UNIVERSITY OF GOTHENBURG

A Structured Editor for Labelled Bracket Notation

MASTER OF SCIENCE THESIS IN THE PROGRAMME COMPUTER SCIENCE

JOHAN IVARSSON

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2012

A structured Editor for Labelled Bracket Notation

JOHAN IVARSSON

Examiner: Bengt Nordström

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

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

ABSTRACT

This report describes the development and resulting product of a web-based editor for a simple programming language. This language is an extension of simply typed lambda calculus and labelled bracket notation, with a simple type-system. Each editing operation will only construct well-typed programs.

The report is written as a basis for someone with some technical knowledge of type-theory and programming, who wants to further develop the application.

The application and its source code can be accessed on http://bonzay.se/dev/master.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE

The purpose of this work is to develop an editor for a simple language incorporating the basic building blocks of XML and the labelled bracket notation used by linguists. The editor will run in a browser. It will present the objects being edited in different ways (indented text strings, trees). There is a simple typing system to this language and each editing operation will only construct type correct objects.

Further developments of this include:

- flexible notation for concrete syntax (a la GF[1])
- a more advanced typing system (with dependent functional types)

- Bengt Nordström, Examiner

1.2 OBJECTIVE

The editor uses placeholders for incomplete expressions. A placeholder is an empty node that only has a type in the abstract syntax tree. The tree is being constructed by replacing these placeholders with expressions. Each placeholder can be replaced by a node if and only if the typing rules are not violated.

The application should be able to:

- Select a node.
- Replace a selected node with a placeholder (deletion).
- Replace a selected placeholder with a node (insertion).
- Create a placeholder.
- Save the content in the editor.
- Load the content to the editor.

The application should also be developed with usability in mind. All commands and interactions should be as intuitive and simple as possible.

[1] http://www.grammaticalframework.org/

1.3 Scope and Limitations

The application should run in a web-browser, which means that it must work for different browsers. This can be a big problem because many of these have different engines and will interpret the code differently. In this project, cross-browsing is not a priority and the application will be developed to work only in Google Chrome. The application will still be working on different platforms because Google Chrome is available for all major operating systems.

1.4 Theory

1.4.1 Trees

A tree is a simple and common way of representing hierarchical data. A tree is built up with nodes. Each node stores some data and has a parent and zero or more child nodes. The topmost node is called a root and is most often the starting point of operations on the tree. Tree structures are very common in Linguistics and Mathematics. For example the mathematical expression

$$(1 + 2) * 3$$

can be represented by the tree in figure 1.1. The parentheses are not present in this representation but its meaning is represented by the structure of the nodes.

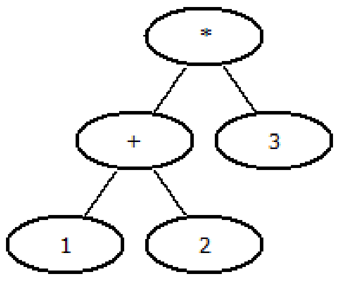

Fig. 1.1: A tree representation of a mathematical expression

In computer science, an abstract syntax tree is often used to represent the source code of a program. Each node will correspond to a constructor, operator or variable of the source code. Its recursive structure makes is simple for a computer to read and manipulate the data stored in it.

The Java program in figure 1.2 will have the abstract syntax tree in figure 1.3

```
if (plus)
         return x + y;
else
         return x - y;
```

Fig. 1.2: A java program

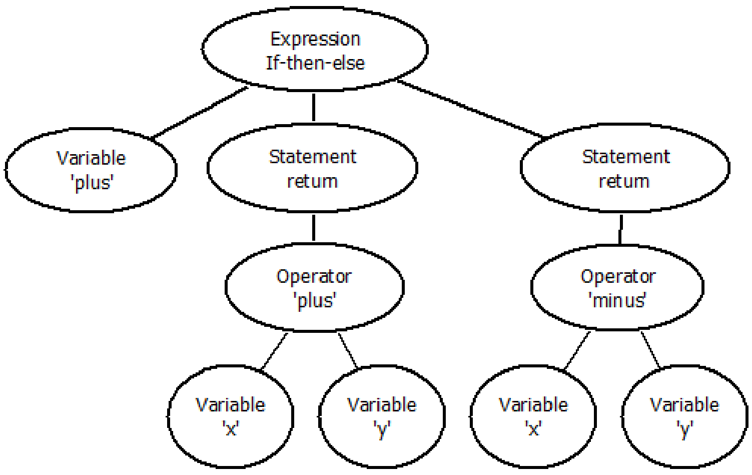

Fig. 1.3: An abstract syntax tree of a simple Java program

1.4.2 Types

"Well-typed programs can't be blamed."

- Philip Wadler and Robert Bruce Finder [1]

Most modern programming languages have some kind of type-system. This is a system that classifies each expression by a type, and then the type-checker tries to prove that no type error has occurred. If a type error is found, the program cannot be guaranteed to execute correctly and can often crash because of errors. If there are no type errors, the program is said to be "well-typed" and this can prevent many execution errors. A type system is used to eliminate errors at an early stage during the development. It can also be used to help the developer reason about the code and simplify proofs.

1.4.3 Web-applications

Every single web-page and web-application on the internet is, by standard, structured and built with HTML[2]. HTML is short for "Hypertext mark-up Language" and describes the layout of a web-page. The HTML code is constructed with tags that have a number of arguments and children that

[2] http://sv.wikipedia.org/wiki/HTML

describe both structure and properties for each tag. These tags build a treelike structure called the DOM (Document Object Model) tree. HTML has no support for programmable functionality or user interaction; therefore another language must be used.

JavaScript is not a standard, but it is by far the most used scripting language used to add functionality such as store information, read input from the user and manipulate the DOM tree in real time. Normally when the client gets data from the server, the web-page needs to be reloaded which takes time and JavaScript loses all information stored in the current session. This can be solved by making an AJAX[3] request. This retrieves information from the server in the background, without the page being reloaded. When passing data in this way, a Json[4] object is very convenient to use. It is a string object that is very similar to a map or dictionary.

1.4.4 Parsing

Parsing is the process of converting the program as a string to an abstract syntax tree so the computer can work with it. The process is generally divided into two parts (figure 1.4): lexical and syntactic analysis. The first part divides the string into a list of tokens where each token represent a meaningful symbol. These tokens are passed on to the second part, the parser. Here a tree is created to represent the program.

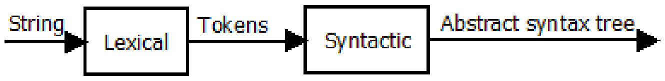

Fig. 1.4: Parsing process

1.4.5 Editors

To make the development process easier, programmers use special text-editors to construct the programs. These editors often have a type-checker and parser implemented, so type errors and parse errors can be presented immediately. It can also suggest values of functions and variables that are available in the program. This is called Autocomplete. Example of such editors is Eclipse[5] and Microsoft Visual Studio[6].

[3] http://sv.wikipedia.org/wiki/AJAX
[4] http://www.json.org/
[5] http://www.eclipse.org/
[6] http://msdn.microsoft.com/en-us/vstudio/aa718325.aspx

1.4.6 Languages

1.4.6.1 Simply typed lambda calculus

Lambda calculus is a very simple, yet powerful, programming language. It is Turing complete which means that it can express and evaluate every computable function.

It is very clumsy to write programs in this language, but this is not the main reason for it. It was constructed to formalize and study concepts as recursion and computability. Its syntax is defined by the BNF in figure 1.5:

```
e ::=      X      variable
         | e e    application
         | \X.e   abstraction
```

Fig. 1.5: BNF for lambda calculus

where X is a variable name.

An extension to this language is to add a simple type system. We need to introduce the syntax for types, and change the abstraction expression to include this. This is shown in figure 1.6. The typing rules are shown in figure 1.7.

```
e ::=      X      variable
         | e e    Application
         | \X:t.e Abstraction


t ::=      T      base type
         | t -> t functional type
```

Fig 1.6: BNF for simply typed lambda calculus and the type system

$$\frac{f \in A \to B \quad a \in A}{f\ a \in B} \qquad \frac{b \in B \quad [x \in A]}{\lambda x. b \in A \to B}$$

Fig. 1.7: Typing rules for simply typed lambda calculus

This language will no longer be Turing complete, but it is guaranteed to always terminate. This extension helps the reasoning about types and simplifies proofs.

1.4.6.2 Labelled bracket notation and XML

Labelled bracket notation is commonly used by linguistics to; for example, name the components of a sentence. Each bracket has a name and some arguments. Fig 1.8 shows the English sentence "John hit the ball" divided into a sentences base components.

$[_{\text{sentence}}\ [_{\text{noun}}\ \texttt{John}]\ [_{\text{verb phrase}}\ [_{\text{verb}}\ \texttt{hit}]\ [_{\text{noun phase}}\ [_{\text{determiner}}\ \texttt{the}]\ [_{\text{noun}}\ \texttt{ball}]]]]$

Fig 1.8: Structure of the sentence "John hit the ball"

This notation is very similar to XML. A part of the same sentence is shown in Fig. 1.9.

```
<verb phase>
         <verb> hit </verb>
         <noun phase>
                 <determiner> the </determiner>
                 <noun> ball </noun>
         </noun phase>
</verb phase>
```

Fig 1.9: Example of XML

2 METHODS

2.1 CHOICE OF PROGRAMMING LANGUAGES

The whole point of making the editor as a web application is to increase the availability so that users can access their programs as long as they have an internet connection. This means that a login system is needed to preserve confidentiality and integrity. The authentication information and the programs need to be stored somewhere, thus a server is needed.

The Applications is developed using HTML and JavaScript. A JavaScript library called jQuery[7] was also used. This is a tool that simplifies DOM node selections, event handling, AJAX requests and much more.

The server side was programmed in PHP. This language was chosen because it is free, simple and powerful. PHP is also very popular and was run by over two million servers all over the world in April 2007 [2].

2.2 DEVELOPMENT

The project consists largely of programming. Therefore an iterative and incremental approach used. This method is based on building a partial application and then expanding it, so start with the basic functionality and then refine the application until the goals are reached. This way of developing will make it possible to always have a suitable testing environment.

To test the application a local server called WampServer[8] was used. This program is a development environment that installs all required programs needed to develop a web application with PHP.

During the development phase, a versioning program was used to handle all code-files and documentation. This helped with backups and to distribute everything among all computers that were used.

[7]http://jquery.com
[8] http://www.wampserver.com/en/

3 Result

The application and its source code can be accessed on http://bonzay.se/dev/master.

3.1 The Program

When the application is freshly started, the login screen will appear (figure 3.1). This login system was needed to be able to bind saved programs to a user, to preserve some level of confidentiality and integrity. If the user has a valid username and password, that can be used to login here. In other cases, the user can press the demo link, which will let them access the editor but the save and load functionality will be disabled. There is currently no support for creating usernames except inserting it directly into the database.

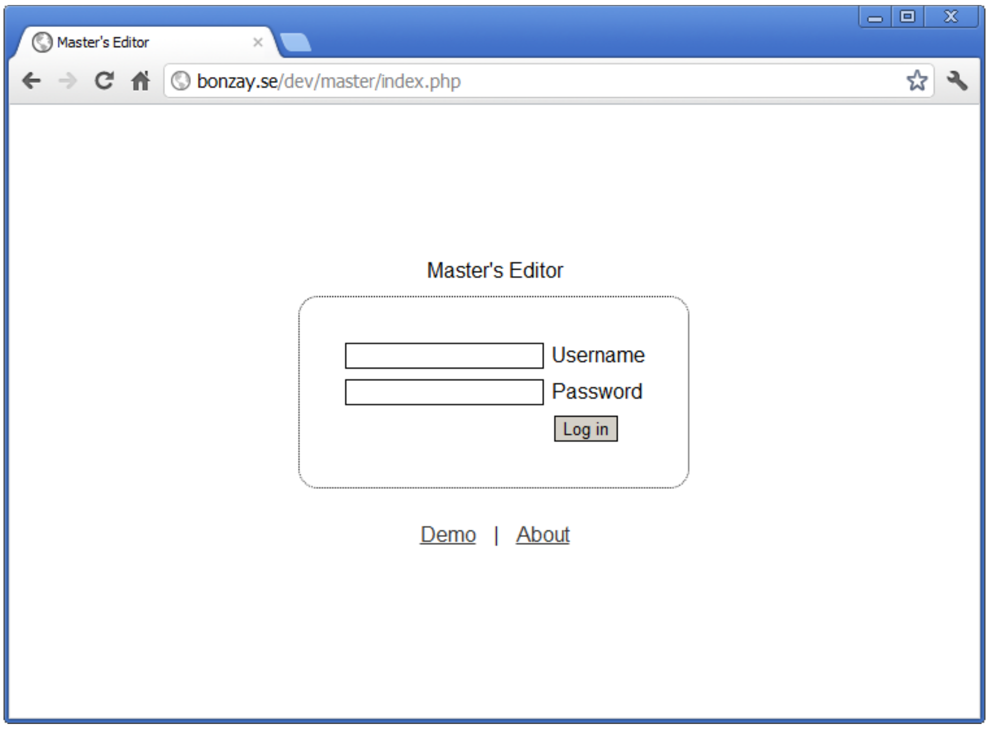

Fig 3.1: Login screen

When the login is successful, the user comes to the main part of this application: the editor itself (figure 3.2). There is a menu with buttons at the top, where the user can for example save, load or create a new program. At the bottom of the screen, there is a status bar. When a node is selected, this will show the type of that node. It will also show information like an error message if there is one available. The editor takes up all space between the menu and the status bar.

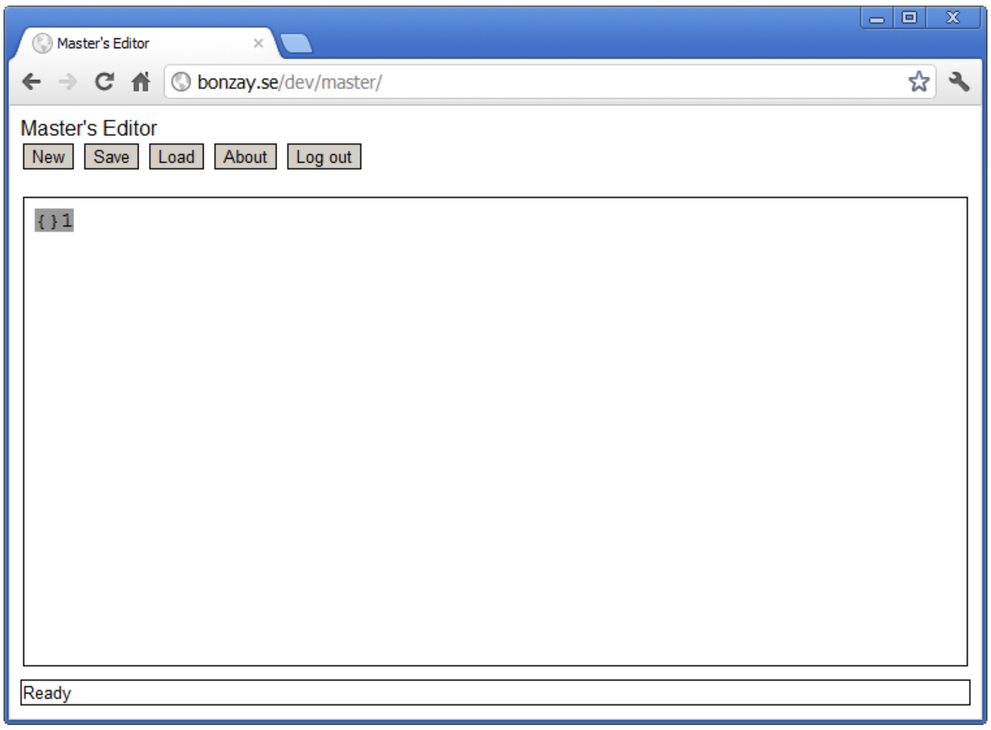

Fig. 3.2: The editor

Some keywords cannot be used by the user as a variable name. There words are "let", "cons", "Decl", "Ident" and "Type". The first two are keywords used to define new constants and create expressions. The last three are used as types. These types are used by the application itself and cannot be used by the user.

3.2 Application Usage

When developing this application, usability was always in mind. The controls are meant to be as intuitive as possible. Many of the keyboard shortcuts are the same or similar to other editors.

The user must make a selection to modify a part of the program. Only one selection can be made at one time. A selection is presented with a blue background, and is made by either clicking on a node with the mouse or jumping between nodes by pressing key-shortcuts.

The user can move the selection between placeholder by pressing the tab-key. If the shift key is hold down at the same time, the order will be reversed. To move the selection between all expressions in the program, hold down Ctrl and press the left and right arrow keys.

The space and enter character are normally used to separate words. But since this application does not allow the user to edit programs as text strings, these characters will never be used as a valid input from the user. Instead, these keys are used to parse the code in the selected placeholder, and if the parsing was successful the current placeholder will be replaced with the expression it was parsed to. The selection will also be moved to another placeholder if possible. If the enter-key was pressed, the editor will create a new placeholder at the bottom of the program and set the selection there, and if the space-key was pressed, the selection will move to the next placeholder in a textual order.

After the user has grown custom to this was of editing a program, these controls will almost give the user the same flow of typing as in a normal text-editor.

The complete set of keyboard shortcuts used in this editor is shown in figure 3.3

[SHIFT +] SPACE	If a placeholder is selected, parse the data and move the selection.
ENTER	If a placeholder is selected, parse the data and create a new placeholder at the bottom of the program.
[SHIFT +] TAB	Move selection to the next placeholder. If shift is down, reverse the order.
CTRL + (LEFT | RIGHT)	Move selection through all expressions and sub-expressions, backwards and forwards.
CTRL + S	Save
CTRL + O	Open
(DELETE | BACKSPACE)	If a non-placeholder is selected, convert this node into a placeholder, removing all its sub-nodes.

Fig. 3.3: Keyboard shortcuts

3.2.1 Example of creating a program

Here follows an example of creating the simple program in figure 3.4, using the editor. This description is also illustrated step-by-step in figure. 3.5. When referring to this figure, the notation "(X.)" is used, where X is the step in the figure.

```
cons Nat : Set;
cons Zero : Nat;
cons Succ : Nat -> Nat;
let Two : Nat = (Succ (Succ Zero));
```

Fig 3.4: A simple program

When the application is started the editor will contain one placeholder with type 'Decl' (1.). This placeholder can be replaced with either the string "cons" or "let". We write 'cons' and the press space-key. The editor parses the text, and creates a ‘Cons’ node which needs two arguments to be valid, so two new placeholder is created (2.).The first placeholder will have type ‘Ident’ which means that it wants a unique variable name. We name it "Nat" (3.) and press space. The other placeholder hat type ‘Type’. We write “Set” here (4.) and press enter. This will parse and create a new placeholder with type ‘Decl’ at the bottom.

We continue to build the program similarly to step (11.). Here we want to create a constant with a functional type. For this we need to write the functional arrow “->” first. This will give us two new placeholders, both with type ‘type’ (12.). We can use “Nat” on both of these (14.) and our definition of the Natural numbers is complete.

At step (15.), a ‘let’ declaration needs three arguments to be valid so three new placeholders is added (16.). The first two are the same as a ‘cons’ declaration, but the third one (placeholder number 15) is a bit different. This expression will have the type that is defined by placeholder number 14. As long as placeholder number 14 is not parsed correctly, placeholder number 15s type will be unknown, and the parser can’t parse it. We give it type “Nat” (18.). In step (19.) we write it “Succ”. This have type “Nat -> Nat” which means that it will have type “Nat” when applied to an expression with type “Nat”, Therefor the editor will add a placeholder for this. Similar to step (21.) where the application is finished.

```
1.  {}1
2.  cons {}2 : {}3;
3.  cons Nat : {}3;
4.  cons Nat : Set;
5.  cons Nat : Set;
    {}4
6.  cons Nat : Set;
    cons {}5 : {}6;
7.  cons Nat : Set;
    cons Zero : {}6;
8.  cons Nat : Set;
    cons Zero : Nat;
9.  cons Nat : Set;
    cons Zero : Nat;
    {}7
10. cons Nat : Set;
    cons Zero : Nat;
    cons {}8 : {}9;
11. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : {}9;
12. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : {}10 ->
    {}11;
13. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    {}11;
14. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
15. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    {}12
```

```
16. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    let {}13 : {}14 =
    {}15;
17. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    let Two : {}14 =
    {}15;
18. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    let Two : Nat =
    {}15;
19. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    let Two : Nat =
    (Succ {}16);
20. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    let Two : Nat =
    (Succ (Succ {}17));
21. cons Nat : Set;
    cons Zero : Nat;
    cons Succ : Nat ->
    Nat;
    let Two : Nat =
    (Succ (Succ Zero));
```

Fig 3.5: A step-by-step example of the process of creating a program

3.3 THE LANGUAGE

The editor is developed to handle one simple language. This language is an extension of Simply Typed Lambda Calculus [3] and labelled bracket notation [4].

3.3.1 SYNTAX

This language will form a program that consists of a list of declarations. Each declaration is either a 'let' or a 'cons' expression, and have the general shape shown in figure 3.6.

```
decl ::=   cons i : t;      Constant declaration
         | let i : t = e;   Let declaration
```

Fig. 3.6: syntax of declarations

where `i` is an unique name and the type `t` is either 'Set' or an expression that has type 'Set'. `e` is an expression that must have type `t`. The syntax for expressions is formed by the BNF in figure 3.7.

```
e ::=      i                Constant
         | (i [e])          Application
         | \ i : t . e      Abstraction
```

Fig. 3.7: syntax of expressions

Figure 3.8 shows the concrete syntax in a notation based on the one used by GF[9]

```
Program. Prg ::= [Decl]
DCons.   Decl ::= "cons" Ident ":" Type ";"
DLet.    Decl ::= "let" Ident ":" Type "=" Exp ";"
TId.     Type ::= Ident
TFun.    Type ::= Type "->" Type
TApp.    Type ::= "(" Ident [Exp] ")"
EId.     Exp ::= Ident
EApp.    Exp ::= "(" Ident [Exp] ")"
EAbs.    Exp ::= "(\" Ident ":" Type "." Exp)
Ident. Ident ::= MATCH /^[A-z_]{1}([A-z_]|[0-
9])*'*$/i
```

Fig. 3.8 Concrete syntax

[9] http://www.grammaticalframework.org/

3.3.2 Types

There are two ground types and one functional type. The ground types are 'Set' or an expression. The functional type is a binary function that takes two other types as arguments. the syntax of types are shown in figure 3.9.

```
t ::=       Set      Set
          | e        Expression
          | t -> t   Functional
```

Fig. 3.9: Syntax for types

The typing rules are shown in figure 3.10:

$$\frac{}{Set\ Type} \qquad \frac{e\ \in\ Set}{e\ Type} \qquad \frac{e_1\ Type \quad e_2\ Type}{e_1\ \rightarrow e_2\ Type}$$

$$\frac{i \in A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B \quad e_1 \in A_1 \ldots e_n \in A_n}{i\ e_1 \ldots e_n \in B} \qquad \frac{i \in A \quad e\ \in B}{\backslash i{:}\,A.\,e \in A \rightarrow B}$$

Fig. 3.10: Typing rules

3.3.3 Program example

Figure 3.11 shows an example of a program that can be created in this editor.

```
cons Nat : Set;
cons Zero : Nat;
cons Succ : Nat -> Nat;
let Two : Nat = (Succ (Succ Zero));
cons List : Set -> Set;
cons Nil : List Nat;
cons Cons' : Nat -> List Nat -> Nat;
let Nats : List Nat =
Cons' Zero (Cons' (Succ Zero) Nil);
let AddZerotoList : List Nat -> List Nat =
(\a : (List Nat). (Cons' Zero a));
```

Fig. 3.11: An example program

3.4 Data storage

When developing programs the most intuitive way to store the programs would be to store them as files. This is not the way it is done in this application. Because all files would be stored on the server and only be presented to the user via the editor, these files would never be visible to the user.

A MySQL[10] database was chosen to store the programs instead. It was convenient to have the programs and the authentication information at the same place. A database also provides easy functionality for searching for, extracting, overwriting and inserting information.

The structure of the tables in the database is shown in figure 3.12.

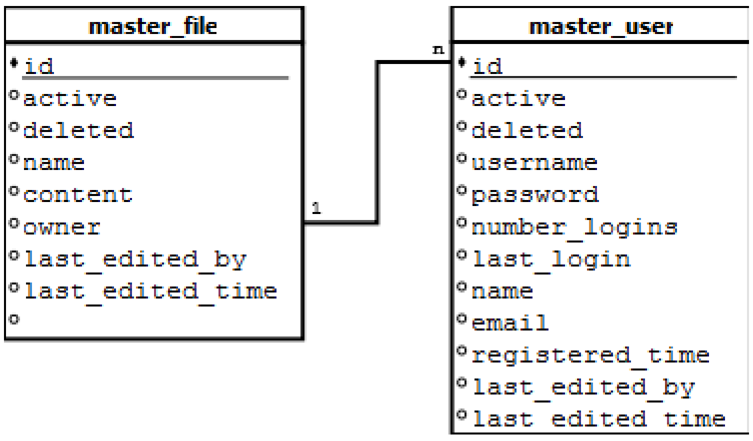

Fig 3.12: Structure of database tables

When the application stores a program, the history is stored rather than the content of the program. This is made to make the saving and loading much easier. The application logs all successful commands and when saving, this information is converted into a string. This string is constructed in such a way that it makes it very easy to retrieve every command in the right order.

3.5 File Structure

The application consists of a number of files, which are dependent on each other in one of the following ways:

- HTML frameset
- HTML script source
- AJAX request
- PHP require

These dependencies can also be seen in figure 3.13.

[10] http://www.mysql.com/

All PHP files in the application should in some way include the "pre.php" file. This file contains some code and operations that initialize the usage of PHP like connections to the database and starting a session for temporary data storage. It also contains functions for checking the authentication etc.

If a file requires authentication, this check is done right after the initialization is finished. If the login is successful the file will continue to execute, in other cases the file will stop executing and instead present the login form.

A web server will by default return the file named “index”, if no special file was requested. So the starting point for this application is the “index.php". This file contains a frameset with three frames: menu, status and editor.

The heart of the application is the editor.js file. This is where the abstract syntax tree and functions for manipulating it is contained. Files that are used via an Ajax request are in the Ajax folder. These files will only print its result as a Json object.

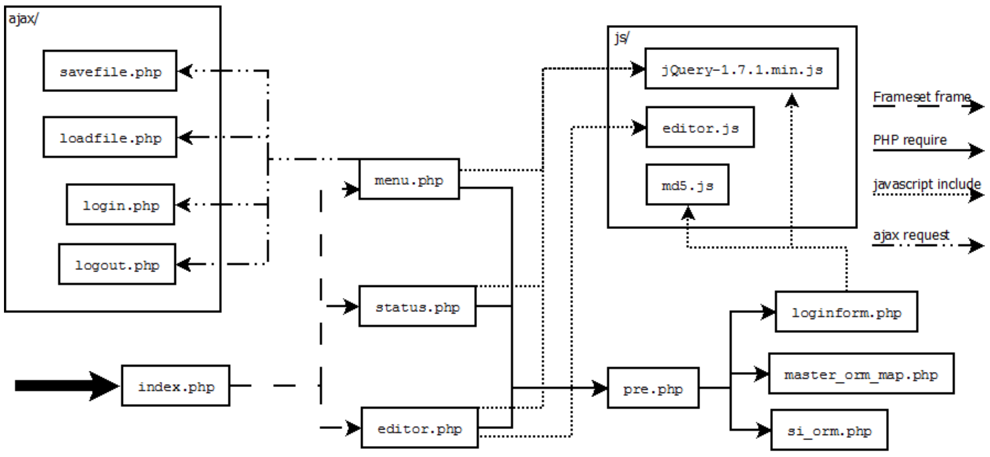

Fig. 3.13: File dependencies

3.6 Internal representation

3.6.1 Q-node

A Q-node is a JavaScript object that describes a node of the abstract syntax tree. This object is simply a data holder and does not contain any functions for manipulation. It has the name Q-node rather than just 'node' to not confuse it with nodes in the HTML DOM Node tree. The definition of this object is shown in figure 3.14.

```
var Q = function (number, nodeType) {
         this.number = number;
         this.numberPH = null;
         this.nodeType = nodeType;
         this.type = false;
         this.down = new Array();
         this.up = null;
}
```

Fig. 3.14: JavaScript definition of a Q-node

Each Q-node must have a nodeType. This describes how the node is used when, for example, printing it. The values this property can take are deduced from the concrete syntax. All possibilities for this are shown in figure 3.15.

The "number" property is a unique number that is used to select and identify each node. These numbers are only used internally, and the user will never see them. What the user does see is the "numberPH". This is visible beside every placeholder. This value is incremented, so for each placeholder that is created, this will be one higher.

All of nodes arguments (children or sub nodes) are stored in the "down" property. The value of this is an array, for which the elements are dependent on the value of the "nodeType" property. Each element in this array does not contain the Q-node under it, but rather the number of that Q-node.

The "type" property expresses what type that node has. This puts a constraint on what expression it can be replaced with.

nodeType	**down[0]**	**down[1]**	**down[2]**
DCons	Ident	Type	
DLet	Ident	Type	Exp
TId	Ident		
TFun	Type	Type	
TApp	Ident	Exp	
EId	Ident		
EApp	Ident	Exp	
EAbs	Ident	Type	Exp
Ident	String		
PH			

Fig 3.15: nodeTypes and their argument.

3.6.2 Qs

This is a list of Q-nodes that describes the complete program that is currently active in the editor as an adjacency list. Each Q-node object in this list will have the same number as its index in the list. In this way, it will be very easy to find a Q-node given a number.

The simple program in figure 3.16 that defines the natural numbers will have the Qs list shown in figure 3.17

```
Cons Nat : Set;
Cons Zero : Nat;
Cons Succ : Nat -> Nat;
```

Fig. 3.16: A program defining the natural numbers

Qlist	NodeType	Down
0:	Ident	["Set"]
1:	Cons	[1 2]
2:	Ident	["Nat"]
3:	Type	[0]
4:	Cons	[5 6]
5:	Ident	["Zero"]
6:	Type	[2]
7:	Con	[8 9]
8:	Ident	["Succ"]
9:	Type	[10 11]
10:	Type	[2]
11:	Type	[2]

Fig. 3.17: The Qs-list of the program defined in figure. 3.16

3.7.3 Contexts

The application has two types of contexts. One global context and one list of local contexts. The global context keeps track of all variables and their type for all declaration nodes in the program. This type of variable can be accessed from everywhere in the program. The local context contains the same information, but only for abstraction nodes. These variables can only be accessed in a sub node of that abstraction node.

3.8 HTML REPRESENTATION

The abstract syntax tree is presented to the user encoded into a HTML string. A placeholder node will be translated to the HTML code in figure 3.18.

```
<span class="Q ph" id="Q_[N]">{
<span contenteditable="true" class="ph_edit"
id="Qedit_[N]">
</span>
}[NPH]</span>
```

Fig.3.18: HTML representation of a placeholder

where `[N]` and `[NPH]` correspond to the “number” and “numberPH” property respectively. The inner span element has the attribute `contenteditabe="true"`, which makes it possible for the user to edit the text inside this element. This means that the user can only edit the text inside these placeholders. The `id` and `class` attributes described a unique number just this element, and also what classes this belongs to. This information is used when pretty-printing and selecting nodes in the editor.

All other nodes except the placeholder have the same general structure, so they will be translated to the HTML code in figure 3.19.

```
<span class="Q" id="Q_[N]">[…]</span>
```

Fig. 3.19: HTML representation of a node

where `[N]` correspond to the "number" property of the node. The text inside this span, where the `[...]` is, is dependent on what nodeType that node has. For example, a node with nodeType "DCons" will be translated into the HTML code in figure 3.20.

```
Cons <b>[...]</b> : <i>[...]</i>;
```

Fig. 3.20: HTML representation of a “DCons” node

where both [...] are a recursive call to the pretty print method for each of the two arguments defined in the down property.

3.9 PARSING AND TYPE CHECKING

The parser for this application only works on one word at the time, which is one placeholder at the time.

Figure 3.21 is a somewhat simplified diagram of the parsing process. In this figure, `Q` is the placeholder being parsed, and `input` is the string. All the `createX` methods on the right side of the figure convert a placeholder into a the given Q-node. This function also handles all its sub nodes. For example the `createDCons` method creates two new placeholders, one with type `Ident` and one with `Type`.

The typecheck-function can have three different results. If the result is a Boolean value, this describes whether the type is correct or not. In other cases, an expression will be returned. This will mean that the type will be correct if the input is applied to the returned expression.

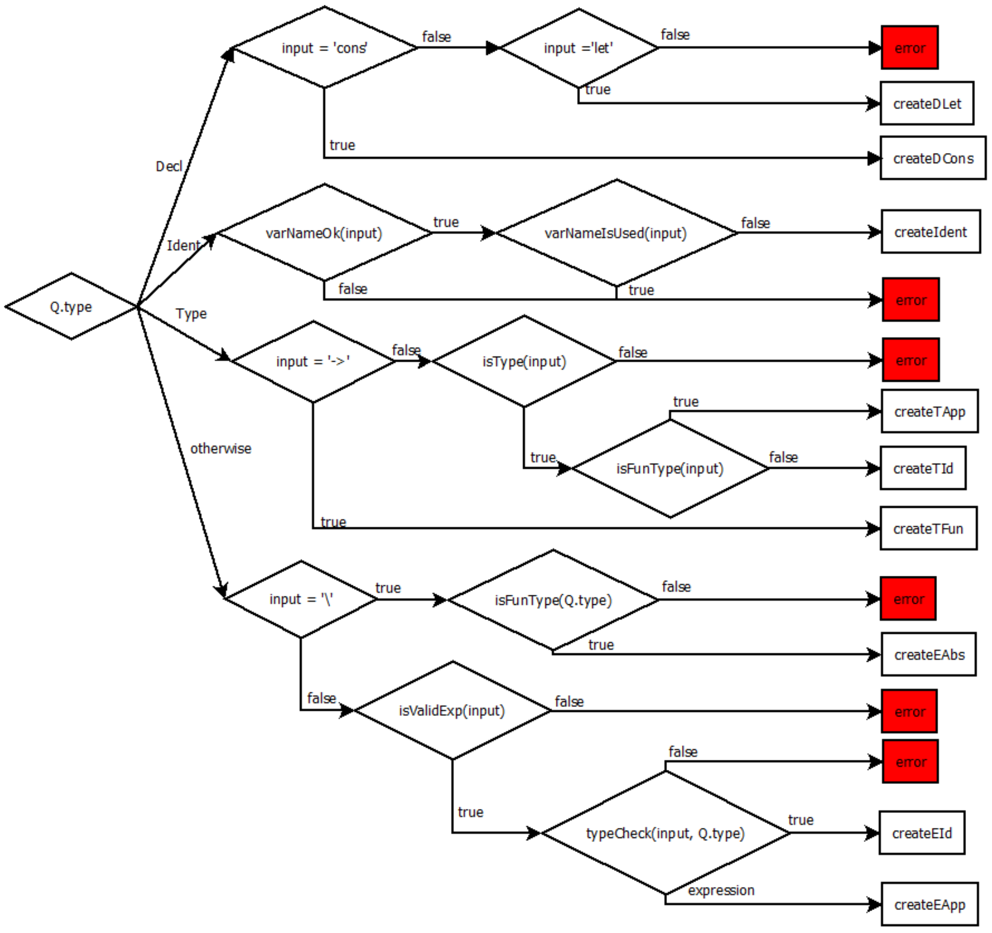

Fig. 3.21: Flow diagram of the parsing process

3.10 Server side

The server main functionality is to store and handle the authentication information and the programs. The login information is stored in a session, which is a temporary storage that lasts as long as the web browser is open.

4 DISCUSSION

4.1 SECURITY

The security of the application was not the most prioritized during this project. Some basic functionality has been added to ensure some level of security, but if a hacker really wanted to make a mess, it wouldn't be that hard. The login system encrypts each password with a MD5-function before sending it to the server for verification. MD5 is not considered to be secure anymore, but it still prevents a hacker to see the password as clear text if he is using a packet sniffing software. When storing the password in the database, it is further encrypted using SHA1-function and a salt. A salt is a randomly generated string that prevents the hacker to find the password using a dictionary search. In the end, the password is stored in the database like this:

```
salt + SHA1(salt + MD5(PASSWORD))
```

When saving and loading a program, the data is sent with clear text to the server and no security can be ensured.

4.2 FURTHER DEVELOPMENT

As the program is now, all functionality that was intended does exist. But there are some things that could make the program better in terms of usability.

4.2.1 COMPLETE PARSER

The parser can only take one word at the time and parse this according to the type. This can be improved to make the application be able to parse an entire expression at one time. This feature would make it possible to copy programs as a string from a text document into the editor, and make the editor understand it.

4.2.2 BETTER TYPE-CHECKER

The type checker works the same way as the parser do, on one expression at the time. In order to make the parser be able to parse a whole expression, the type-checker also needs to be improved to handle many expressions at once.

4.2.3 Define Concrete syntax

One of the extra goals was to create the application is such a way that the concrete syntax could be defined by the user, and in this way be able to present the program to the user in other ways. It would for example be possible to define an infix notation for some constants so the expression

```
+ 2 3
```

could instead be written as

```
2 + 3
```

which is much more readable and recognizable to the human.

4.2.4 Dependent functional types

The second extra goal was to introduce a more advanced type system. This would make the application be able to express much more generic and complex programs.

Figure 4.1 shows an example that could be expressed with dependent functional types.

```
Vec : Nat -> Set
Nil : (Vec Zero)
Cons : (N <- Nat) -> Nat -> Vec N -> Vec (Succ N)
```

Fig. 4.1: A possible way of defining a vector

4.2.5 Autocomplete

Many editors that has an autocomplete functionality. This can be very useful when the programs get bigger, and number of variable names that is used grows.

Autocomplete functionality in this editor would need to check the typing information and then suggest all expressions that can follow the rules.

4.2.7 KNOWN BUGS

Here is a list of known bugs that was discovered, but not fixed because of time and complexity reasons.

- It is possible to construct a program that does not follow the type rules. This is because the type-checker does not check the whole program when something has been changed. When removing a node with type 'type', all nodes that are dependent on this will be erroneous, and this will not be visible to the user. A way of solving this could be to implement the improvement of the type-checker in section 4.2.2 and every time a node was deleted, the type-checker would check the whole program and highlight all erroneous nodes, and force the user to update these so that no type-rules are violated.

5 References

[1] Philip Wadler and Robert Bruce Findler, Well-typed programs can't be blamed, ICFP, 2008

[2] PHP Usage stats, http://php.net/usage.php, 11 May 2012

[3] Benjamin C. Pierce: Types and Programming Languages, page 51, the MIT Press, 2002.

[4] Labelled brackets, http://chars.lin.oakland.edu/General/LabeledBrackets.pdf, 25 May 2012