions are compiled in the same way; JavaScript's ordinary function application is used. Expressions of the form $f a_1 \dots a_n$ where f is an arbitrary function or data constructor expression and a represents its arguments, are compiled into curried JavaScript applications:

```
f(a1) ... (an).
```

This approach, coupled with how compilation of lambda expressions is handled, allows for partial application of functions. An application of the form $f(a_1) \dots (a_{n-2})$ will return a JavaScript expression of the form `function (pn-1) { return function (pn) { return rhs } }`.

Case expressions are compiled to a call to an internal function `match`, where the argument is an expression to match and an array of alternatives. Expressions of the form `case exp of alts` are compiled into `match(exp, alts)`. The workings of `match` are explained in 4.3.5.1.

Alternatives are compiled into pairs of patterns and right hand sides. Expressions of the form `pat -> exp` are compiled into anonymous JavaScript objects `{ p: pat, f: function((b1, ... , bn) { return exp } }` where b represents expressions bound in the pattern and f is a function that evaluates exp with the bound expressions as arguments.

4.3.5.2.3 Patterns and pattern matching Constructor patterns are compiled into JavaScript objects with one member containing the constructor identifier as a string, and one member containing an array with the constructor's arguments. Patterns of the form $D p_1 \dots p_n$, where D is an arbitrary data constructor and p represents arbitrary patterns, compile into a call to a JavaScript constructor with the string representing D and the array containing $p_1 \dots p_n$.

Tuple patterns are compiled into JavaScript objects containing an array with all tuple members. Patterns of the form (p_1, \dots, p_n) are compiled into JavaScript constructor calls with the argument $[p_1, \dots, p_n]$.

Wildcard patterns are compiled into a JavaScript object representing wildcard patterns.

Integer patterns are compiled into JavaScript objects containing an integer literal.

Variable patterns are compiled into an object containing the variable identifier as a string.

Pattern matching is compiled into calls to an internal function `match`, described in [4.3.5.1](#), that emulates the behaviour of Haskell's pattern matching in JavaScript.

4.3.5.2.4 Identifiers and literals Variable identifiers can be compiled in two ways. If they are qualified, they are declared at the top-level and thus compiled as members of the object representing the module, using JavaScript's associative array notation to enable members with non-alphanumeric characters as identifiers. A variable identifier *id*, which is a reference to an identifier declared at the top level in any module, is compiled into JavaScript: `modid["id"]` where *id*' is the unqualified identifier for *id*.

If a variable identifier is not qualified, it is declared locally, and is compiled as a JavaScript string.

Data constructor identifiers can only refer to top-level declarations, and will always be compiled the same way as top-level variable identifiers.

Since the foreign function interface as described in the Haskell report[23] is currently not implemented, any identifier located in the module `JSHC.Internal` is treated as reference to a JavaScript function in the internal libraries. These internal functions will be compiled into a curried function just as for lambda expressions [4.3.5.2.2](#), but both parameters types and the return type will also be converted between the JavaScript type and the corresponding Haskell type as needed. e.g The JavaScript implementation of the `>= :: Int32 -> Int32 -> Bool` operator is `JSHC.Internal.int32ge`, where no parameter is converted as the representation is the same in JavaScript, but the result will be converted from a JavaScript boolean value to the `True` or `False` data constructor of `Prelude.Bool`. If the internal function takes no parameters (such as the implementation of `undefined`, which just throws a JavaScript exception) it is just an expression to be evaluated.

4.3.5.2.5 Laziness To achieve lazy evaluation, the evaluation of some compiled expressions must be delayed.

Affected expressions are case expressions, function application, and compiled internal functions with 0 parameters, while unaffected expressions are integer constants, functions (lambda expressions and compiled internal function, like constructor applications, `s` with `>0` parameters), and tuples.

The laziness is achieved by wrapping a compiled expression *e* in a `Thunk` and a function as described in [4.3.5.1](#). *e* will now only be evaluated when the thunk is evaluated.

To prevent evaluation of function arguments, all function arguments are `Thunks`. Since the result of a function may be the argument to another function, but not used within it, results of functions are also thunks.

Whenever an expression is compiled, the result is a pair containing the string of the compiled Haskell expression and a boolean with the strictness of the expression.

Whenever a sub-expression is lazy and need to be strict, the sub-expression `Thunk` is evaluated. If a sub-expression was strict and need to be lazy, the sub-expression is wrapped in a `Thunk` without function call, as the function is not necessary.

Since a name in a top-level binding might be accessed before it's expression is compiled (e.g recursive definitions), all expressions in top-level bindings are wrapped in a `Thunk` and a function to prevent reading it until after the code generation is complete.

4.3.6 Standard libraries

The standard Haskell libraries used for JSHC are a subset of the libraries described in [23]. The contents of the JSHC Prelude are detailed in [Table 4.2](#).

Name	Description
Int32	32 bit integer datatype
Bool	Boolean datatype
Maybe a	Nullary datatype
Either a b	Choice datatype
&&, , not, otherwise	Boolean functions
undefined	Undefined function
+, -, *, /	Arithmetic operators
<, >, <=, >=, /=, ==	Arithmetic comparison operators
max, min, negate, abs, signum	Arithmetic functions
id	Identity function
const	Constant function
.	Function composition
flip	Argument switch
seq	Strict evaluation
\$, \$!	Function application operators
map, ++, filter, concat, concatMap, head, tail, null, length, foldl, foldl1, foldr, foldr1, iterate, repeat, replicate, cycle, take, drop, splitAt, reverse, and, or, any, all, elem, notElem, sum, product, maximum, minimum	List functions
fst, snd	Tuple functions
foldr	Right fold

Table 4.2: Functions and datatypes included in the JSHC Prelude

Chapter 5

Discussion

5.1 Implementation details

5.1.1 Export checking

The only implementation of Haskell that we have found that implements mutually recursive modules is Programatica [6]. Hugs[3] and GHC[31] do not implement mutually recursive modules as specified in the Haskell report. We were unsure of how to implement the computation of the exported names, as mutually recursive modules means that to compute the exports of one module in the cycle, it appears that one must also compute the exports of all the others. The only idea we had that seemed to actually work was a fix point iteration to find exported names until no more names can be found. One concern was efficiency, and hope of finding a simpler and more efficient way of computing the exported names. It appears to us that most people try to avoid module cycles, so we had some trouble finding resources about implementing it, but eventually found[21], which formally describes the Haskell module system using a fix point computation on, but we have not had enough time to implement something like it.

5.1.2 Dependency checking

Dependency checking was not expected to be as big a part of the implementation as it turned out to be. It is something which one does not see as a Haskell user, and is not needed in many other languages that we have experience with, such as C or Java, as they do not have cyclic dependencies between language constructs. It is also only scarcely mentioned in the specification ([23] chapters 4.5 and 4.6), which somewhat hides the fact that it is a central concept in the workings of a Haskell compiler.

5.2 Comparison with other implementations

We have chosen to only compare with other Haskell interpreters/compiler written in JavaScript, and we only know of one.

5.2.1 Other possible solutions to client-side Haskell

If making a plug-in, one still required writing a back-end to JavaScript and implementing an FFI to JavaScript code.

If compiling an existing compiler like GHC to JavaScript, it would require an existing JavaScript back-end, and the performance of it would be important. There seems like there would be issues with how I/O is used within the compiler since it is not available in

JavaScript, which would mean that the compiler need to be modified and can not just be compiled as is.

5.2.2 HIJi (Haskell in JavaScript)

This is a Haskell interpreter written in JavaScript as part of a bachelor thesis project.

Their parser is not generated from a grammar description, but instead written in JavaScript using a parser combinator library called JSParse[14].

From the report[10], it is unclear which exact subset of Haskell they have implemented, as it is not specified, and there is no mention of mutually recursive modules or dependency groups.

They have an interpreter that loads an unknown subset of the Prelude and allows evaluation of expressions. There are no error messages and no type checking is done as the type checker is not fully integrated with the rest of the code, but they have an implementation of type classes according to their report.

Wildcards and prefix operators does not appear to work in the interpreter, but they do support guards and sections, which we do not.

It is supposed to be possible to load modules by placing them on the same server as the interpreter in the same way as in our compiler, but we were unable to load anything.

The interpreter has a history of written expressions, which we have yet to implement.

5.3 Use of JSHC for web programming

As it is, JSHC works reasonably well for the subset of Haskell it supports, and provides both static type checking and a module system, as well as other high-level features (such as lazy evaluation) to the web-scripting domain, and it works as a proof of concept in that regard. It can be used as a library in a website to compile and run programs in the subset of Haskell that is supported. The Ymacs interface can also potentially be used by users new to Haskell who want to try the language out.

However, developing advanced web applications in JSHC would be problematic, as several key features needed to make this practical are missing (as described in 5.4).

In addition, the code generated by JSHC has not been benchmarked for efficiency, and is likely many times slower than hand-written JavaScript (mostly because of the laziness). To be able to advertise JSHC as a feasible development platform, optimisation and benchmarking of the generated code needs to be done.

5.4 Future work on JSHC

5.4.1 Additional support of the Haskell specification

As only a small subset of Haskell is supported in JSHC, the most important future work is to extend its support to larger parts of the Haskell specification (and eventually to supporting it in full).

There are several minor syntactic features missing, such as sections (syntactic sugar for partial application of binary operators), list comprehensions and guards.

Some major missing features are described below.

5.4.1.1 FFI: Foreign Function Interface

Adding a foreign function interface to JavaScript code would simplify the writing of the Prelude, as primitive operations such as addition could be described using foreign imports instead of a name with a special prefix for which the type would be stored in the type checker when looking up the name instead of the type being written in a foreign function declaration.

More importantly, it will also allow writing Haskell libraries that wrap JavaScript APIs, which would greatly increase the flexibility in the development capabilities of JSHC.

5.4.1.2 Type classes

Function overloading through type classes is one of the most important features to add. Overloading is an important part of Haskell, and is used by many of the most common functions in the standard Prelude.

5.4.1.3 Input / output

Haskell's input/output needs to be implemented. Since most implementations of JavaScript do not define standard I/O channels such as `stdin`, `stdout` and `stderr`, some kind of simulation of these would be necessary. For example, `stderr` could be represented by a browser's JavaScript console. A file system would have to be simulated, for example as is done in JSHC to store the code in a buffer, by a JavaScript map from file names to contents.

5.4.2 Possible additions to the compiler

5.4.2.1 Loading files asynchronously

This is not difficult, but since only local files or files on the same domain may be accessed, there does not appear to be a reason to implement this.

5.4.2.2 Pretty printer

To give good error messages related to expressions within functions, a pretty printer is required. Currently, data constructors are shown using unnecessary parentheses, and only the abstract syntax tree is available for expressions and patterns.

5.4.2.3 Coloring Haskell code

A syntax-highlighting mode for Haskell code in the Ymacs buffers should be added.

We find the existing library support for coloring in Ymacs buffers insufficient for our needs, so a custom library needs to be implemented.

5.4.2.4 Concurrency

To speed up compilation, the dependency group traversal could be implemented to run concurrently, as the implemented algorithm allows for taking out many groups which do not depend on each other. Even though extensions for concurrency exists, speed is not important in the compiler in it's current state. Lexical analysis and parsing of modules could also benefit from added concurrency.

5.4.2.5 Code optimisation

As code generation has been designed without regard to speed, the generated code is most likely slower than necessary. For example, the naive method of solving lazy evaluation used in JSHC (wrap everything that might need to be lazy in a `Thunk`, see [4.3.5.2.5](#)) is unnecessarily inefficient. One can reduce the amount of lazy evaluation used in a program by implementing **strictness analysis**[\[27\]](#).

5.5 Issues

During the course of the project, we encountered some issues that significantly affected our workflow and deserve mention.

5.5.1 Project size

As mentioned in 1.3, the intention of this project was initially to implement the full Haskell 2010 specification. This was, however, not possible within the scope of a bachelor's thesis. Instead, a small subset of Haskell was implemented (described in 4.1).

There are several reasons for this underestimation of project size.

Our experience in writing compilers comes mainly from courses we have attended, where we have implemented compilers for simple imperative and functional languages. As we managed these courses with relative ease, implementing a full scale programming language like Haskell seemed feasible.

As the Haskell specification is often brief in its description of complex language features and implementation details, such as polymorphic type checking or overloading through type classes, as well as the features discussed in 5.1, the challenge of implementing seemed lesser than it actually was.

In addition, none of the group members had used JavaScript before the start of this project. The problems caused by JavaScript's weak dynamic typing and lack of module system[25] were surprisingly difficult to overcome as the project grew.

5.5.2 The Haskell grammar

Certain features of the Haskell grammar occurred to us as peculiar and/or problematic when implementing the language.

The grammar in the Haskell report is at least an LR(2) grammar as it requires a lookahead of 2 when reading top declarations, and it has conflicts as an LALR grammar. This means that it must be rewritten for use in a LALR(1) parser generator such as Jison or Bison. This is problematic, as rewriting the grammar can be a source of errors that can be hard to detect. The grammar also contains at least one ambiguity. Both the `impdecl` and the `gendec1` have empty productions, and both can be present at the top level. Although this ambiguity only applies to programs with empty declarations, and should not affect implementations, allowing for ambiguities in the formal specification of a language strikes us as bad practice, as it potentially makes the grammar less clear and also harder to implement a parser for.

Another issue we had was that the specification defines some declarations as legal where the semantics are unclear or nonsensical. For example, `2 = 3` is a valid top level declaration in Haskell, which, we assume, defines a pattern match which can never succeed as it is never reached to begin with.

5.5.3 Interpreter: accessing URLs

The idea was that the interpreter should support adding paths to Haskell files residing on any url or file path. Unfortunately, for safety reasons, most common browsers prevent getting files from domains other than the one that the script is executed on. If using Google Chrome/Chromium, the security policy even prevents accessing local files using the `file://` protocol when running a script from a local directory i.e. not through a web server. The Ymacs interface must be placed on a web server to work for browsers with such a restriction.

Bibliography

- [1] ghcjs. <https://github.com/sviper11/ghcjs>, May 2011.
- [2] Git - fast version control system. <http://git-scm.com/>, May 2011.
- [3] The hugs 98 user's guide. http://cvs.haskell.org/Hugs/pages/users_guide/index.html, May 2011.
- [4] Open source initiative osi - the bsd 3-clause license. <http://www.opensource.org/licenses/BSD-3-Clause>, June 2011.
- [5] Open source initiative osi - the mit license. <http://www.opensource.org/licenses/mit-license.php>, June 2011.
- [6] The programatica project. <http://programatica.cs.pdx.edu/>, May 2011.
- [7] W3counter - global web stats. <http://www.w3counter.com/globalstats.php?year=2011&month=4>, May 2011. Statistics over global web browser usage April 2011.
- [8] V. Aho, M.S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2007.
- [9] Mihai Bazon. Ymacs - ajax source code editor. <http://www.ymacs.org/>, March 2011.
- [10] Adam Bengtsson, Mikael Bung, Johan Gustafsson, and Mattis Jeppsson. Haskell in javascript. Bachelor's thesis, Department of Computer Science and Engineering Chalmers University of Technology, Division of Computer Engineering Gothenburg University, May 2010.
- [11] Zachary Carter. Jison. <http://zaach.github.com/jison/>, March 2011.
- [12] Atze Dijkstra. Haskell to javascript backend. <http://utrechthaskellcompiler.wordpress.com/2010/10/18/haskell-to-javascript-backend/>, May 2011.
- [13] Charles Donnelly and Richard Stallman. *Bison - The Yacc-compatible Parser Generator*. Free Software Foundation, Boston, 2.4.3 edition, 2010.
- [14] C. Double. Jsparse. <https://github.com/doublec/jsparse>, May 2011.
- [15] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*, 5th edition, December 2009.
- [16] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.
- [17] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 111-122, Venice, Italy, January 2004.
- [18] Dmitry Golubovsky. Haskell web toolkit. http://www.haskell.org/haskellwiki/Haskell_in_web_browser, May 2011.

- [19] Michael Hanus. Putting declarative programming into the web: Translating curry to javascript. In *PPDP07 Proceedings of the 9th ACM SIGPLAN international conference on principles and practice of declarative programming*, pages 155--166, Venice, Italy, July 2007.
- [20] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21:359--411, September 1989.
- [21] Thomas Hallgren Iavor S. Diatchki, Mark P. Jones. A formal specification for the haskell 98 module system. In *Proceedings of the 2002 Haskell Workshop*, 2002.
- [22] David Majda. Peg.js - parser generator for javascript. <http://pegjs.majda.cz/>, March 2011.
- [23] S. Marlow (editor). *Haskell 2010 language report*. 2010.
- [24] Erik Meijer. Server side web scripting in haskell. *Journal of Functional Programming*, 10:1--18, 2000.
- [25] Tommi Mikkonen, Antero Taivalsaari, Tommi Mikkonen, and Antero Taivalsaari. Using javascript as a real programming language. Technical report, Sun Labs, 16 Network Circle, Menlo Park, CA 94025, 2007.
- [26] Neil Mitchell. Yhc is dead. <http://yhc06.blogspot.com/>, April 2011.
- [27] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269--281. Springer Berlin / Heidelberg, 1980.
- [28] Thiemann Peter. Modeling html in haskell. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*, pages 263--277. Springer Berlin / Heidelberg, 2000.
- [29] Benjamin C. Pierce. *Types and Programming Languages*. 2002.
- [30] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic web documents. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 290--301, New York, NY, USA, 2000. ACM.
- [31] The GHC Team. The glorious glasgow haskell compilation system user's guide, version 7.0.3. http://www.haskell.org/ghc/docs/latest/html/users_guide/, May 2011.
- [32] Johan Tibell. Results from the state of haskell, 2010 survey. <http://blog.johantibell.com/2010/08/results-from-state-of-haskell-2010.html>, May 2011. Statistics from a survey sent to Haskell programmers regarding their usage of Haskell.

Appendix A

Syntax reference

Detailed in this appendix is a reference to the subset of the Haskell syntax supported by JSHC, as well as the intermediate representation used by the code generation.

A.1 Context-free syntax

The same notational convention used in [23] is used here, i.e:

```
[pattern] : optional
{pattern} : zero or more repetitions
(pattern) : grouping
pat1 | pat2 : choice
nonterm -> alt1 | alt2 | ... | altn : production

module → module modid [exports] where body
      | body

body → { impdecls ; topdecls }
     | { impdecls }
     | { topdecls }

impdecls → impdecl1 ; ... ; impdecln (n ≥ 1)

exports → ( export1 , ... , exportn [ , ] ) (n ≥ 0)
export → qvar
       | qtycon [(..) | ( cname1 , ... , cnamen )] (n ≥ 0)
       | module modid

impdecl → import modid [impspec]
        | (empty declaration)
impspec → ( import1 , ... , importn [ , ] ) (n ≥ 0)
        | hiding ( import1 , ... , importn [ , ] ) (n ≥ 0)
import → var
        | tycon [ (..) | ( cname1 , ... , cnamen )] (n ≥ 0)
cname → var | con

topdecls → topdecl1 ; ... ; topdecln (n ≥ 0)
topdecl → data simpletype [= constrs]
        | decl

decls → { decl1 ; ... ; decln } (n ≥ 0)
decl → gendekl
```

```

    | (funlhs | pat) rhs
gendecl → vars :: type (type signature)
    | fixity [integer] ops (fixity declaration)
    | (empty declaration)

ops → op1 , ... , opn (n ≥ 1)
vars → var1 , ... , varn (n ≥ 1)
fixity → infixl | infixr | infix

type → btype [-> type] (function type)
btype → [btype] atype (type application)
atype → gtycon
    | tyvar
    | ( type1 , ... , typek ) (tuple type, k ≥ 2)
    | [ type ] (list type)
    | ( type ) (parenthesized constructor)
gtycon → qtycon

simpletype → tycon tyvar1 ... tyvark (k ≥ 0)
constrs → constr1 | ... | constrn (n ≥ 1)
constr → con [!] atype1 ... [!] atypek (arity con = k, k ≥ 0)

funlhs → var apat { apat }
    | pat varop pat
    | ( funlhs ) apat { apat }
rhs → = exp [where decls]

exp → infixexp :: type (expression type signature)
    | infixexp
infixexp → lexp qop infixexp (infix operator application)
    | - infixexp (prefix negation)
    | lexp
lexp → \ apat1 ... apatn -> exp (lambda abstraction, n ≥ 1)
    | let decls in exp (let expression)
    | if exp [;] then exp [;] else exp (conditional)
    | case exp of { alts } (case expression)
    | fexp
fexp → [fexp] aexp (function application)
aexp → qvar (variable)
    | gcon (generalized constructor)
    | literal
    | ( exp ) (parenthesized expression)
    | ( exp1 , ... , expk ) (tuple, k ≥ 2)
    | [ exp1 , ... , expk ] (list, k ≥ 1)

alts → alt1 ; ... ; altn (n ≥ 1)
alt → pat -> exp [where decls]

pat → lpat
lpat → apat
    | gcon apat1 ... apatk (arity gcon = k, k ≥ 1)
apat → var
    | gcon (arity gcon = 0)
    | literal
    | _ (wildcard)
    | ( pat ) (parenthesized pattern)

```

| (pat₁ , ... , pat_k) (tuple pattern, k ≥ 2)

gcon → [] | qcon (generalized constructor)
var → varid | (varsym) (variable)
qvar → qvarid | (qvarsym) (qualified variable)
con → conid | (consym) (constructor)
qcon → qconid | (gconsym) (qualified constructor)
varop → varsym | ` varid ` (variable operator)
qvarop → qvarsym | ` qvarid ` (qualified variable operator)
conop → consym | ` conid ` (constructor operator)
qconop → gconsym | ` qconid ` (qualified constructor operator)
op → varop | conop (operator)
qop → qvarop | qconop (qualified operator)
gconsym → : | qconsym

A.2 Intermediate representation

For detailing the intermediate representation, these notational conventions are used:

node → { item₁ , ... , item_n }
item → name: value | [name: value]
rule₁ | rule₂ : choice
name : the name of a n item inside a node
value : the value of an item inside a node, values can be both terminals and nonterminals

module → {name: "module", modid: <modname>, body: <body>}

body → {name: "body", topdecls: <topdecls>}

topdecls → <topdecl>₁ ... <topdecl>_n

topdecl → {name: "topdecl-decl", decl: <decl>}

 | {name: "topdecl-data", typ: <simpletype>, constrs: <constrs>}

decl → {name: "decl-fun", ident: <varname>, rhs: exp}

simpletype → {name: "simpletype", tycon: <dacon>}

constrs → <constr>₁ ... <constr>_n

constr → {name: "constr", dacon: <dacon>, types: <types>¹}

exps → <exp>₁ ... <exp>_n

exp → {name: "lambda", args: <args>, rhs: <exp>}

 | {name: "case", exp: <exp>, alts: <alts>}

 | {name: "application", exps: <exps>}

 | {name: "tuple", members: <exps>}²

 | {name: "negate", exp: <exp>}

 | literal

 | dacon

 | varname

alts → <alt>₁ ... <alt>_n

alt → {name: "alt", pat <pat> exp: <exp>}

pats → <pat>₁ ... <pat>_n

pat → {name: "conpat", con: dacon, pats: <pats>}

 | {name: "tuple-pat", members: <pats>}²

 | {name: "wildcard"}

 | varname

 | dacon

 | literal

modname → {name: "modname", id: dacon}

dacon → {name: "dacon", id: <STRING>}³

varname → {name: "varname", id: <STRING>, [loc: <STRING>]}³

literal → {name: "integer-lit", val: <NUMBER>}

STRING → a JavaScript String

¹Types is an array containing the arguments to a data constructor. At code generation, the arguments themselves are irrelevant, only the length of the array matters.

²At this stage, it is legal for a tuple or tuple-pat to have members.length == 1

³loc is used to determine whether a dacon or varname is qualified, and what that qualification is. At this stage, anything that is not qualified is a local variable.

NUMBER -> a JavaScript Number

Appendix B

Declaration of contributions

Both group members have been equally active in the project, from planning, through execution to final stages.

Regarding implementation, Peter Holm has had the main responsibility for lexical analysis and code generation, while Staffan Björnesjö has had main responsibility over large part of the semantic analysis (name check, type check), dependency group analysis, loading of modules, the module system and user interface. All other parts have been developed collaboratively.

Regarding report, participants have been responsible for documenting their areas of responsibility in the implementation. All other parts have been documented collaboratively.