Nearly Instantaneous Slide Show Expressions

May 2012

By: Group SW403F12

AALBORG UNIVERSITET
Student Report

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://cs.aau.dk

Title:

Nearly Instantaneous Slide Show Expressions

Theme:

Design, Definition and Implementation of Programming Languages

Project period:
P4, Spring Term 2012

Project group:
SW403F12

Participants:
Ali Mansour Nazim
Jakob Lynge Albertsen
Johan Sørensen
Jonathan Bernstorff Nielsen
Tommy Knudsen

Supervisor:
Benjamin Bjerre Krogh

Circulation: 7

Page count: 93

Appendix count and type: 5, Context free grammar of NISSE, "kfG Edit" settings, Slideshow example, Font Type, Front Page Code

Finished on May 25^{th} 2012

Synopsis:

In this project, a programming language for creating slideshows has been developed.
Furthermore a compiler for the language has been developed, which makes it possible to compile the written code into an HTML-file.
The development of the language is described throughout the report.
The main focus of the project was to make a programming language that could make it possible to create slideshows without using a pointing device.
Another requirement for the language was that the language should be faster to creating slideshows in than already existing languages, such as LaTeX Beamer.

The developed slideshow programming language fulfils the major language requirements described in the "requirements" section 4.1.

Signatures:

Ali Mansour Nazim

Jakob Lynge Albertsen

Johan Sørensen

Jonathan Bernstorff Nielsen

Tommy Knudsen

Contents

Chapter 1

Introduction

Many people are using slideshows, in various forms, at a daily basis. There are a lot of different methods for creating a slideshow; there are programs that make it possible for the user to create slideshows relatively simple, and there are programming languages that are designed for the purpose of creating a slideshow. The slideshow solutions that will be discussed briefly are; Apple Keynote and Microsoft Office PowerPoint and more detailed about LaTeX Beamer and the language of this report NISSE (Nearly Instantaneous Slide Show Expressions).

A similarity that Keynote and PowerPoint have in common is that they are both pointing device[1] based slideshow programs, which means that without using a pointing device you cannot use them properly, whereas LaTeX Beamer is not pointing device dependent. With LaTeX Beamer not being a pointing device dependent slideshow programming language makes it a rival to the NISSE language. A functionality that LaTeX Beamer lacks out on is the option for the user to use it properly on a mobile device[2]. With "properly" it is meant that you can write it on a mobile device, but LaTeX is a complex programming language with a lot of packages, which also applies to LaTeX Beamer so the complexity is the same.
A goal with NISSE is to create a slideshow programming language, where the user does not have to remember a lot of different packages to gain certain functionality, or have to search on the Internet how to get some specific functionality. Then when the user wants to compile it, he can send it to a server that automatically compiles it for him and returns the compiled slideshow in a single .HTML-file, which makes the user able to see the slideshow.

After a walkthrough of the compiler, there will be a conclusion to summarize the report in general. Following this, the "Further Development" section presents ideas and functionalities which could be implemented in the compiler. Included with the report is a CD containing the developed compiler, the source code of

[1] A pointing device is defined as a mouse, trackball, touchpad or pointing stick.
[2] A mobile device is defined as a laptop, tablet, smartphone, etc.

the developed compiler and the report. The front page is a slide generated by NISSE with it's code viewable in appendix A.5

1.1 Initiating Problem

It is hard to use a pointing device to create slideshows while using a mobile device. To this extent, a more mobile slideshow language will be created, which will be a great way to create slideshows while "on the go".

How can a language remove the need for a pointing device when creating slideshows?

Chapter 2

Problem Formulation

How can you make a better alternative to LaTeX Beamer?
How can you create a programming language in which a user can make a slideshow presentation without the need for a pointing device, and not have to think about the layout of single slides, but only define the general layout?
Furthermore, how can you display the presentation in a way, such that it will look the same on all devices, while still being usable on these devices?
How can you make a suitable domain specific language with weight on consistency and simplicity, while enabling the user to focus on the content rather than the layout?

The following questions provide a limitation for the scope of the project:

Which platform should be used for presentation?
Which layout decisions does the user need?
What are the language limitations?
What can be expressed in the language?

Part I

Analysis

Chapter 3

Known Slideshow Applications

Apple Keynote and Microsoft PowerPoint are slideshow applications which use drag-and-drop functions, which is why these will not be focussed on further, because this approach to create slideshows does not fulfill the requirements of being a non-pointing device based application, as specified in the problem formulation 2 of this report. The main focus will be on the differences between LaTeX Beamer and the slideshow programming language, NISSE, developed in this project.

3.1 LaTeX Beamer

An example of LaTeX Beamer is listed in listing 3.1.

Listing 3.1: Beamer example

```
1  Main_document.tex
2  \begin{document}
3  \include{Slide_document.tex}
4  \end{document}
5
6  Slide_document.tex
7  \frame {
8    \frametitle{Welcome to this course}
9
10 This course will contain information about how you \↩
       underline{underline} things, and how you do other \↩
       textit{weight stuff} on sentences. \\
11 \textbf{{Like this}} \\ }
12 }
```

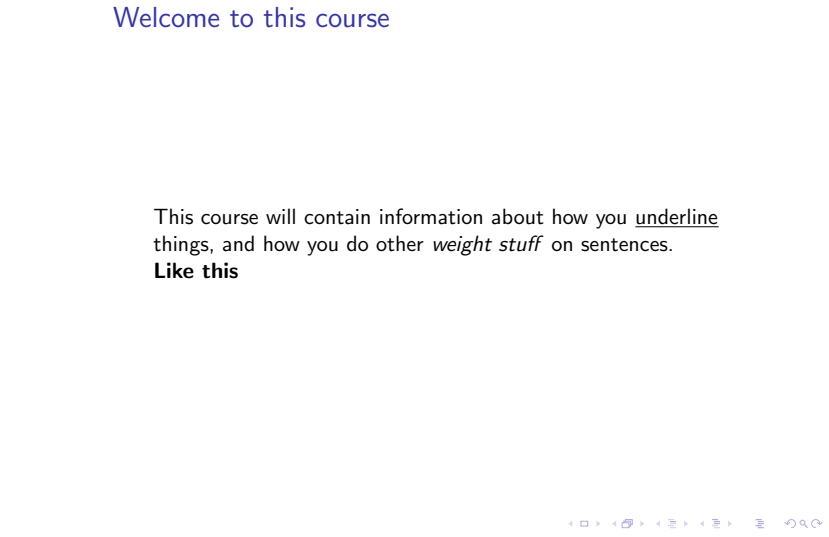

Figure 3.1: LaTeX Beamer output

The example seen in listing 3.1, is a LaTeX Beamer-code example for expressing the output in figure 3.1. The LaTeX Beamer-code seen in listing 3.1 is only to express the output shown. To make a slideshow using LaTeX Beamer you have to set up a main document. In this document, all settings about; theme, colours, inputs (other files), etc., for the slideshow is set up. LaTeX editors (e.g. TexMaker), only generates a very small amount of the main document, which leaves a lot of setup for the user, if additional settings is wanted. If only a general slideshow is required, the main document will not need much work. A general slideshow is without colours, themes or the need for additional packages. Compared to the LaTeX Beamer-code the developed language should be made more compact to make slides faster to express.

A test between LaTeX Beamer and the developed language will be made to determine which and why the one language is better suited for expressing slideshows than the other.

Chapter 4

Language Design

This section has been based on the book "Concepts of Programming Languages" [1]. The design of the language has been made using the language criteria in table 4.1 horizontal. These criteria can be categorised into three primary categories; *readability*, *writability* and *reliability*. These overall categories have a number of secondary categories, which will be used for specifying the language design. The secondary categories that affect the overall categories can be seen in table 4.1 vertical.

Characteristics	*Criteria*		
	Readability	Writability	Reliability
Simplicity	X	X	X
Orthogonality	X	X	X
Data types	X	X	X
Syntax design	X	X	X
Support for abstraction		X	X
Expressivity		X	X
Type checking			X
Exception handling			X
Restricted aliasing			X

Table 4.1: Language evaluation criteria from the book "Concepts of Programming Languages"[1]

Writability
Writability is an important criterion in designing a programming language. It refers to how easy it is to write programs using the language. In the developed programming language it should be easy to write and express slideshows. A programming language is usually easier to use, if it shares some characteristics with popular programming languages.
The writability has been deemed "very important", because the developed language is a slideshow programming language, and considering that a slideshow

is only made once writability is very important.

Portability
Portability is also known as the ease with which programs can be implemented on another platform than the one they were originally developed for.
Software is portable when it is less expensive to port a program to a new platform than writing the program from scratch. The lower the cost of porting software the more portable the software is.

In the programming language of this project, the portability has been deemed "very important", because it is very important that the programming language is made in a way where it is working "out of the box", regardless of the operating system the programmer is using. Furthermore, the group who is to develop the programming language in question are using different operation systems, such as; Ubuntu, Mac OS X, and Windows, which makes a good foundation for checking whether the programming language is working on these three operating systems, which in turn gives the developers an idea of the portability of the language. Furthermore, the slideshows created using the language should be able to be shown on different monitor resolutions.

Reliability
Reliability and correctness is an important criteria in designing a programming language. A language is reliable if it performs correctly and without errors under all conditions, in this programming language meaning that the presentation behaves exactly as the user wants it to. The ability for the language to perform correctly under all conditions, has been deemed "important", because the user should be able to rely on the language in a way that makes him know what it will do, using the language.

Orthogonality
Orthogonality can be seen as a consistent set of rules for combining primitive constructions. Such constructions as underlined and bold text, which both are primitive constructions.
An example of orthogonality, from the programming language, is text formatting, which means that changes can be included not only in normal text, but also in figure text, title text, etc.
A high level of orthogonality makes it possible to express a lot with a few primitives.
A low level of orthogonality results in that the user has to use many primitives to express a lot, e.g., format text in different ways.

In the programming language of this project, the orthogonality has been deemed "important", because it is important that there are different ways to express text formatting - making it simpler and easier to use.

Expressivity
Expressivity is an expression covering the ease with which different operators in a programming language is designed. An example of this is using $count++$ instead of $count = count + 1$. Expressivity can be seen as an extension to writability, in that it can make it easier to express statements in a programming language. In the developed programming language, expressivity has been deemed "important" because there has been a lot of focus on making the language easy and convenient to write and express slideshows in.

Readability
Readability is an important criterion in designing a programming language. The definition of readability is that the language, which is being designed, should be easy to read and understand. When a programmer is adapting to a new language, the most difficult things to remember is usually the context of the language and the name of the data types. Creating a syntax, that in some way is similar to popular languages such as Java or C like languages, which would make the language more readable and understandable.
Readability of the developed programming language has been deemed "less important", because it is not as important as the writability or the reliability, though it still has to be possible to read code written in the language.

Cost
When talking about the cost of making a programming language, there are some different aspects to consider, these are:

1. The cost of training the programmers that is going to use the language, which is a function of simplicity, orthogonality and the experience of the programmers.
2. The cost of writing programs in the language, which is a function of writability.
3. The cost of implementing the programming language.
4. The cost of, potential, poor reliability. If, for example, a system is unreliable, it can be costly to increase its reliability.
5. The cost of compiling programs in the language.

In the programming language of this project, the cost has been deemed "less important" because;

1. It is not going to be taught to other programmers than the developers, because it is a proof of concept.
2. The only slideshows that are going to be written in the language are written by the developers themselves.
3. The language is not meant to be developed for bigger corporations, which is why the implementation of the language is held to a minimum.

4. Even though the reliability is important, the potential cost of increasing reliability would not be big because the language is relatively small.

5. The performance of the compiler is not the main focus, but it is important to make sure that there will not be unnecessary waiting for the users of the programming language.

In table 4.2 the language design criteria that has been prioritized for the programming language design for this project are listed.

	Very important	Important	Less important	Irrelevant
Writability	X			
Portability	X			
Reliability		X		
Orthogonality		X		
Expressivity		X		
Readability			X	
Cost			X	

Table 4.2: Language evaluation criteria defined for the programming language to be developed.

4.1 Requirements

The following requirements are set for the language to consider it as a complete language:

Capabilities

- It must be possible to make bullet points and numerations in the language.
- It must be possible to import images from the Internet in the language.
- The user should be able to change font- family, colour, size and line height.
- The user should be able to make a transition between each slide.

Error handling

- The compiler should be able to tell the user where an error has occurred and what the error is.

Test

- The language should be easy to write in, determined by the test:

  - The experienced user of the language should be able to make five slides, with decent content, using standard settings, within ten minutes, where the content is prewritten.

Limitation

It is estimated that it can become a source of distraction if people have too many options in the language, which can lead to a decrease in slides made per minute.

- Tables will not be implemented in the language due to the features complexity.
- Animation of text in the slideshow is not a necessity for a slideshow, which is why the feature will not be implemented.

Chapter 5

Lexing and Parsing

5.1 Generating Tools

In this section a number of lexer and parser generators will be presented, in order to determine the best generator for the project.
The lexer and parser can be made manually or generated automatically with a number of different generators. The advantage of having the lexer and parser generated by a generator is that changes in the CFG (Context Free Grammar) can easily be implemented and checked compared to doing this manually. This will be useful when expanding the possibilities in the language, which is why a generator will be used to generate the lexer and parser.

5.1.1 SableCC

SableCC is a lexer and parser generator. As its input it takes an extended EBNF (Extended Backus–Naur Form) grammar. In order for the grammar to be accepted, the grammar has to be LALR(1), which is the kind of parser SableCC makes. A feature of SableCC is that it can separate the generated code, from the developer's code, which makes it easy to update the compiler, as code does not have to be moved. SableCC also provides a visitor pattern to traverse the parse tree that the parser generates; this makes it easier to make the semantic analyser and the code generator.
A SableCC file consists of six optional sections; `package`, `helpers`, `states`, `tokens`, `ignored tokens` and `productions`.

- `Package` indicates what package the generated files should be made using.

- `Helpers` are either character sets or regular expressions denoted by an identifier. Helpers can only be used in `Tokens`.

- `States` are used to switch between states.

- `Tokens` are defined much like `Helpers`. The lexer will return the longest matching token or the token listed first in tokens if two or more tokens of the same length, are matched.

- `Ignored Tokens` are tokens that are not returned by the lexer.

Finally, `Productions` are the normal production rules of the grammar, although an identifier has to be given for every alternative to a production rule, and an identifier has to be given when more than one occurrence of a production rule or token is in a given alternative.

5.1.2 JavaCC

JavaCC is a lexer and parser generator. As its input it takes an EBNF grammar. However it is not in the form of a normal EBNF. It is in a special form derived from the Java syntax that looks a little like a Java method definition. An example of token definitions and production rules is given in listing 5.1

Listing 5.1: JavaCC token and production specification

```
1  TOKEN : { < PLUS : '+' > }
2  TOKEN : { < NUMBER : (['0'-'9'])+ > }
3  void Start() :
4  {
5    <NUMBER>
6    (
7      <PLUS>
8      <NUMBER>
9    )*
10   <EOF>
11 }
```

The example shows that the form is far from the original form of EBNF, but it is still readable as the production is named `Start` and the rule is that a `NUMBER` token is followed by none or more `PLUS NUMBER` token sequences. In the language specification JavaCC can also take some amount of action code, such that part of the code generator is also specified there.

Conclusion

SableCC has been chosen as the lexer and parser generator, because it takes an extended EBNF grammar, which is easy to translate to, from NISSE's grammar

which is written in EBNF. SableCC also includes a visitor pattern, which makes it much easier to not get slowed down by not having to write the visitor pattern for all of the production rules of this project. A SableCC specification file is also simple and easy to get started with.

5.2 Parser Strategy

There are two ways of parsing an abstract syntax tree, which is seen below:

Top Down

Top down derives the parse tree from the left. This means that a nonterminal cannot have two derivations with the same start terminal or nonterminal, e.g., $A \rightarrow BC|BD$. This way the parser does not know which derivation to take, because both of them start with the same nonterminal. This can be avoided by using a `bottom up` method.

Bottom Up

Bottom up derives the parse tree from the right, which makes the expression: $A \rightarrow BC|BD$ legal for a bottom up parser, because it looks at C and D first and then decides which derivation to pick.

Conclusion

A Bottom Up (LALR (Look Ahead Left-scan, Rightmost derivation in reverse)) derivation of the parser will be used in this language, because of the use of SableCC, which makes a LALR-parser, which can be seen in section 5.1.1.
If the lexer and parser were to be made manually, an LL parser would be preferred, because it can be visualized more easily, compared to an LR parser.

Part II

Solution

Introduction

Constructing a compiler is split into four categories; lexer, parser, semantic analyser and code generator, as shown in figure 5.1. The user inputs some code, to the compiler that he would like to have compiled, which is first met by the lexer. The lexer's job is to make a token for each of the characters that is provided in the code. If a character is not recognized, this stage will fail.

The parser asks the lexer for a token, which is then handled by the parser. This is being repeated until there are no tokens left. The parser's main function is to check the syntax of the source code, to check whether it contains any invalid sign of formatting. The parser is doing this by creating a parse tree, which it uses to check that all tokens are given in the right order. If the parser is able to create more than one AST (Abstract Syntax tree), the code could behave differently each time it is compiled. There are also a number of ways for a parser to build an AST, which was discussed in section 5.2. If the parser fails to build an AST, this stage will fail.

The AST is then given to the semantic analyser, which visits each node. The semantic analyser applies code from the developer to each node, in order to check if the semantic in the usercode is valid, and if not throw an exception. The semantic analyser also builds a symbol table, in order to keep track of variables.
The last phase of the compiler is the code generator. The code generator's main function is to translate the source code into the target code.

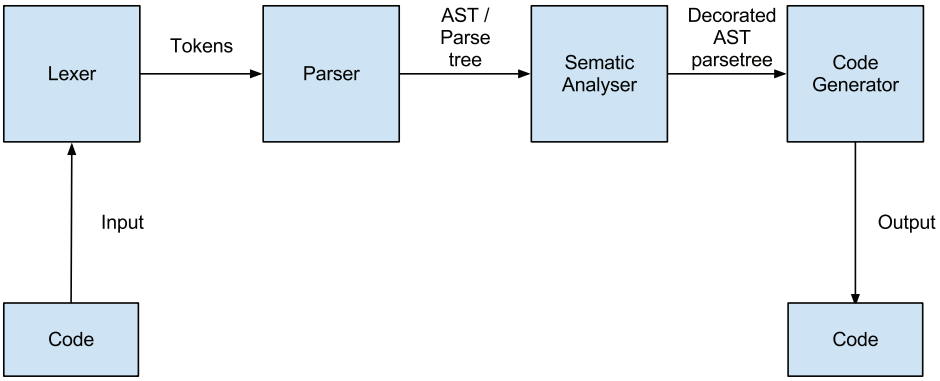

Figure 5.1: Compiler construction

Chapter 6

Syntax

The requirements for the language is defined in section 4.1. Keywords are in the language to give structure and flexibility. These have different outcomes, although some are similar. Expressions are used for the structuring of slides in a slideshow, and used for text formatting or image placement. To give a higher level of readability, keywords and expressions must have meaningful names according to their use and effect. Secondly, it has to be defined in which order these are allowed, and what they mean in that particular order.

6.1 Expressions and Keywords

Expressions are the words with the @-prefix and may not be used otherwise, because the @-prefix should state that an event is about to happen.

6.1.1 @begin

The expression `@begin` determines when a slide begins. This has a `transition` parameter separated by a pipe character, after the pipe is the name of the slide, but the name of the slide has no effect at that moment. An example of this is shown in listing 6.1.

Listing 6.1: Generic `begin` expression example

```
1 @begin{transition|slidename}
```

Where `transition` is the form of transition to the slide. Transitions will be explained in section 6.1.3

6.1.2 @end

The expression `@end` is used to determine the end of a slide, by writing the line in listing 6.2.

Listing 6.2: Generic `end` expression example

```
1 @end{slidename}
```

Listing 6.3 shows some of the basic elements in the language.

Listing 6.3: Simpel slide

```
1 Input:
2 @begin{slidename}
3    Hello World
4 @end{slidename}
5
6 Output: Hello World
```

Which states that a slide begins, and that the slide contains the text "Hello World", and then the slide ends.

6.1.3 Transition Keyword

Each slide can contain a transition, transitions are set as keywords, which is `fade`, `swipe`, `scale`, `rotatescale`.
The transition is inserted before the pipe in an `@begin` expression. The pipe is inserted only when a transition is wanted, as shown in listing 6.4.

Listing 6.4: Hello World with transition

```
1 Input:
2 @begin{fade|slidename}
3    Hello World
4 @end{slidename}
5
6 Output: Hello World
```

6.1.4 @setting

This expression sets a new setting for a specific type of font setting for a specific type of text. A generic example of this is shown in listing 6.5.

Listing 6.5: Generic `setting` expression example

```
1 @setting{FontChanges:Value | Type}
```

Where `FontChanges` is what kind of font setting that is changed, which can be `@font_family`, `@font_color`, `@font_size`, `@font_weight`. A more detailed description of these expressions can be read in appendix A.4. `Value`, is the value that the font change is set to.
`Type` is referring to which font class it will change, these font classes are set as keywords and is `title`, `subtitle`, `global`, `text`, `image`, `url`

Global Keyword

The `global` keyword makes it possible to apply settings globally, which overwrites the settings of normal text, titles, subtitles, image descriptions and URLs. Globals work from the place in the slide where it is applied or from the place outside a slide.

An example with the use of `@setting` inside a slide is shown in listing 6.6

Listing 6.6: Hello World with setting

```
1 Input:
2 @begin{fade|slidename}
3    @setting{@font_color:blue|text}
4    Hello World
5 @end{slidename}
6
7 Output: Hello World
```

6.1.5 @apply

The expression `apply` changes the font for the specific text.

Listing 6.7: Generic `apply` expression example

```
1 @apply{FontChanges:Value | Text}
```

Where `FontChanges` is what kind of font type that will be changed. These can be `@font_color`, `@font_family`, `@font_size` and `@font_lineheight`

`Value` is the value that it is set to. `Text` refers to the text which is inserted in the slides with the font changes that is made on the other side of the pipe. The `@apply` expression only changes the font for the specific text which is in the expression. An example with the use of `@apply` is shown in listing 6.8

Listing 6.8: Hello World with apply

```
@begin{fade|slidename}
   @setting{@font_color:blue|text}
   Hello World
   @apply{@font_size:70|Amazing world}
   Hello you
@end{slidename}

Output: Hello World
Amazing world
Hello you
```

6.1.6 @url

The expression `@url` along with `@apply` makes it possible for the user to include URLs, as hyperlinks, that redirects the user to the specific URL. An example of the use of URL links is shown in listing 6.9

Listing 6.9: Hello World with an URL

```
@begin{fade|slidename}
   @setting{@font_color:blue|text}
   Hello World
   @apply{@font_size:70|Amazing world}
   Hello you
   @apply{@url:http://www.somelink.com/ |This URL}
@end{slidename}

Output: Hello World
Amazing world
Hello you
This URL
```

6.1.7 @i @u @b

These expressions are similar to each other, in that they are all formatting text, as seen below

`@u` creates underlined text.
`@i` creates italic text.
`@b` creates bold text.
The full expression looks like listing 6.10

Listing 6.10: Generic `font weight` expression example

```
1 @b{ Text }
```

Where `Text` is the text that is shown in the slide, with the proper change of font weight, in this case to bold. In this case a pipe is not needed because there is nothing to put there. It is possible to make font changes, using `@u` / `@i` / `@b`, as in listing 6.11

Listing 6.11: Generic `font weight` expression example

```
1 @b{ FontChanges:Value | Text }
```

Which looks familiar to the `@apply` expression, and the functionality is basically the same, even though the text is set as any of the font weights `@u` / `@i` / `@b` to begin with. The {`@u` / `@i` / `@b` makes it much faster to make a weight on a text instead of writing: `@apply{ @font\_weight:bold | Text}` each time to make bold text. The use of weight expressions is shown in listing 6.12

Listing 6.12: Hello World with font weight

```
1  @begin{fade|slidename}
2     @setting{@font_color:blue|text}
3     Hello World
4     @apply{@font_size:70|Amazing world}
5     Hello you
6     @apply{@url:http://www.somelink.com/ |This URL}
7     @b{@font_color:red |This text is red and bold}
8  @end{slidename}
9
10 Output: Hello World
11 Amazing world
12 Hello you
13 This URL
14 This text is red and bold
```

6.1.8 @title

The expression `@title` makes a title for the slide. The title can be formatted as all other text, with the different font weights, font sizes, etc. The user can use the default settings for the title, regarding font family, font colour, font size, or define them himself inside the `@title` expression. An example of the use of `@title` is shown in listing 6.13

Listing 6.13: Hello World with title

```
@begin{fade|slidename}
    @setting{@font_color:blue|text}
    Hello World
    @apply{@font_size:70|Amazing world}
    Hello you
    @apply{@url:http://www.somelink.com/ |This URL}
    @b{@font_color:red |This text is red and bold}
    @title{@font_weight:underlined|This is an underlined title}
@end{slidename}

Output: Hello World
Amazing world
Hello you
This URL
This text is red and bold
This is an underlined title
```

6.1.9 @subtitle

The expression `@subtitle` makes a subtitle for the slide, per default with a smaller font size than title and with the color grey. The subtitle can be formatted as all other text, with the different font weights, font sizes, etc. The user can use the default settings for the subtitle, regarding font family, font colour, font size, or define them himself inside the `@subtitle` expression. An example of the use of `@subtitle` is shown in listing 6.14.

Listing 6.14: Hello World with subtitle

```
@begin{fade|slidename}
    @setting{@font_color:blue|text}
    Hello World
    @apply{@font_size:70|Amazing world}
    Hello you
    @apply{@url:http://www.somelink.com/ |This URL}
    @b{@font_color:red |This text is red and bold}
    @title{@font_weight:underlined|This is an underlined title}
    @subtitle{This is a subtitle}
@end{slidename}

Output: Hello World
Amazing world
Hello you
This URL
This text is red and bold
```

17 This is an underlined title
18 This is a subtitle

6.1.10 @image

The `@image` expression is used for importing images to the slideshow, either from the Internet of from the user's computer. It is discouraged to provide absolute paths for images, because this would only make the user's computer able to show the image upon compiling. Instead, it is encouraged to specify the name of the image, and place it in the same folder as the HTML-file. This is only the case if there are more contributors to the slideshow.
When importing an image from the Internet, the URL has to be provided, whereas when importing a locally stored image the file path to that specific image has to be provided, which can be seen in listing 6.15.

Listing 6.15: Generic `image` expression example

```
1 @image{@url:URL | Text }
```

`@url` is the expression indicating that a URL will be inserted, whereas **''URL''** is the actual URL address that has been inserted. When using a `@image` expression, `@url` has to be included, otherwise the compiler will output an error.
`Text` refers to the text which is under the image, also called the image description. The only thing that the user have to be aware of, is that the image have to be placed in the same directory as the compiler if it is to function properly. An example of the `@url` expression is listed in listing 6.16.

Listing 6.16: Hello World with an image

```
1  @begin{fade|slidename}
2     @setting{@font_color:blue|text}
3     Hello World
4     @apply{@font_size:70|Amazing world}
5     Hello you
6     @apply{@url:http://www.somelink.com/ |This URL}
7     @b{@font_color:red |This text is red and bold}
8     @title{@font_weight:underlined|This is an underlined ↩
          title}
9     @subtitle{This is a subtitle}
10    @image{@url:https://www.google.dk/images/srpr/logo3w.png ↩
          |This is Googles logo}
11 @end{slidename}
12
13 Output: Hello World
14 Amazing world
15 Hello you
```

```
16 This URL
17 This text is red and bold
18 This is an underlined title
19 This is a subtitle
20
```

Figure 6.1: This is googles logo

6.1.11 Bullet Points (*)

The * expression starts a bullet point, where the text after the asterisk will be the text after the bullet. A asterisk has to be set on each line that is going to be a bullet. Multilevel bullet points can be made using two asterisks. The maximum multilevel that can be made is five.
Bullet points must never have spaces before the asterisk, which makes the compiler think it is plain text. According to the rules specified for this language, the user must not have plain text before numerations.

A bullet list is made in listing 6.17.

Listing 6.17: Bullet point example.

```
1 Input:
2 @begin{slidename}
3 List of things to buy
4 * 2 x milk
5 * Bread
6 ** Light
7 * BKI coffee
8 @end{slidename}
9 Output:
```

List of things to buy

- 2 x milk
- Bread
  - Light
- BKI coffee

6.1.12 Numeration (#)

The `#` expression starts a numeration list, starting from one, and for each line the hash tag is used, the number is incremented. Multilevel numerations can be made using two or more `#`. The maximum multilevel that can be made is five. If the line after a hash tag line is not a hash tag line, the numeration list is ended and thus a new hash tag line would have its numeration starting from one.
Numeration must never have spaces before the #, which makes the compiler think it is plain text. According to the rules specified for this language, the user must not have plain text before numerations.

A numeration list is made in listing 6.18:

Listing 6.18: Numeration example

```
Input:
@begin{slidename}
Agenda
# Introduction
# Presentation
## Code examples
# Evaluation
@end{slidename}

Output:
```

Agenda

1. Introduction
2. Presentation
   (a) Code examples
3. Evaluation

Chapter 7

Lexer

In this chapter the requirements of the lexer for NISSE will be presented, along with a listing of the tokens for NISSE.

7.1 Requirements

The requirements for the lexer are:

- The lexer should be able to take any plain text file as input.
- The lexer should be able to recognize the input and make tokens according to the token list for NISSE.
- The lexer should be able to output meaningful error messages when it cannot match the input to any token.
- The lexer should be able to output tokens such that the parser can use them.

7.2 Token List

In order to create the syntax in chapter 6, tokens have to be identified for later use.

The tokens for the language is listed in listing 7.1.

Listing 7.1: Token List of NISSE in EBNF.

```
1  char            = ? [a-åA-Å]+ ? ;
2  digit           = ? [0-9]+ ? ;
3  underscore      = '_' ;
4  hyphen          = '-' ;
5  dotv1           = '.' ;
6  commav1         = ',' ;
7  space           = ' ' | '	';
8  atsign          = '@' ;
9  lcurly          = '{' ;
10 rcurly          = '}' ;
11 pipe            = '|' ;
12 fslash          = '/' ;
13 bslash          = '\' ;
14 colon           = ':' ;
15 scolon          = ';' ;
16 blist           = '*' ;
17 nlist           = '#' ;
18 percent         = '%' ;
19 exclamation     = '!' ;
20 eolv1           = \r | \n | \r\n ;
21 format_kwd      = '@u' | '@b' | '@i' | '@apply' | '@image' |↩
       '@title' | '@subtitle';
22 setting_kwd     = '@setting' ;
23 begin_kwd       = '@begin' ;
24 end_kwd         = '@end' ;
25 url             = '@url' ;
```

An example is the line `S = a {b}+ [ {c}+ ] ( a | b ) ;` that translates to `S = a`, followed by at least one `b`, followed by none or at least one `c`, followed by `a` or `b`. The special sequence is used for writing anything that is not directly an EBNF, defined in 7.1. An example is `char = ? a-åA-Å ? ;` in listing 7.1 where all letters from a to z is defined using both capital and non-capital.
The abnormal in this token list is `format{\_}kwd`, `setting{\_}kwd`, `begin{\_}↩kwd` and `end{\_}kwd`, (kwd is short for "keyword"). These keywords are explicit, because in the productions in which they occur they can only be on the left hand side of a left curly bracket. Their specific use is explained in chapter 6.

The reason each expression has its own token and not a general production, is to easily see the difference between the special tokens and the normal ones,

Usage	Notation
definition	=
termination	;
alternation	|
option	[...]
repetition	{ ... }
grouping	(...)
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?

Table 7.1: EBNF syntax

without the need to check that a given token in the syntax analyser is the right token.

`Char` is used whenever characters needs to be used in the code. `Char` is used both for plain text and identifiers.
`Digit` is used whenever numbers need to be used in the code.

7.2.1 NFA

Figure 7.1 is a NFA (Nondeterministic Finite Automaton) for tokens when a user inserts an `atsign`, which can become a lot of different tokens. The lexer will always take the token with the longest length, and if the input matches two token specifications of the same length it will choose the token that is written first in the token specifications.
The atsign can create a number of different tokens, shown in figure 7.1, these are; `formatkwd, settingblock, beginkwd, endkwd, url` and if the text does not match any of these, it is tokenized as an `atsign` and the text after the `atsign` is tokenized as a another token.

Regular expression

The regular expression for figure 7.1 is shown in listing 7.2

Listing 7.2: Regular expression

1 $@((i(\epsilon \cup mage)) \cup (u(\epsilon \cup rl)) \cup (b(\epsilon \cup egin)) \cup title \cup (s(ubtitle \cup etting)) \cup end \cup \epsilon)$

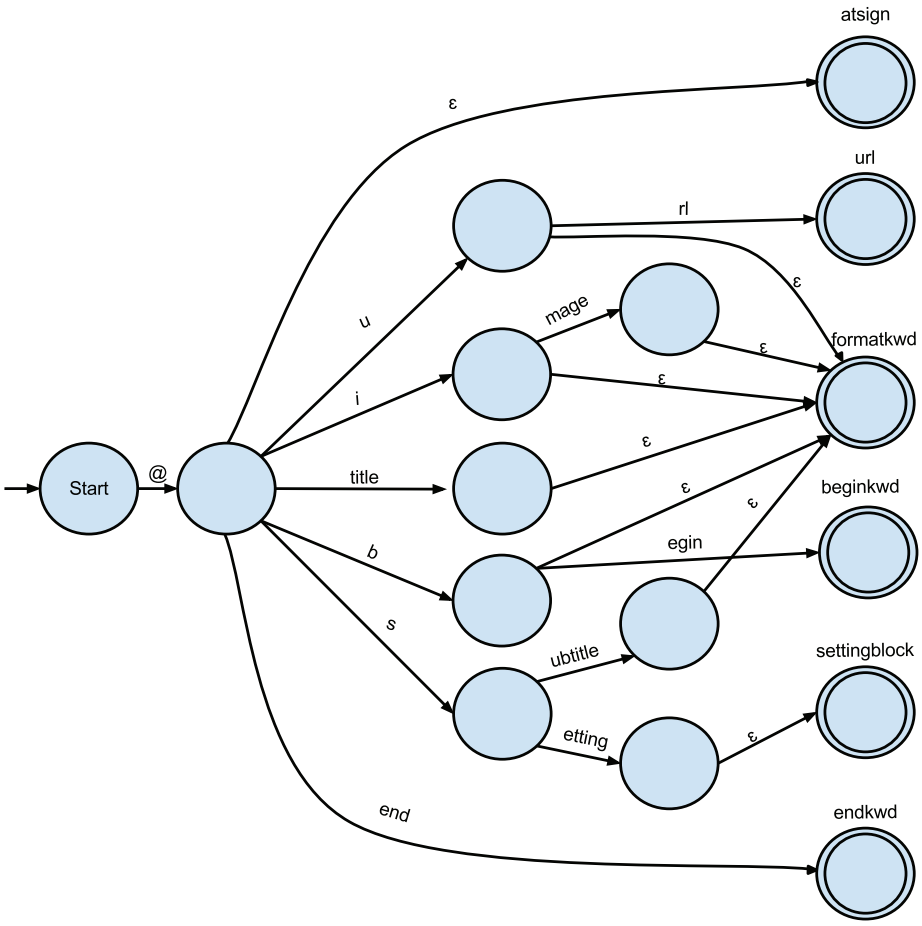

Figure 7.1: NFA for the tokens starting with atsign

Chapter 8

Parser

In this chapter the requirements for NISSE will be presented, along with the CFG hereof.

8.1 Requirements

The requirements for the NISSE parser are:

- The parser should be able to receive tokens from the lexer, which it then can convert into a parse tree.
- It should be able to report useful error messages if the user has written something illegal according to the CFG of NISSE.
- The CFG should be unambiguous, such that it is possible to make a parser for it.

8.2 CFG

Listing 8.1: CFG of NISSE in EBNF.

```
1  SS              = Blocks ;
2  Blocks          = BeginBlock {Lines} Endblock | SettingBlock↩
       ;
3  Lines           = SettingBlock | Numeration | Itemlist | ↩
       Plains eol ;
4  Numeration      = nlist (Plains eol | Numeration) ;
5  Itemlist        = blist (Plains eol | Itemlist) ;
6  BeginBlock      = beginkwd {space} BEBlock eol ;
7  EndBlock        = endkwd {space} BEBlock eol ;
8  BEBlock         = lcurly {space} char {space} [BEBlockv1] ↩
       rcurly ;
9  BEBlockv1       = pipe {space} char {space}  ;
10 SettingBlock    = settingkwd lcurly ShortIdent {space} pipe ↩
       {space} char {space} rcurly {space} eol ;
11 Plains          = (ShortBlock | CharAll) {(ShortBlock | ↩
       CharAll)} ;
12 ShortBlock      = format_kwd lcurly {space} [ShortIdents] ↩
       Plains rcurly ;
13 ShortIdents     = ShortIdent pipe ;
14 ShortIdent      = Kwd {space} colon {space} ShortIdentv1,{↩
       ShortIdentv1} {space} ;
15 ShortIdentv1    = char | digit | Float | colon | fslash | ↩
       dot ;
16 Kwd             = atsign char | url ;
17 CharAll         = colon | digit | scolon | percent | fslash ↩
       | bslash | exclamation | dot | comma | char | space | ↩
       underscore | hyphen;
18 Float           = digit dot digit ;
19 eol             = eolv1 , {eolv1} ;
20 dot             = dotv1 , {dotv1};
21 comma           = commav1,{commav1} ;
```

Listing 8.1 shows the CFG for NISSE in EBNF. With the rules of table 7.1.

NISSE's grammar is able to construct everything that is required of NISSE. Figure 8.1 shows how the CFG would parse the AST, with the input of listing 6.3.

8.3 Conclusion

The CFG is concluded to be LALR according to section 5.2, the CFG is not ambiguous either, as per the definition of LALR ([2] Chapter 6).
The grammar for the language is LL(1) according to the tool called "kfG Edit", which was introduced in a lecture. Later it was tested if the grammar was

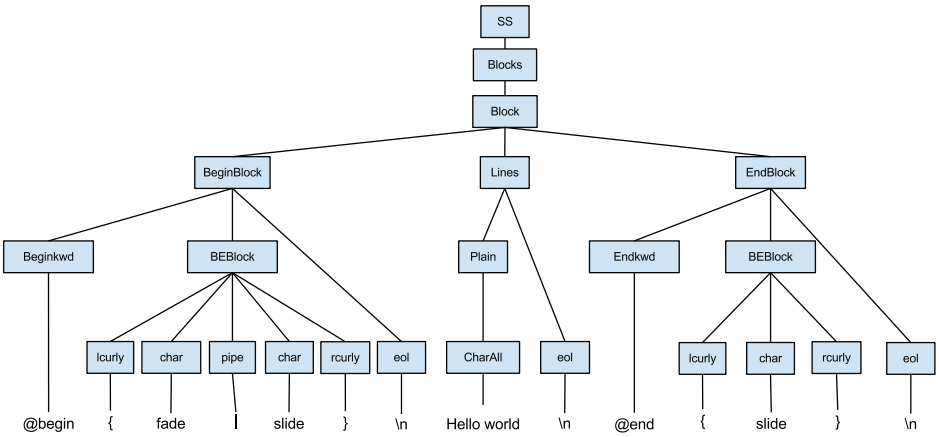

Figure 8.1: Parse tree for a simple slide.

LALR(1) by using SableCC to try and generate a parser, and it was proved that the grammar was LALR(1). "kfG Edit" uses a special syntax that can be seen in appendix A.1.

Chapter 9

Operational Semantics

In this chapter some of the operational semantics of NISSE will be shown. In order to be able to read the operational semantics a few things should be known.
`S` denotes slides and is an environment consisting of variables (slides) pointing to a location with the `body` of the slide.
Also included in the environment is the keyword `next` which points to the next empty variable, and the function `new` which gets the next variable after its parameter.
`env` denotes an environment for settings, which also consists of variables pointing to locations containing the value of that setting. `env` does not need the `next` keyword and the `new` function because all of the settings is predefined and the only thing changed is the settings' value.

[*slideshow*]

$$\langle S, env\rangle \rightarrow show$$

The transition describes how the entire slideshow is made, which are with elements in `S` that has the variables in the environment.

[*specification*]

$$\frac{\langle slide, S\rangle \rightarrow S'\langle setting, env\rangle \rightarrow env'}{\langle slide\ setting, S, env\rangle \rightarrow S', env'}$$

This transition describes when the command slide is executed in `S`, which changes S. And when a setting is executed in the variable environment, the environment is changed.

[*setting*]

$$\frac{\langle shortident, env\rangle \rightarrow env'\langle scope\rangle}{\langle @setting\{shortident|scope\}, env\rangle \rightarrow env'}$$

This transition describes the inside of a setting, how the element inside a setting block changes in the variable environment in the specific scope.

[*slide*]

$$\frac{\langle block, S[L \mapsto block][next \mapsto newL]\rangle \to S'}{\langle block, S\rangle \to S'}$$

$$L = S(next)$$

This transition describes when a block is executed in `S`. This inserts the slide inside `S`, and moves the pointer to the next location where a slide can be inserted.

[*block*]

$$\frac{\langle setting, env\rangle \to env'\langle plain,\ S\rangle \to S'\langle num\rangle\langle ilist\rangle}{\langle begin\ setting\ plain\ num\ ilist\ end, S\rangle \to S'}$$

This transition shows how commands inside a block (slide) is executed, resulting in a change in a slide.

[*plain*]

$$\frac{\langle plaintext,\ S\rangle \to S'\langle shortblock,\ env\rangle \to env'}{\langle plaintext\ shortblock,\ S\rangle \to S'}$$

This transition illustrates how plain text and/or shortblock is evaluated, with the environment `env`.

[*shortblock*]

$$\frac{\langle formatkwd\rangle\langle shortident, env\rangle \to env'\langle plain\rangle}{\langle formatkwd\{shortident|plain\}, env\rangle \to env'}$$

This transition changes the environment according to the `formatkwd` and `shortident`, for the specific plain text.

[*shortident*]

$$\frac{\langle env[L \mapsto V]\rangle \to env'}{\langle kwd : V, env\rangle \to env'}$$

$$L = env(kwd)$$

This transition changes the environment according to the keyword and the variable `V`.

9.0.1 Derivation Tree

An example of a derivation tree is made in figure 9.1 of the language code in listing 9.1.

Listing 9.1: NISSE example

```
1 @begin{slide}
2     @title{Next lecture}(1)
3     @subtitle{On 15.05.2012}(2)
4     This was all for now.(3)
5     Have a nice day.(4)
6     The current slideshow can be found at(5) @apply{@url:↩
          http://www.somelink.com | this link }(6).
7 @end{slide}
```

The number at the end of almost each line i listing 9.1 corresponds to the number in figure 9.1, for what it represents. The `Shortblocks` in figure 9.1 should be seen as being on the same line as `Plaintext`. `plain(1)` and `plain(2)` in figure 9.1 derives to two separate shortblocks. Even though the figure only shows one shortblock, which then changes the environment, according to the shortident and formatkwd, to the `Plain`, which derives to Plaintext, and changes the content of the slide. The same goes for `plain(6)`

$$\dfrac{\begin{array}{l}\textit{begin}\quad \dfrac{\dfrac{< formatkwd\{\dfrac{kwd:V,\ env \to env}{shortident} \mid \dfrac{Plaintext,\ S \to S}{Plain}\},\ env >\to env}{Shortblock,\ S \to S(1)(2)}}{plain(1)\quad plain(2)} \quad \dfrac{Plaintext,\ S \to S(3)(4)(5)}{plain(3)\quad plain(4)\quad plain(5)} \quad \dfrac{< formatkwd\{\dfrac{kwd:V,\ env \to env}{shortident} \mid \dfrac{Plaintext,\ S \to S}{Plain}\},\ env >\to env}{Shortblock,\ S \to S(6)} \quad \textit{end} \\ \hline \qquad\qquad \dfrac{< slide,\ S,\ env >\to S,\ env}{< S,\ env >\to show}\end{array}}{plain(6)}$$

Figure 9.1: Derivation Tree for listing 9.1

Chapter 10

Semantic Analyser

The semantic analyser checks for errors that the lexer and the parser do not catch, which only leaves runtime errors remaining. The semantic analyser for the language has the following requirements to be able to generate the expected slide.

10.1 Requirements

An example of the requirements for the semantic analyser are in the listing below the requirment, and the important words are highlighted in bold.

- Check that the setting in the setting block is valid.

```
1 @setting{@font_color:blue | text}
```

- Check that it is a valid input after the colon in a setting block.

```
1 @setting{@font_color:blue | text}
```

- Check that it is a valid scope on the right side of the pipe in a setting block.

```
1 @setting{@font_color:blue | text}
```

- Check that the transition used exists in the database.

```
1 @begin{fade | slide}
```

- Check what type of slide it is, for use in the code generator.

These requirements are put into categories, with more explanation to specify the function of the semantic analyser.

10.2 Semantic Analysis for Text Formatting

10.2.1 Expression Existence

In order for the settings to work, a check to see if a specific setting actually exists in that context is needed. In order to check if the setting exists, a list of settings for a specific context is set up beforehand which it can be checked against. If the setting exists in the list, the value of the setting is checked to see if it matches a valid value of that setting. If it does not exist, an error is written to the user.

10.2.2 Type Checking

For a specific setting there is a number of valid values, two examples could be the settings `font_size` and `font_weight`, which sets the size or weight of a specific text, respectively.
In the case of `font_size`, a valid input would be any integer.
In the case of `font_weight`, a valid input would be `bold, italic` or `underline`.
As with the keyword existence of an error is written if the value of the setting is not valid. The type checking for integers and floats is done by converting the value from a string to an int, or float, respectively. Existence checking for a specific string is done with `if` sentences, if no specific string matches any `if` sentence, an error occurs.

10.2.3 Scope Checking

Targeted text

Every setting block has to be given a scope of what it is going to affect. An example could be `global`, which, for a given setting would set all types of text in the given effect level unless it is overridden at a later stage.

Effect Level

When a setting block is inside a begin- and end block, the setting only affects that particular slide.
A setting block can also be used outside a line, which would make that setting apply to every slide following that specific slide, at which it is defined, unless it is overridden at a later stage. These scope settings have to be checked as to see what text the setting should apply to.

Listing 10.1 is a code example of when the compiler enters a setting node.

The "Visibility" word in this section means which font type is changed; title, text, subtitle, etc.

Listing 10.1: SettingBlock

```
1  public void inASettingblock (ASettingblock node){
2    String SettingType = node.getShortident().toString().trim↩
       ();
3    String Visibility = node.getChar().toString().trim();
4    String Test = node.parent().parent().getClass().toString()↩
       ;
5    if (Test.equals("class nisse.node.ABlockBlocks")){
6
7      if (SettingType.startsWith("@ font _ color")){
8        String Value = SettingType.substring(17);
9        if (Visibility.equals("global")){
10           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewTextFontColor] = Value;
11           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewTitleFontColor] = Value;
12           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewSubtitleFontColor] = Value;
13           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewImageFontColor] = Value;
14           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewUrlFontColor] = Value;
15       }
16       else if (Visibility.equals("text")){
17           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewTextFontColor] = Value;
18       }
19       else if (Visibility.equals("image")){
20           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewImageFontColor] = Value;
21       }
22       else if (Visibility.equals("title")){
23           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewTitleFontColor] = Value;
24       }
25       else if (Visibility.equals("subtitle")){
26           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewSubtitleFontColor] = Value;
27       }
28       else if (Visibility.equals("url")){
29           SymbolTable.Scope[SymbolTable.ScopeLevel][↩
               SymbolTable.NewUrlFontColor] = Value;
30       }
31             else {
32             Error.MakeError("Visibility existence" , Value);
```

```
33        }
34      }
35      else {
36            Error.MakeError("Font color existence" , Value);
37      }
38    }
39    else if (SettingType.startsWith("@ font _ family")){
40      String Value = SettingType.substring(18);
41     ....
```

Listing 10.1 illustrates "setting type" check, and changes parameters according to the "visibility" parameter, which is placed just before the right curly bracket in a setting block.
Line 2 fetches the setting, node *Shortident*, which contains information about what setting to change and what to change it to, which is converted to a string, then the excess spaces are removed. Spaces are added by the `toString` method, to seperate tokens from each other.
Line 3 fetches the visibility declaration, which is in the node *Char*, which is also converted to a string and the excess spaces are removed.
Line 4 finds out which class the setting block belongs to. This is necessary to determine whether the setting block is inside or outside a slide, which is the first thing to be checked in line 5. An *else* construction is made for this *if* sentence which is executed when the setting should be applied on every upcoming slide.
Line 7 checks which keyword is going to be changed, in this case it is *Font_ Color*. The parameter is then stored in a string called *Value*, which takes the substring of SettingType, starting at character 18. The reason it is 18 is because a space is added after each token. So there is a space after *font_ color* and a space after the colon (:) which makes the parameter start at the 18^{th} character.
From line 12 to 33 it checks what the "visibility" keyword is. Depending on the word, it changes the parameter for that specific visibility.
Line 13 enters the document "SymbolTable" and the array *Scope*, which inserts the parameter at the current scope level, and in the cell containing the information about text font colour. From line 34 to 41 creates errors in the error table if the lines are executed.

10.3 Semantic Analysis for Blocks

This section focuses on blocks. A block consists of at least two lines, a `begin` line and an `end` line. Between those two lines, information can be stored.
The generic setup for the two lines are shown in section 6.1.1 and 6.1.2. 10.2 is a listing of semantic for transition existence and slide type checking.

Listing 10.2: Function: OutBlockBlocks

```
1  public void outABlockBlocks (ABlockBlocks node){
2      String Transition = node.getBeginblock().toString();
3      Transition = Transition.substring(9);
4      String Transition1 = "none";
5      if (Transition.startsWith("slide")){
6      Transition1 = SymbolTable.Transition;
7      }
8      else if (Transition.startsWith("fade")){
9        Transition1 = "fade";
10     }
11     else if (Transition.startsWith("swipe")){
12       Transition1 = "swipe";
13     }
14     else if (Transition.startsWith("scale")){
15       Transition1 = "scale";
16     }
17     else if (Transition.startsWith("rotatescale")){
18       Transition1 = "rotatescale";
19     }
20     else {
21       Transition1 = SymbolTable.Transition;
22       Error.MakeError("Transition existance" , Transition);
23     }
24     String SlideType = "Unknown";
25     Object[] Slide = node.getLines().toArray();
26     int Lines = Slide.length;
27     int i = 0;
28     int title = 0;
29     int subtitle = 0;
30     int image = 0;
31     for (i=0; i<Lines; i++){
32       String Slide1 = Slide[i].toString();
33       if (Slide1.startsWith("@setting")  ||  Slide1.↩
             startsWith("@note")){
34       }
35       else if (Slide1.startsWith("@title") ) {
36         title++;
37       }
38       else if (Slide1.startsWith("@subtitle") ) {
39         subtitle++;
40       }
41       else if (Slide1.startsWith("@image") ) {
42         image++;
43       }
44       else {
45         SlideType = "Normal";
46         SymbolTable.SlideTableAdd(SlideType, Transition1);
47         indent--;
```

```
48          return;
49        }
50      }
51      if (title > 0 && subtitle == 0 && image == 0){
52        SlideType = "Title";
53      }
54      else if (title == 0 && subtitle > 0 && image == 0){
55        SlideType = "Subtitle";
56      }
57      else if (title == 1 && subtitle == 0 && image == 1){
58        SlideType = "TitleImage";
59      }
60      else if (title == 1 && subtitle == 1 && image == 1){
61        SlideType = "TitleSubtitleImage";
62      }
63      else if (title == 0 && subtitle == 1 && image == 1){
64        SlideType = "SubtitleImage";
65      }
66      else if (title == 1 && subtitle == 0 && image == 2){
67        SlideType = "TitleDImage";
68      }
69      else if (title == 1 && subtitle == 1 && image == 2){
70        SlideType = "TitleSubtitleDImage";
71      }
72      else if (title == 0 && subtitle == 1 && image == 2){
73        SlideType = "SubtitleDImage";
74      }
75      else if (title == 1 && subtitle == 0 && image == 3){
76        SlideType = "TitleTImage";
77      }
78      else if (title == 1 && subtitle == 1 && image == 3){
79        SlideType = "TitleSubtitleTImage";
80      }
81      else if (title == 0 && subtitle == 1 && image == 3){
82        SlideType = "SubtitleTImage";
83      }
84      else if (title == 1 && subtitle == 0 && image == 4){
85        SlideType = "TitleQImage";
86      }
87      else if (title == 1 && subtitle == 1 && image == 4){
88        SlideType = "TitleSubtitleQImage";
89      }
90      else if (title == 0 && subtitle == 1 && image == 4){
91        SlideType = "SubtitleQImage";
92      }
93      else if (title > 0 && subtitle > 0 && image == 0){
94        SlideType = "XTitleXSubtitle";
95      }
96      else if (title == 0 && subtitle == 0 && image == 1){
97        SlideType = "Image";
```

```
98      }
99      else if (title == 0 && subtitle == 0 && image == 2){
100       SlideType = "DImage";
101     }
102     else if (title == 0 && subtitle == 0 && image == 3){
103       SlideType = "TImage";
104     }
105     else if (title == 0 && subtitle == 0 && image == 4){
106       SlideType = "QImage";
107     }
108     else if (title > 0 && subtitle > 0 && image > 0){
109       SlideType = "Lots";
110 }
111     SymbolTable.SlideTableAdd(SlideType, Transition1);
112   }
```

10.3.1 Transition Existence

In order for the compiler to apply a transition on a slide, the analyser has to check whether the written transition exists in the list of transitions. If the transition does not exist an error occurs. If the transition does exist, it continues without error.

In listing 10.2 from line 2 to 18 the function finds out if, and what, transition the slide has.
In line 2 the function creates the `@begin` line to a string, which can be operated on.
In line 3 the function creates a substring of line 2, beginning at character 9 which is where the transition should be written, if any.
Line 4 sets the transition to `none`.
From line 5 it checks if the transition is `slide`, in this case there is no explicit transition, which is why it fetches the transition variable set by a setting block, this variable is `none` by default.
From line 8 to 19 it checks what kind of transition it is, and sets it to the found transition, if the transition is not found, the slide will have no transition.
In line 20 to 23 the function sets the transition to default, and creates an error, because it was not the correct transition which was wanted by the user.

10.3.2 Slide Type Checking

There are different types of slide layouts. For instance, most slide presentations starts with a title slide, with or without a subtitle. The front page contains a title, and in some cases a subtitle, which both is in the middle of the slide. This placement is defined per default, instead of providing the user the explicit choice of this, to make it less complicated for the user.
The semantic analyser will check this, and creates the correct type of slide ac-

cording to the text input from the user.

In listing 10.2 from line 24 to 112, the type of slide is determined. It starts by being `Unknown`, but it should never be that in the end. Then, in line 25 it creates an array of each object (line) in the slide. The number of lines is determined in line 26.
The number of title-, subtitle- and image lines is set in line 28 - 30.
To check how many there are, by checking the number of lines, a loop for each of the lines is made, and increment the variable for the specific type. If a normal text line is encountered, it will automatically be a `normal` slide.
From line 51 to 110, the function checks how many lines there are of each type, and depending on the amount, it sets the `SlideType` to the correct slide type, in order to show the slide properly.
Line 111 adds the slide properties to the slide table.

10.3.3 Exception Handling

Errors found in the semantic analysis, for instance typing "bluue" instead of "blue" in a `font_color` command, or writing letters in a `font_size` command, results in the command being ignored and a description of the error is added in an error table containing the slide number, a description of the error and the `Value` containing the error, in this case the description would be "font colour existence" and `Value` would be "bluue".
Listing 10.3 creates an object in the `ErrorList` array, where the first dimension states which error number it is. The second dimension in cell 0 contains the information about what slide number the error has occured.
In cell 1 the type of error is inserted, and in cell 2 is the Value of the error.
An example of errors are made in table 10.1, where the the user tries to change the font colour in slide number 2 to `SomeColor`, and the font color in slide 3 to `bluue`, which is not a valid colour.

Listing 10.3: Function: MakeError

```
1  static int Slidenr = 1;
2  static int Errors = 0;
3
4  public static void MakeError(String Type, String Value){
5    ErrorList[Errors][0] = Integer.toString(Slidenr);
6    ErrorList[Errors][1] = Type;
7    ErrorList[Errors][2] = Value;
8    Errors++;
9  }
```

Slide number	Type of error	Value
2	Font colour existence	SomeColour
3	Font colour existence	bluue

Table 10.1: An example of a Errorlist, with 2 errors

10.4 Generated Tables

The semantic analyser builds two tables for the code generator and one table for errors:
The symbol table, containing the symbols that are in the slide,
The slide table, containing information about a slide, and
The error table containing information about errors made by the user.

10.4.1 Symbol Table

In order to know the properties for each token that is written by the user, a symbol is added to the symbol table, containing the necessary information for the code generator to display the token correctly.

Listing 10.4: Function: SymbolTableAdd

```
1 public static void SymbolTableAdd(String Text, String Type, ↩
      String FontSize, String FontFamily,String FontColor, ↩
      String FontLineheight, String FontWeight, String Link){
2 String[] Values = {Text, Type, FontSize, FontFamily, ↩
      FontColor, FontLineheight, FontWeight, Link};
3 SymbolTable1.put(GetCurrentSymbolNumber(), Values);
4 NextSymbolNumber();
5 }
```

Listing 10.4 shows the function called to create a symbol in the symbol table. It stores all the variables in an array which is inside a hash table. Each hash entry has a unique number in order to fetch the entry. Before the function ends the number is incremented by one, which is the next unique hash key.

10.4.2 Slide Table

An additional table is created to keep track of the overall properties in a slide, meaning the transition between the slides, and which type of slide they are, e.g. a title page or a normal page. The slide type is determined in listing 10.2. In line 56 in listing 10.2, it calls the function SlideTableAdd, which is shown in listing 10.5, where the transition at the beginning is set to `none`, but can be modified in the function `outABlockBlocks` in listing 10.2. The slide table looks more or less like the symbol table, only with fewer elements in the array.

Listing 10.5: Function: SlideTableAdd

```
1 static String Transition = "none";
2
3 public static void SlideTableAdd(String Type, String ↩
    Transition){
4       String[] Values = {Type, Transition};
5       SymbolTableForSlide.put(GetSlideNumber(), Values);
6       NextSlide();
7     }
```

Chapter 11

Code Generator

The code generator takes the decorated AST from the semantic analysis and with it, it generates the output code. In the case of NISSE, the output from the code generator is a mixture of HTML (Hypertext Markup Language), CSS(Cascading Style Sheets) and JavaScript. The code generator has the following requirements in order to generate a correct slideshow.

11.1 Requirements

- Print out the header of the HTML-files, along with the necessary libraries needed for the slideshow to function.
- Retrieve information about slide type and slide transition from the slide table about all of the slides.
- Print out all of the plain text, numeration and item lists with the right settings, according to the symbol table.
- Keep track of when to open and close tags, such that correct HTML is output, while keeping the file as small as possible.
- End the HTML-file and output the generated code to the output file.

Each of these requirements will be described further in the following subsections.

11.2 Header File and Libraries

The common header file of all generated slides is kept in a java file called `prefix` inside the compiler. The header file consists of; HTML-metatags, the common CSS of all slides, the JavaScript libraries and the start of the body of the HTML-file.

The HTML-metatags are there such that the slideshows works on mobile devices (e.g. Apple Iphone/Ipad or Android phones/tablets), because without them the slideshows would not scale correctly for the mobile device screens. The metatags also disables user zoom such that the user can use swipes to change slide without the need to worry about placement or size of the slide.

The common CSS of all slides is mainly how the page itself is setup, how a slide looks and the background colour.

A few JavaScript libraries are used to make the slideshows work, these libraries are; `jQuery` and a plugin for `jQuery` called `Touchswipe.`
`jQuery` is mainly used to facilitate all of the transitions of the slides, e.g., it makes it easy to animate CSS attributes.
`jQuery` is also used to change CSS attributes. The plugin `Touchswipe` is used to make it possible to use swipe motions on mobile devices to change slide, which would otherwise not be possible.

A custom library is also included which uses `jQuery` and `Touchswipe` to implement the different transitions. A key hook is set up such that when the user presses a button, the code checks to see if the user pressed any of the buttons to change the slide. A similar `swipe hook` is set up using `Touchswipe`, to catch user swipes and check what direction the swipe was in. These hooks then calls a common function called `go_to_slide` which sends the user to the requested slide. The `go_to_slide` function checks the `data-transition` tag of the requested slide to check which transition to run. The function then calls the appropriate function which will then change to the requested slide with the given transition. The last thing included in the custom library is such that when desktop or laptop users resize their windows, the slide and text size is automatically scaled down such that the slide still fits the whole window.

11.3 Transitions

The compiler supports four different types of transitions between slides. These are; `fade`, `swipe`, `scale` and `rotatescale`. All of the animations required for the transitions are done through `jQuery`. NISSE supports the use of different transitions on slides in the same slideshow. The way this is done is that the first slide in the slideshow is placed in the centre of the screen, while the rest of the slides are moved 100 % down to a slide stack and is hidden. The reason the rest of the slides are moved 100 % down is that there is a limitation in HTML such that if elements are hidden above other elements, the user is not able to click at hyperlinks. Each transition moves the slides into place in order for the animation to work.
The fade transition starts by moving the slide it is switching to, to the middle of the screen (while it is still hidden). Using `jQuery` the CSS option opacity

is then animated to 0 for the current slide. After that animation has finished the next slides' opacity is then animated to 1, making it visible. Finally the previous slide is moved back to the slide stack.
The swipe transition starts by moving the slide it is switching to, to either the left or right of the screen (depending on whether the user is moving forward or backwards in the slides) and moves the slide up to the centre of the screen. The current slide is then animated either to the left or right, while the next slide is animated into the centre of the screen. While this is going on, the current slides' opacity is animated to 0, while the next slides' is animated to 1. Finally, the previous slide is moved back to the slide stack.
The scale transition starts by moving the slide it is switching to, to the middle of the screen. The scale transition makes use of the CSS option transform, which is implemented in major browsers using their own `tag`. In order to support all of the major browsers, all four versions of the transform option is animated. The four versions are;

`-webkit-transform` for all browsers built upon webkit (e.g. Chrome or Safari),
`-moz-transform` for all of the Mozilla browsers,
`-o-transform` for the Opera browser, and
`-ms-transform` for all of the Microsoft browsers.

The option of transform used is the `scale()` option. Because `jQuery` can only animate numeric CSS options, a workaround is used in order to animate the scale option. A CSS option that is not used in this context (in this case `border-spacing`) is animated in the range wanted, and every time the animate function steps another function is called which modifies the transform option with the value that the animate function has reached. At the same time, the next slide is faded in. Finally the previous slide is then moved back to the slide stack.
The `rotatescale` transition works similarly to the scale transition, but while animating the scale option it also animates the rotate option. Like all other transitions, the previous slide is moved back to the slide stack.

11.4 End of the Header

All of the previous code was placed inside the `header` of the HTML-file. The code generator now starts the `body` of the file and outputs the `wrapper` of the slides.

11.5 Starting Slides

Slides in the HTML-file are wrapped in a `div` that has the class `slide_wrapper`. The div also has to have some tags set in order for the slideshow to work. The

first one is the `id` of the `div`, each slide has to have a unique id starting with `slide0` and incrementing the number for every slide, this is used when the slideshow moves to a slide.
The second tag is the `data-transition` tag, which specifies what transition should be used when moving `away` from that slide.
The last tag that has to be set is the style of the `div`. The first slide of the slideshow is placed in the middle of the screen and is visible, specified by the CSS `top:0%; opacity:1;`. The rest of the slides is placed 100 % below the screen and is hidden, specified by the CSS `top:100 %; opacity:0;`.
The slides is then also wrapped in a `div` with the class equal to the type of slide, e.g. `titleslide`, `imageslide`.

11.6 Printing Out Plain Text

Because most of the work figuring out what the style of plain text should be is done in the semantic analyser, the code generator part of printing out plain text is relatively simple. All of the style for text can be retrieved from the symbol table. The only challenge is figuring out when to open and close tags in the HTML-file. Because text has to be surrounded with tags the code generator has to check when to close tags when new text is met, instead of when moving "out" of the text. Using this method the code generator is also able to reduce the amount of code outputted by only starting a new `span` when the style of text changes. Titles and subtitles are also special in that style can change while still in the same title, which means that `span` has to be used inside titles in order for style to change inside it.

11.7 Ending the HTML-file

After all of the slides have been generated, the last thing the code generator does is output some last HTML-tags in order to close out the HTML-file.

11.8 Compiling a NISSE Slide Show

The NISSE compiler is packaged as a JAR (Java ARchive) file, which has to be run in a terminal. The syntax for the compiler is "java -jar nisse.jar input [output]" where `[]` specifies optional arguments. The same can be seen if the compiler is run with zero arguments. If no output is specified a file named "output-timestamp.html" is outputted, where timestamp is a time stamp of when the file was created.

Chapter 12

Test

12.1 Time Efficiency Test

In the beginning of this project, some requirements for the slideshow programming language were defined. One of these was that the language should be faster to create slideshows in, than already existing slideshow programming languages.
To see whether the developed language was able to fulfil this requirement, a test was made, to see how long it would take to create slideshows in the developed language, and compared it to other languages and programs for creating slideshows.

Earlier in the report a few other slideshow solutions was mentioned; Apple Keynote, Microsoft PowerPoint and LaTeX Beamer. These solutions will be compared to the developed language.
The test consists of five slides, these slides can be found in appendix A.3. The slides were the same on each of the tests, to make the complexity the same. The test person was one of the group members, who has experience in the developed language, and also the rest of the slideshow solutions. The result of the test is shown in table 12.1. The tests were made in the same order they occur in the table.

Program/Language	Time
NISSE (PC)	09 minutes 45 seconds
LaTeX Beamer (PC)	$\sim$ 45 minutes
Microsoft PowerPoint (PC)	05 minutes 55 seconds
Apple Keynote (Ipad)	12 minutes 40 seconds
NISSE (Ipad)	09 minutes 00 seconds

Table 12.1: Table showing the test results

The testing of LaTeX Beamer took a lot of time, because additional packages had to be installed on the test computer, and specific commands had to be found on the Internet in order to create the test slideshow.
The Microsoft PowerPoint presentation was the absolutely fastest way to create the test slideshow, but this test was mainly made to compare pointing languages with non-pointing language. This showed a significant amount of saved time compared to the rest.
The main result of the test is that creating the test slideshow using LaTeX Beamer on a PC and NISSE, shows that it is more time efficient to create slideshows using NISSE. NISSE beats LaTeX Beamer because the user has to browse the Internet for certain commands and packages, while using LaTeX Beamer.

The reason that it was faster to create the test slideshow with NISSE on an Ipad than on PC, might have been that the test person knew what was on the slides, after creating the same slideshow multiple times.

12.2 Language Tests & Known Errors

To find errors or bugs in the language or compiler a series of tests have been run to verify that a given input would result in a viable output. Testing was done by writing many different viable constructs and combining them in different ways. Below is some of the constructs that was made, and some of the errors have been corrected due to faulty code in the compiler, others have been addressed for further development.

Listing 12.1: NISSE `@apply` construct

```
1 @begin{slide}
2 @apply{@font_family: Times New Roman | this text is in Times↩
      New Roman}
3 @end{slide}
```

In listing 12.1 the construct results in the text family being `Times New Roman`. That is the expected behaviour of the text family, but it does not work, because the parser only recognizes;
`digits`, `floats`, `colons`, `forward slashes`, `dots`, `underscores` or `hyphens`, (because the font family is recognized as a `shortidentv1`). It is not only `Times New Roman` that does not work; it is every font family that contains space(s) in their names, which is because the compiler does not accept a `space` as a setting, in the font family name.

Listing 12.2: NISSE construct

```
1 @begin{slide}
2 @image{@url:http://google.com/logo.png | @i{Image text.}}
3 @end{slide}
```

Listing 12.3: NISSE construct

```
1 @begin{slide}
2 @i{@image{@url:http://google.com/logo.png | Image text.}}
3 @end{slide}
```

In listing 12.2 the construct results in the image description being italic. That is the expected behaviour of format keywords.
In listing 12.3 the same construct is given, but the order of format keywords is opposite so the image is defined inside an italic construct.
This will result in the same as listing 12.2, which is also the expected outcome.

Listing 12.4: NISSE construct

```
1 @begin{slide}
```

```
2 @image{@url:http://google.com/logo.png | @title{Image text ↩
      as title text.}}
3 @end{slide}
```

Listing 12.5: NISSE construct

```
1 @begin{slide}
2 @title{@image{@url:http://google.com/logo.png | Image text↩
      .}}
3 @end{slide}
```

In listing 12.4 the same image construct as in listing 12.2 is given, but the image description here poses as a title. This will result in the image text have the properties of a title thus not being image text.
However, in listing 12.5 the construct order is the opposite as in listing 12.3 but the outcome is not the same. Here it will not get the title property as it is the inner construct that defines which type the text will be. This is therefore also the expected outcome but it is not consistent with other format keywords behaviour in the same context.

Listing 12.6: NISSE construct

```
1 @begin{slide}
2 ÆØÅ æøå
3 @end{slide}
```

In listing 12.6 danish characters is used, but the lexer does not properly recognize these as valid tokens. This is a bug that will not be corrected at this time, but will have to be done before initial release.

Listing 12.7: NISSE construct

```
1 @begin{s}
2 Slide content...
3 @end{s}
```

Listing 12.7 shows at line 1 the slide name of the slide simply being one character which is the same on line 3. This is a valid construct according to the intended use of slide names.
However in listing 12.8 line 1 and 3 does not match each other but no error is given. This is not as intended and should be corrected before initial release.
In listing 12.9 line 1 specifies a slide name that is also a valid input as a transition type, thus the slide will have that specific transition type. This is not as intended, the intention is that a pipe is given between transition type and slide name.
This is a bug which will not be corrected at this time, but should be before initial release. An example of a correct input is in listing 12.10.

Listing 12.8: NISSE construct

```
1 @begin{s}
2 Slide content...
3 @end{f}
```

Listing 12.9: NISSE construct

```
1 @begin{fade}
2 Slide content...
3 @end{s}
```

Listing 12.10: NISSE construct

```
1 @begin{fade | slideone}
2 Slide content...
3 @end{slideone}
```

Other tests include how the compiler reacts to syntactically invalid slideshows and which errors is given in different error scenarios. If an error is met in the lexer or parser the compiler will stop and give an error message that explains what the error is, and how it might be corrected. If an error occurs in the semantic analysis, and in the code generator, the faulty input is simply ignored and an error is written to an error table which can be viewed by setting the appropriate debug level when compiling.

Part III

Perspective

Chapter 13

Further Development

The developed slideshow programming language and compiler can be improved. The improvements are specified in a priority list, meaning that the those listed first should be granted the highest priority.

1. More describing error messages have to be made, in the lexer, parser and the semantic analyser, to make errors more understandable for the user.
2. The language has to be systematically, and thoroughly, tested, to make sure that no exceptions are not caught by the compiler.
3. The lexer should be able to support all the font families that CSS does, such that the user would be able to use every font family he would like to use.
4. It should be possible to make regions of slides, meaning that a user defined number of slides is in a region name or number. This region name, or number, can be used when changing settings, to target that region only.
5. Add the functionality behind giving slides names.
6. It would be nice to have a more programmer minded language. It could have a "main" function, which calls the slides. Giving the opportunity to name slides, and an easier way to switch the order of slides, gives the opportunity to call the same slide more than once.
7. Speakers' notes should also be implemented at some point, to make it more applicable in presentation situations, e.g. meetings.
8. The compiler should be able to complete the lexer even with errors, and if there are errors, then terminate when the lexer has finished, finally calling all of the errors. The same goes for the parser.
9. Another feature that could be implemented is the ability to change the background colour on the slides, or even insert an image as background.

10. Since slides have unique names, a reference function could be made, which writes the slide number of a specific slide name in a slide.

Chapter 14

Conclusion

The goal of this project is to make a complete programming language with a full functioning compiler, therefore during this project we have learned how to create our own programming language, and how to make a compiler for our language.

The product of this project can make a decent presentation, which was intended in the beginning of the project. The requirements defined earlier in the report for the product are solved.
The platform of the output (presentation) is an HTML-file, because it only requires a web browser to be able to be viewed.
The compiler outputs a presentation as an HTML-file, that makes for broad web browser compatibility. The user of the language does not need to have the layout of the slideshow in mind, because it is handled by the compiler itself. This creates a faster development of the slideshow for the user perspective.
The limitations of the product is mainly that it is very limited in how much you can design the presentation (because of its default settings). More limitations can be seen in the section about further development 13.
The language is able to express; titles, subtitles, URLs, images and normal text.
More technically, the "settings" include; font weight, colour, size and family changes, and when to begin and end a slide.

From this project, and the experiences obtained through it, it can be concluded that NISSE is simpler to express slideshows in, compared to LaTeX Beamer, through the performed tests herein, because;

- There are no packages that need to be included/downloaded to be able to use specific functionalities
- The language has proved to be faster to express slideshows in, because it is an HTML-based slideshow programming language, which means that all it takes to create a slideshow is a text editor and an Internet connection

to send the file to the server, have it compiled and have the HTML-file sent back to the user.

- NISSE cannot be used to create more extravagant slideshows, than you can when using LaTeX Beamer, which limits the capabilities of NISSE.

Part IV

Appendix

Appendix A

Appendix

A.1 kfG

Listing A.1 is the settings used in the program "kfG Edit", to be able to write ASTSableCC grammar for examples of the defined language. This is an older iteration of the cfg, which means it does not fit into the latest one.

Listing A.1: kfG grammar for the language

```
1  SS           -> Blocks
2  Blocks       -> Block Blocks | EPSILON
3  Block        -> BeginBlock Lines EndBlock | SettingBlock
4  Lines        -> Line Lines | SettingBlock Lines | EPSILON
5  Line         -> List | Plains eol
6  List         -> Numeration | Itemlist
7  Numeration   -> nlist Numv1
8  Numv1        -> Plains eol | Numeration
9  Itemlist     -> blist Itemv1
10 Itemv1       -> Plains eol | Itemlist
11 BeginBlock   -> beginkwd BEBlock eol
12 EndBlock     -> endkwd BEBlock Eols
13 BEBlock      -> lcurly BEBlockv1 pipe Ident Space  rcurly
14 BEBlockv1    -> Ident Space | EPSILON
15 SettingBlock -> settingkwd lcurly Kwd colon Settingv1 Space ↩
       pipe String rcurly eol
16 Settingv1    -> Ident| Number
17 Plains       -> Plain Plains | EPSILON
18 Plain        -> ShortBlock | CharSpace | Number
19 ShortBlock   -> Kwd lcurly ShortIdents pipe Plains rcurly
20 ShortIdents  -> ShortIdent | EPSILON
21 ShortIdent   -> Kwd colon Settingv1 Space ShortIdents
22 String       -> CharSpace Stringv1
23 Stringv1     -> String  | EPSILON
```

```
24 CharSpace    -> char | pace
25 Ident        -> char Identv1
26 Identv1      -> Ident | EPSILON
27 Number       -> digit Numberv1
28 Numberv1     -> Number | EPSILON
29 Kwd          -> atsign Ident
30 Space        -> pace Space | EPSILON
31 CharAll      -> colon | char | digit | nlist | blist | ↩
       scolon | percent | fslash | bslash
32 Eols         -> eol Eolv1
33 Eolv1        -> Eols | EPSILON
34
35
36 pace       -> ' '
37 settingkwd -> '@setting'
38 beginkwd   -> '@begin'
39 endkwd     -> '@end'
40 atsign     -> @
41 lcurly     -> {
42 rcurly     -> }
43 seperator  -> <>
44 pipe       -> '|'
45 fslash     -> /
46 bslash     -> \
47 colon      -> :
48 scolon     -> ;
49 blist      -> *
50 nlist      -> #
51 percent    -> %
52 eol        -> \n | \r | \r\n
53 char       -> a|b|...|z|A|B|...|Z| _ |.|,
54 digit      -> 0|1|...|9
```

A.2 CFG(SableCC)

Listing A.2 is the settings used in the program "SableCC", to generate the lexer and parser for the language.

Listing A.2: SableCC grammer for the language

```
1  Package nisse;
2
3  Helpers
4          charv1     = ['a' .. 'z'] | ['A' .. 'Z'] | 'æ' | 'ø' |↩
               'å' | 'Æ' | 'Ø' | 'Å';
5          digitv1    = ['0' .. '9'] ;
6          dot          = '.' ;
7          comma        = ',' ;
8          eolv1      = 13 10 | 13 | 10;
9
10 Tokens
11         format_kwd  = '@u' | '@b' | '@i' | '@apply' | '@image↩
               ' | '@title' | '@subtitle' | '@note' ;
12         url          = '@url' ;
13         space        =  ' ' | 9 ;
14         settingkwd   = '@setting' ;
15         beginkwd     = '@begin' ;
16         endkwd       = '@end' ;
17         atsign       = '@' ;
18         lcurly       = '{' ;
19         rcurly       = '}' ;
20         pipe         = '|' ;
21         fslash       = '/' ;
22         bslash       = '\' ;
23         colon        = ':' ;
24         scolon       = ';' ;
25         blist        = '*' ;
26         nlist        = '#' ;
27         percent      = '%' ;
28         exclamation  = '!' ;
29         underscore   = '_' ;
30         hyphen       = '-' ;
31         eol          = eolv1+;
32         char         = charv1+ ;
33         digit        = digitv1+ ;
34         float        = digitv1+ dot digitv1+ ;
35         dot          = dot+;
36         comma        = comma+ ;
37
38 Productions
39         nisse          = blocks* ;
40         blocks         = {block} beginblock lines* endblock
```

```
41                      | {setting} space* settingblock ;
42          lines       = {setting} settingblock
43                      | {numeration} numeration
44                      | {itemlist} itemlist
45                      | {plaintext} plains eol ;
46          numeration  = nlist numerationv1 ;
47          numerationv1 = {plaintext} plains eol
48                      | {numeration} numeration;
49          itemlist    = blist itemlistv1 ;
50          itemlistv1  = {plaintext} plains eol
51                      | {itemlist} itemlist;
52          beginblock  = beginkwd space* beblock [second]:↩
                space* eol ;
53          endblock    = endkwd space* beblock [second]:space*↩
                 eol ;
54          beblock     = lcurly [first]:space* char [second]:↩
                space* beblockv1? rcurly ;
55          beblockv1   = pipe [first]:space* char [second]:↩
                space*;
56          settingblock = settingkwd lcurly shortident pipe [↩
                second]:space* char [third]:space* rcurly space* ↩
                eol;
57          plains      = plainsv1+;
58          plainsv1    = {shortblock} shortblock
59                      | {charall} charall;
60          shortblock  = format_kwd space* lcurly shortidents?↩
                 plains rcurly ;
61          shortidents = shortident+ pipe ;
62          shortident  = kwd space* colon [first]:space* ↩
                shortidentv1+ [second]:space*;
63          shortidentv1 = {char} char
64                      | {digit} digit
65                      | {float} float
66                      | {colon} colon
67                      | {fslash} fslash
68                      | {dot} dot
69                      | {underscore} underscore
70                      | {hyphen} hyphen;
71          kwd         = {at} atsign kwdv1+
72                      | {url} url ;
73          kwdv1       = {char} char
74                      | {underscore} underscore;
75          charall     = {colon} colon
76                      | {digit} digit
77                      | {float} float
78                      | {semicolon} scolon
79                      | {percent} percent
80                      | {forwardslash} fslash
81                      | {backslash} bslash
82                      | {exclamation} exclamation
```

```
83                          | {dot} dot
84                          | {comma} comma
85                          | {char} char
86                          | {space} space
87                          | {underscore} underscore
88                          | {hyphen} hyphen;
```

A.3 Slide Example

The test code for NISSE in the "Time Efficiency Test" in section 12.1 is shown in listing A.3. The test slides contain a variation of elements, which can be expressed in NISSE. Figure A.1 shows the output of the listing A.3.

Listing A.3: Time efficiency test code for NISSE

```
1  @begin{slide}
2  @title{Frontpage}
3  @subtitle{d. 07-05-2012}
4  @end{slide}
5
6  @begin{slide}
7     @title{Lecture 1}
8
9     This is lecture 1 of 10 lectures in programming languages
10    The basic elements are:
11 *Loop
12 *Functions
13 *if statements
14 @end{slide}
15
16 @begin{slide}
17    Later in this course you will learn how to create ↩
          pictures. This is a block picture.
18    @image{@url:http://freeimagesarchive.com/data/media/3/2↩
          _black.jpg | http://media.desura.com/images/members↩
          /1/288/287053/black_1.2.jpg}
19 @end{slide}
20
21 @begin{slide}
22    We also have a lot of @b{breaks} in this course.
23    We will have a coffee break until the end.
24 @end{slide}
25
26 @begin{slide}
27    @title{Next lecture}
28    @subtitle{On 15.05.2012}
29
```

```
30    This was all for now.
31    Have a nice day.
32    The current slideshow can be found at @apply{@url:http://↩
          www.somelink.com | this link }.
33  @end{slide}
```

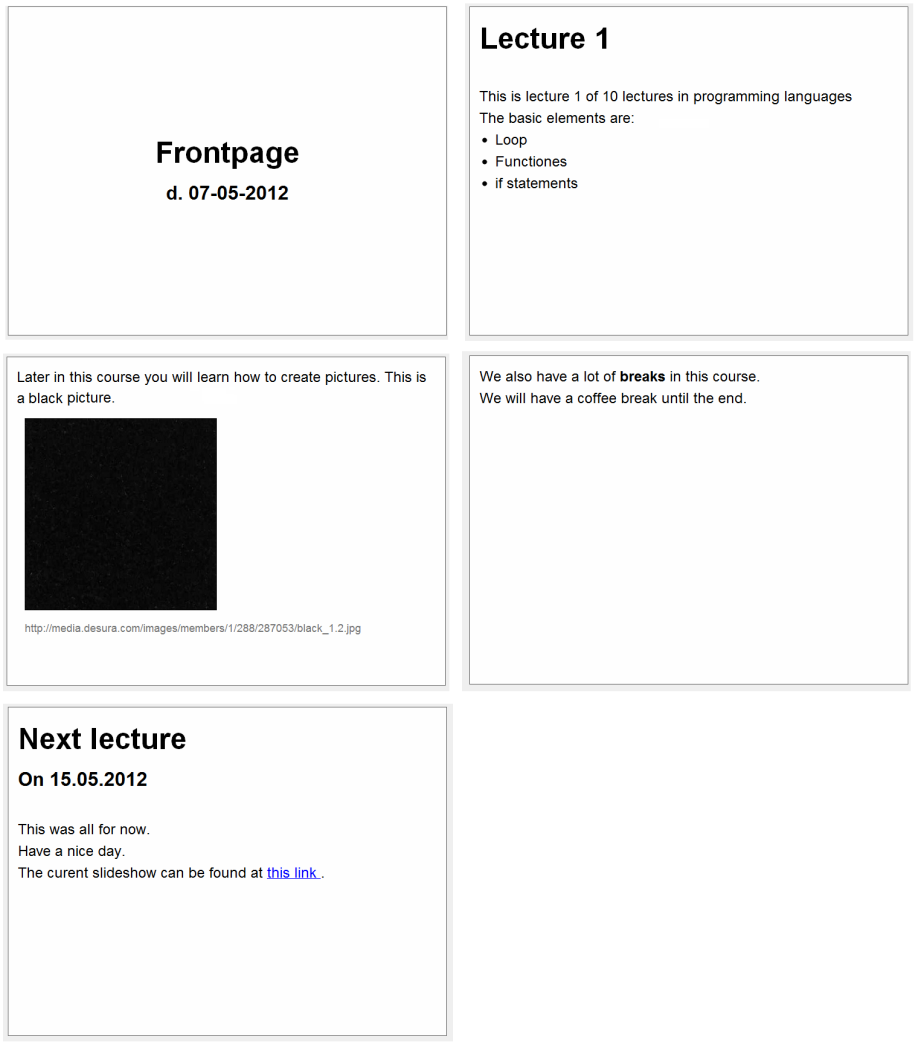

Figure A.1: Output of the code

A.4 Font Type

The following sections contains descriptions about the font types that can be changed in NISSE.

A.4.1 Font_family

The expression `font_family` sets the font family such as `Arial`, `Verdana`, etc., from the beginning of the slideshow or at a specific slide, such that the font family can be applied locally or globally. The language supports all the same font families as CSS does. An example of the use of changing the font family is shown in listing A.4

Listing A.4: Changing font family

```
1 @apply{@font_family: Verdana | this text is in Verdana ↩
      font family}
```

A.4.2 Font_color

The expression `font_color` sets the font colour from the beginning of the slideshow or at a specific slide, such that the colour can be applied locally or globally. The language supports all the same font colors as CSS does.
CSS 3 has inherited its extended colour keywords from SVG (Scalable Vector Graphics), which are the colours that can be used in NISSE [3]. An example of the use of changing the font colour is shown in listing A.5.

Listing A.5: Changing font colour

```
1 @apply{@font_color: Blue| this text is in blue colour}
```

A.4.3 Font_size

The expression `font_size` sets the font size from the beginning of the slideshow or at a specific slide, such that the size can be applied locally or globally. The language supports all the font sizes as the CSS does. The font size is defined in pixels. An example of the use of changing the font size is shown in listing A.6.

Listing A.6: Changing font size

```
1 @apply{@font_size: 70 | this text is in size 70}
```

A.4.4 Font_weight

The expression `font_weight` refers to the term `font weight`, which is the various settings that can be defined for the `font`, `bold` `italic` and `underline`. An example of the use of changing the font weight is shown in listing A.7.

Listing A.7: Changing font weight

```
1  @apply{@font_weight: italic| this text is in italic}
```

A.5 Front Page Code

Listing A.8 shows the code used to generate the frontpage of this report.

Listing A.8: NISSE code generating the frontpage

```
1 @begin{slide}
2 @title{Nearly Instantaneous}
3 @title{Slide Show Expressions}
4 @subtitle{@font_size:26|May 2012}
5 @subtitle{@font_size:20|By: Group SW403F12}
6 @end{slide}
```

Bibliography

[1] *Concepts of Programming Language,* 9^{th} *edition.* Pearson, 2010.

[2] *Crafting a Compiler.* Pearson, 2010.

[3] The World Wide Web Consortium. Recognized color keyword names. Website, August 2011. `http://www.w3.org/TR/SVG/types.html#ColorKeywords`.