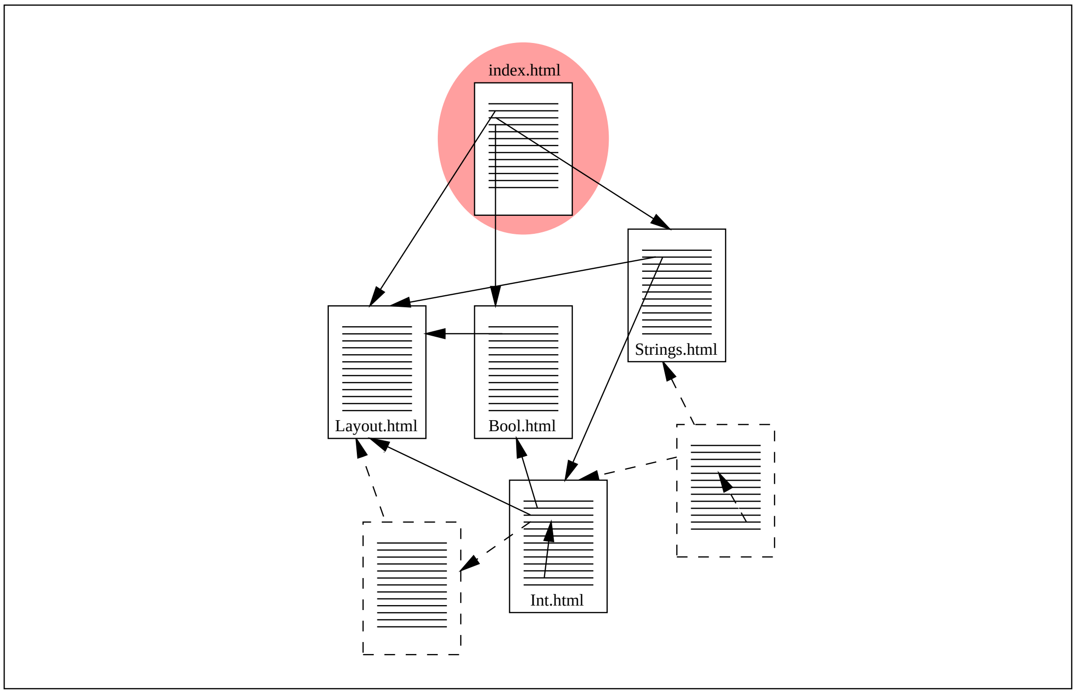
University of Amsterdam
Programming Research Group



A specification of Box to HTML in ASF+SDF

Maarten van der Graaf

Report P9720 november 1997

University of Amsterdam
Department of Computer Science
Programming Research Group

A specification of box to html in asf+sdf

Maarten van der Graaf

Report P9720 november 1997

M. van der Graaf

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7586
e-mail: graaf@wins.uva.nl

A specification of Box to HTML in ASF+SDF

Maarten van der Graaf

Doctoraalscriptie Informatica
Universiteit van Amsterdam
Faculteit WINS, sectie Programmatuur
Afstudeerdocent: prof. dr P. Klint
Begeleider: dr M.G.J. van den Brand

Augustus 1997

Abstract

Good documentation is important for the production of reusable and maintainable software. Tools that support the production of accurate and legible documentation are therefore essential for any programming environment. Documentation can be provided as printed output, like a typeset manual, or it can be provided on-line, as an electronic document. We present a translation from Box, a special purpose formatting language, to HTML. The translation is formulated in the executable algebraic specification formalism ASF+SDF. It is capable of producing effective representations of computer programs in HTML and supports the generation of hypertext links. It can be used as a back-end for the Box tools, a system that is capable of generating formatters for arbitrary programming languages from their syntax.

Contents

Chapter 1

Introduction

Good documentation is important for the production of reusable and maintainable software. For any programming environment it is essential to have tools that support the production of accurate and legible documentation. This relieves documentation writers from typesetting programs by hand and ensures that documentation is correct and up-to-date.

Documentation can be provided as printed output, like a typeset manual, or it can be provided on-line, as an electronic document. On-line documentation is often constructed using a hypertext system, which provides tools to create and browse a large collection of documents in a flexible way. The introduction of the HyperText Markup Language (HTML), which combines hypertext with conceptual markup, has made this approach even more powerful by defining a world-wide, platform independent standard for hypertext systems. With HTML we can create documentation that is accessible from all around the world through internet, easy to use and good-looking.

Current documentation tools need to be extended to support the production of documentation in HTML format. In this thesis we will describe how the language dependent parts of a documentation tool for the specification formalism ASF+SDF [BHK89, HHKR92, DHK96] can be reused to create a formatter that produces HTML code. This formatter can be used to typeset programs but can also be embedded in a more sophisticated documentation tool.

1.1 The project

To produce a program listing that is easy to understand by humans, we need a *prettyprinter*. A prettyprinter improves the readability of a program by adding layout, like linebreaks and indentations. When a program text is processed by a parser for that language, the parser produces an abstract syntax tree. This is a representation of that program where all syntactic structures have been identified and labeled. We can define a prettyprinter as a program that has an abstract syntax tree for some program in a language L as its input, and produces formatted output for a display or printing device (L $\rightarrow$ Display). This translation defines a prettyprinter that is specific for a source language and target device. However, if we introduce a general formalism that describes layout, we can decompose this process into a translation (L $\rightarrow$ B) from abstract syntax trees to this formalism, and a translation (B $\rightarrow$ Display) from this description to formatted output. With this approach, we can design formatters that are independent of the source language.

In this thesis we will present an executable ASF+SDF specification that translates Box expressions into HTML (Box $\rightarrow$ HTML). In this translation the Box language acts as the formalism for describing layout (B), while we use HTML to represent formatted output. The Box language has been designed as part of the Box tools [BV96], which currently consist of a formatter generator and a number of back-ends for producing human-readable output. The formatter generator drives a formatter for a programming language from its context-free grammar. This generated formatter will transform the abstract syntax tree of the program which is to be formatted into a Box expression, which is a declarative descriptions of the intended layout of a program. Back-ends translate Box to formatted output.

We have investigated how we can produce formatted output using HTML, and how this formatter can be integrated with the text and TeX formatters for Box to create sophisticated documentation tools. The main questions that we will try to answer are:

HTML Markup Are the markup capabilities of HTML powerful enough to produce typographically effective presentations of computer programs?

Hypertext linking Hypertext is a powerful concept for documenting computer programs. For example, we can link uses of functions and variables to their respective definitions. Can we generate these links from an abstract syntax tree?

Comments When creating both typeset and on-line documentation, we do not want them to be completely similar. For example, we want to have more elaborate comments in the typeset version while we want clickable links as comments in the hypertext version. How are we going to support this?

As an alternative way of producing documentation in HTML format, it is possible to use any of the current back-ends to produce text or TeX output and to use the result as input for a tool which converts this output to HTML. However, this requires yet another step in creating the final output and, more important, it does not produce the desired result. When we format our output as text, we loose information about fonts and alignments, while the output produced by Box to TeX and ToLaTeX [Vis95] is too complex to be successfully converted to HTML.

1.2 Outline

We start with an introduction of the used formalisms, e.g., the ASF+SDF Meta-Environment (Chapter 2), the Box language and tools (Chapter 3) and the HTML and SGML languages (Chapter 4). In Chapter 5, we present an initial specification for a Box to HTML formatter. In Chapter 6, we will discuss an improved version along with some case studies. Chapter 7 discusses related and future work. Finally, we present our conclusions in Chapter 8.

1.3 Acknowledgments

When I was first introduced to this project, it appealed to me by because of the clear practical applications of HTML along with the versatility of the box tools. Getting started took a lot of time. I had to refresh my knowledge of ASF+SDF and get to know the box tools, parts of which are not very well documented. At times, this was a confrontation with exactly the kind of problems this project is trying to address. I would like to thank my supervisor, Mark van den Brand, for his support during the entire project. Eelco Visser provided me with some helpful hints and remarks. Furthermore, I would like to thank my roommates and fellow students Harold Breebaart, Ahmed Ebrahim, Peter Manuel, Ramin Monajemi and Gert Vos. Finally, Eva Breedveld for keeping me at work...

Chapter 2

ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [Kli93] is an interactive system for developing language specifications and generating interactive programming environments from those specifications. The formalism that is used in the Meta-environment is ASF+SDF. It is a combination of the Algebraic Specification Formalism (ASF) [BHK89] and the Syntax Definition Formalism (SDF) [HHKR92]. ASF is an algebraic specification formalism that supports modularization. SDF has been developed to support the definition of lexical and context-free syntax for user defined syntax.

The Meta-environment generates both scanners and parsers from ASF+SDF specifications as well as term re-writers, thus allowing the execution of ASF+SDF specifications. Moreover, it generates syntax-directed editors for the specification languages. The system is able to perform several static semantic checks on the specifications, and supports the testing of specifications by executing them. It can be used to specify arbitrary (programming) languages by supporting the definitions of the syntax and semantics of a language.

2.1 Algebraic specification

An algebraic specification consists of a signature, a set of variables, and a set of equations. The signature describes sorts and functions over these sorts. In the equations variables and functions are used to construct terms and to define (possibly conditional) equations between those terms.

Suppose, for example, that we want to specify the classic Boolean data-structure. The signature in this case would define one sort `BOOL`; two constants of sort `BOOL`, `true` $\rightarrow$ `BOOL` and `false` $\rightarrow$ `BOOL`; one unary function `not: BOOL` $\rightarrow$ `BOOL`; and finally two binary functions, `and: BOOL # BOOL` $\rightarrow$ `BOOL` and `or: BOOL # BOOL` $\rightarrow$ `BOOL`. Suppose the variable *Bool* exists, then we can syntactically derive the following terms: `true`, `false`, *Bool*, `not(true)`, `not(`*Bool*`)`, `and(true, false)`, `or(not(false), and(true, false))`, and so on.

To evaluate these terms we have to add the semantics for these functions: the semantics of `not`, `and`, and `or` should be defined. The semantics can be defined by means of equations. An example equation that partially defines the semantics for the `and` function is `and(true,` *Bool*`) =` *Bool*. This algebraic specification can be transformed into a term rewriting system by interpreting the equations as rewrite rules from left to right. As a result, the semantics specification of the booleans can be used to compute the value of any (syntactically correct) term over the signature and to produce its normal form, either `true` or `false`, as answer.

2.2 The ASF+SDF formalism

The ASF+SDF formalism started out as ASF (*Algebraic Specification Formalism*) alone. SDF (*Syntax Definition Formalism*) was added shortly after to provide the specifier with more syntactical freedom: lexical, context-free, and abstract syntax are supported and incremental parsers can be generated from the syntax.

2.2.1 Lexical syntax

Lexical syntax is the definition of a low level structure of sentences (sequences of characters) in terms of lexical tokens. A lexical token is a pair consisting of a *sort name* and a *lexeme*. The former is used to distinguish classes of lexical tokens, the latter is the actual text of the token. For every lexical sort a *lexical constructor function* is created to provide access to the individual characters of that sort. A lexical syntax contains a set of declarations for *lexical functions*, each consisting of a regular expression and a result sort. For example

```
[a-z][a-z0-9]* → IDENTIFIER
```

defines `IDENTIFIER` to be built from one or more alpha-numerical characters, starting with a letter. Examples are `a`, `b1`, `counter` and `ref002`.

2.2.2 Context-free syntax

The context-free syntax describes the concrete and abstract syntactic structures of sentences in a language. It contains a set of declarations for context-free functions, each consisting of zero or more literals, sort names or lists followed by '→' and a result sort. They may be followed by attributes that control how parentheses and brackets affect the abstract syntax or by attributes that define the associativity of a rule. Priorities can be attributed to functions in order to settle ambiguities. All functions with the same result sort together define the alternatives for that sort. For example:

```
ELEMENT "∈" SET → BOOL
```

defines a function "∈" with arguments of sort `ELEMENT` and `SET` and a result sort `BOOL`.

List structures

List functions and list variables are an essential part of the ASF+SDF formalism. The list function uses two constructs, `*` and `+`, indicating zero or more repetitions and one or more repetitions respectively. Suppose we want to define the set. In the context-free syntax we would define the following list function:

```
"{" {ELEMENT ","}* "}" → SET
```

The comma is used as a separator between `ELEMENTS`. A set therefore is defined here as consisting of `ELEMENTS`, separated by commas and delimited by { and }. The non-terminal, the separator, and the delimiters can be chosen freely for list functions. The list notation is an abbreviation for the declaration of infinitely many functions each with a different number of arguments.

2.2.3 Equations

With equations a meaning may be added to functions declared in the context-free syntax. The equality of two terms is defined by an *unconditional equation* of the form:

```
[TagId] S = T
```

where S and T must be terms of the same sort. A *conditional equation* consists of a number of *conditions* and a conclusion. These equations can be written in two ways:

```
[TagId]  C1, C2, ...
         =============
             S = T

[TagId]  S = T when C1, C2, ...
```

where *C1, C2, ...* are conditions which may be either positive (and have the form S = T) or negative (and have the form S != T). We can also define 'default' equations, which should only apply when no other equation is successful, by giving them a tag of the form `[default-TagId]`.

2.3 Modules

An ASF+SDF specification is organized in *modules*. Each module defines syntax rules as well as semantic rules. The semantic (ASF) part uses the signature derived from the syntax (SDF) part. A module can import other modules in the specification. All visible entities of the imported module become available in the importing module. It is possible to hide entities in an imported module by using the *hiddens* keyword. Its inverse is the *exports* keyword. Below we present the keywords allowed in an ASF+SDF module:

- **imports**
  The names of imported modules are listed in this section.
- **exports**
  The items exported by the module are listed in this section. These items can be used by modules that import the module.
- **hiddens**
  The items that are hidden in the module are listed here. These items can not be used by importing modules.
- **sorts**
  Is followed by the sort names used in the lexical and context-free syntax sections.
- **lexical syntax**
  Followed by definitions of layout conventions and lexical tokens.
- **context-free syntax**
  Followed by context-free function declarations.
- **priorities**
  Priorities between context-free functions are listed here.

- **variables**
  Followed by declarations of variables used in the equations.
- **equations**
  Followed by the (conditional) equations of the module.

Sort declarations, lexical functions, context-free syntax, and variables must either be part of an `export` section or be declared as `hiddens`.

2.4 Example

In Figure 2.1 we present the Booleans example.

```
exports
   sorts BOOL

   context-free syntax
      true                 → BOOL
      false                → BOOL
      BOOL or BOOL         → BOOL {left}
      BOOL and BOOL        → BOOL {left}
      not "(" BOOL ")"     → BOOL
      "(" BOOL ")"         → BOOL {bracket}

   variables
      Bool[0-9']*          → BOOL

   priorities
      or < and


equations
   [1] true or Bool = true
   [2] false or Bool = Bool

   [3] true and Bool = Bool
   [4] false and Bool = false

   [5] not(false) = true
   [6] not(true) = false
```

Figure 2.1: Booleans in ASF+SDF

Some observations:

- The term attributes `{left}` and `{right}` are used for disambiguation; it defines a function to be either left or right associative. For instance, the attribute `{left}` in `BOOL and BOOL` → `BOOL` indicates that the term `true and false and true` should be interpreted as `(true and false) and true` rather than as `true and (false and true)`. In this case it is just for disambiguation: the normal forms of any term using the and function would be identical for either left or right. This is not the case however in the example 2^{3^4}, where clearly $2^{(3^4)}$ has a meaning different from $(2^3)^4$.

- The attribute {`bracket`} is added for grouping and disambiguation only, context-free rules with this attribute are not included in the abstract syntax tree.
- Declarations of variables are treated as declarations of lexical syntax. The `[0-9']*` notation indicates that all variables `Bool` immediately followed by any number of digits and/or quotes are now declared.
- The 'and' binds stronger than the 'or', since its priority is higher. This means that the term `true and false or true` should be parsed as `(true and false) or true`, rather than as `true and (false or true)`, which reduces to a different normal form.
- The numbers in front of the equations are labels and can be of arbitrary value (except for the 'default' prefix to indicate a default equation). They are useful when debugging the specification.

Chapter 3

Formatting in the ASF+SDF Meta-environment

Software documentation consists of (partial) listings of a program along with explanations of its goals and functionality. An important tool for the production of software documentation is a *prettyprinter.* A prettyprinter for a programming language is a program that rearranges the layout of a program listing by putting linebreaks in appropriate places, emphasizing keywords and indenting subconstructs. Pretty printers for programming languages relieve documentation writers from typesetting programs by hand and ensure that documentation is correct and up-to-date. They are also useful to visualize the structure of programs, which can be of great help when writing programs as well as for teaching and analyzing purposes. When embedded in a more sophisticated documentation tool, prettyprinters can be used to quickly produce accurate and legible documentation in different formats.

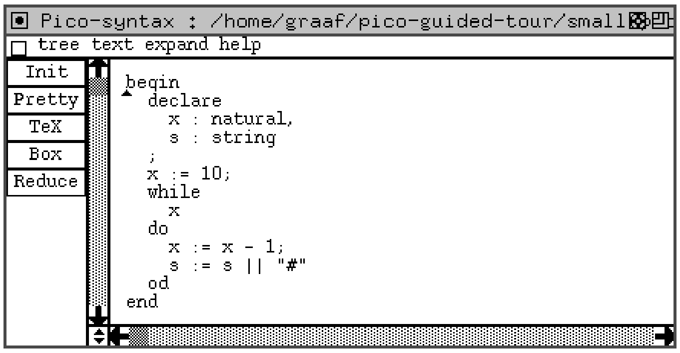
In Figure 3.1 a small program is shown to illustrate the prettyprinting process. The editor shown is the ASF+SDF term editor which supports reformatting of the program text during editing. An unformatted program (3.1a) is formatted using the `Pretty` button (3.1b). The TEX button creates typeset output (3.1c).

We can define a prettyprinter as a program that has an abstract syntax tree of some language L as its input, and produces formatted output for a display or printing device (L $\rightarrow$ Display). Historically, the term *prettyprinter* refers to a program that produces plain text while *formatter* is more commonly used for tools that create typeset output. From now on, we will use *formatter* when referring to formatting tools in general.

3.1 The Box project

Current formatting technology has a number of problems. For many programming languages no formatters are available, and these have to be implemented from scratch. This is a time-consuming activity, especially for a language under development. Furthermore, most formatters are geared towards one text processing or typesetting system and can not be easily connected with other systems. The Box project [BV96] aims at simplifying this process by offering a set of tools for creating formatters for arbitrary programming languages. It offers a modular setup that makes formatters independent from both source language and target device. To achieve this, the formatting process is decomposed in a translation (L $\rightarrow$ Box) from abstract syntax trees to Box expressions and a translation (Box $\rightarrow$ Display) from Box expressions to an output format. Box expressions are declarative descriptions of the intended layout of a program. In Section 3.3 we

(a) unformatted



(b) formatted

begin
    declare
        x : **natural**,
        s : **string**
    ;
    x := 10;
    while
        x
    do
        x := x - 1;
        s := s || "#"
    od
end

(c) typeset

Figure 3.1: This example shows the ASF+SDF structure editor with an unformatted (a) and formatted (b) Pico program and typeset output (c) created by TEX.

will introduce the Box language and tools but we will first have a look at the set of formatting tools currently available in the ASF+SDF Meta-Environment.

3.2 Formatting tools

The current ASF+SDF Meta-Environment provides two tools which are based on a number of hard-wired heuristic rules. They are used to unparse the results of the term-rewriter and to substitute templates during syntax directed editing, as well as for creating LaTeX code.

- **Default Prettyprinter**
  The default prettyprinter is written in LeLisp, and can be used to prettyprint terms. The old version used PPML [MCC86] as an intermediate presentation, the new version maps an ASF+SDF term to a term over Box.

- **ToLaTeX**
  The ToLaTeX tool [Vis95] typesets ASF+SDF specifications. Like the default prettyprinter, it is written in LeLisp. It translates the abstract syntax trees which represent an ASF+SDF module directly into LaTeX code.

The main drawback of these formatters is that they can not be changed in a flexible way, unless the LeLisp code is adapted. The need to adapt the default prettyprinter led to the development of the formatter generator, and a new generation of other tools built around the Box language, which we will introduce in the next section.

- **Formatter generator**
  The formatter generator is a system that produces a language specific formatter from its SDF definition. The formatter itself is an ASF+SDF specification which maps a term in the language to a Box term. It has been used to produce formatters for PSF [Bru95], μCRL [Hil96] and COBOL [Res94].

- **AsFix to Box: ToBox**
  The development of the formalism AsFix yielded another application of this technology, e.g., an AsFix to Box formatter which can produce the original ASF+SDF specification given its AsFix representation. A first attempt to redesign the original Asfix to Box formatter to make it more flexible is called ToBox [Bar96].

3.3 The Box tools

The project started with an initial specification of the Box language [Vos90] and has since then been extended with formatters for text and TeX [BV94, BV95] and a formatter generator [BV96]. Figure 3.2 shows the architecture of the current system.

- **The formatter generator**

  Given a context free grammar of a language L the formatter generator generates a formatter for L. This generated formatter is an ASF+SDF specification. It traverses an abstract syntax tree of an L program and produces a box term. As a separate part, the formatter contains functions to restore comments in the original text by comparing it with the generated box term.

- **The Box language**

  A box expression is a declarative, language-independent description of the layout of programs.

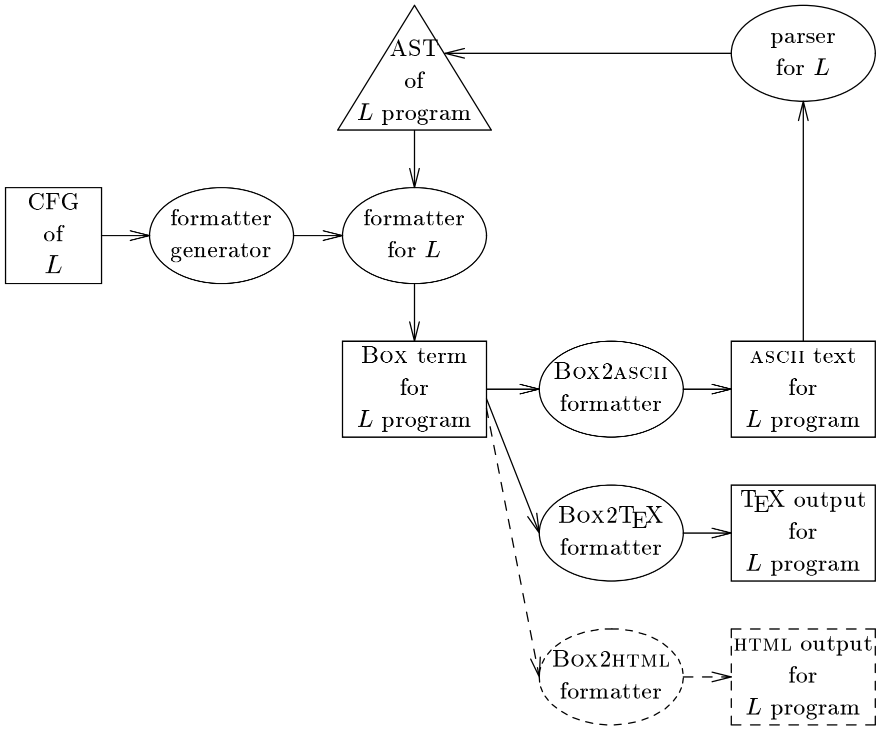


Figure 3.2: Architecture of the Box tools.

- **The formatting tools**

  The back-ends do the hard work of producing a legible text. These tools translate the language-independent box expression into ASCII text, TEX code or other formats.

3.4 BOX: A Box language

Box is a special purpose language for formatting. The most elementary boxes are strings of characters enclosed in double quotes. Boxes are composed by operators, which define the relative positioning of the boxes. We distinguish six basic operators: H (horizontal composition), V (vertical composition), HV (horizontal and/or vertical composition), HOV (horizontal or vertical composition), I (Indentation) and WD (invisible box with the same width as some visible box). A box term can be parameterized with *space options* to control the amount of layout between the boxes. The horizontal (hs), vertical (vs), and indentation (is) offset between boxes can be adapted.

H [B_1 ... B_n] This is the horizontal composition operator. All sub-boxes are aligned next to each other. If the result is too wide for the screen, boxes extending past the right margin will be invisible. This operator may have one option hs to adjust the horizontal space between boxes.

$$\mathsf{H}[\ \boxed{B_1}\ \boxed{B_2}\ ] = \boxed{B_1}\boxed{B_2}$$

V $[B_1 \ldots B_n]$ Vertical composition. All subboxes are placed below each other in the same column. Accepts one option *vs* to adjust the vertical space between boxes.

$$\mathsf{V}[\ \boxed{B_1}\boxed{B_2}\ ] \;=\; \begin{array}{c}\boxed{B_1}\\ \boxed{B_2}\end{array}$$

HV $[B_1 \ldots B_n]$ We can interpret this as horizontal *and* vertical composition. This operator aligns boxes according to the margins of the page. Subboxes are aligned horizontically until the right margin is encountered. The next subbox is printed on a new line, starting at the column in which the first box started. This operator may have the options *hs*, *vs* and *is*. The last one is used to change the column in which the next box is printed if it is printed on a new line. We can express the formatting in terms of H and V operators.

$$\begin{array}{rcl}\mathsf{HV}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] & = & \mathsf{H}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] \text{ or}\\ & & \mathsf{V}[\ \mathsf{H}[\ \boxed{B_1}\boxed{B_2}\ ]\ \boxed{B_3}\ ] \text{ or}\\ & & \mathsf{V}[\ \boxed{B_1}\ \mathsf{H}[\ \boxed{B_2}\boxed{B_3}\ ]\ ] \text{ or}\\ & & \mathsf{V}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ]\end{array}$$

HOV $[B_1 \ldots B_n]$ This operator will only place its subboxes horizontically if they all fit on the same line. If this is not possible, it will act as a V operator. Accepts both *hs* and *vs* options.

$$\begin{array}{rcl}\mathsf{HOV}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] & = & \mathsf{H}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ] \text{ or}\\ & & \mathsf{V}[\ \boxed{B_1}\boxed{B_2}\boxed{B_3}\ ]\end{array}$$

I $[B]$ Indents all of the subboxes. This operator may have one option *is* to adjust the indentation offset. The default value for *is* is 2.

$$\mathsf{V}[\ \boxed{B_1}\ \mathsf{I}[\ \boxed{B_2}\ ]\ \boxed{B_3}\ ] \;=\; \begin{array}{l}\boxed{B_1}\\ \quad\boxed{B_2}\\ \boxed{B_3}\end{array}$$

WD $[B]$ Yields an invisible box with the same width as its argument.

$$\mathsf{V}[\ \boxed{B_1}\ \mathsf{H}[\ \mathsf{WD}[\ \boxed{B_1}\ ]\ \boxed{B_2}\ ]\ ] \;=\; \begin{array}{ll}\boxed{B_1} & \\ & \boxed{B_2}\end{array}$$

The box language can be easily extended with new operators. Standard extensions of the box language have been designed to support, for example, *alignments*, *fonts*, and *crossreferences*. Here are a few examples.

F ***fn=KWf*** $[B]$ This is the font operator. The option (fn) identifies the argument as a keyword (KWf), which will be emphasized in print. Other options are available to control, for example, the size of a font.

A [B] This is the alignment operator, which we can use to produce aligned columns. Its argument is a list of R boxes (R[B]) which represent each row. All R boxes should have the same number of arguments, equal to the desired number of columns.

O [B_1 *STRING* B_2] This operator was introduced for typesetting conditional equations in ASF+SDF specifications (O stands for 'over'). The boxes B_1 (conditions) and B_2 (conclusion) are placed vertically but will be separated by a horizontal bar, as specified by the (first character of the) string. The length of this bar depends on the number of characters in the conditional part of the equation.

LBL [*STRING* B] With this operator we can attach a label to a specific box.

REF [*STRING* B] With this operator we can refer to a label defined somewhere else.

3.4.1 Box syntax

A box term consists of a box operator, zero or more spacing options, and is followed by zero or more boxes. Here is an example of a box term:
V ["Warning:" H *h*s=2 ["This" "text:" HV ["was" "boxed"]]]
A valid interpretation for this term could be:

```
Warning:
This␣␣text:␣␣was␣boxed
```

The kernel box language is simple but can be easily extended. A specification writer can easily define a new module which imports this kernel box language. For each extension the back-ends have to be adapted, but because of the strong modularity of ASF+SDF this can be done in a clean manner.

3.5 The formatter generator

The formatter generator produces a front-end for the box tools for a specific language from its context-free grammar. This generated formatter translates abstract syntax trees into box expressions. A formatter is produced by generating a prettyprinting function for each non-terminal in the context-free language. Formatting decisions are made by classifying each context-free grammar rule and applying preset formatting rules for each class. Because the generated formatter itself is an ASF+SDF specification, it can easily be adapted. For a complete discussion of this tool see [BV96].

3.5.1 Restoring comments

Comments in source code are usually treated as layout. When a parser processes a program text and builds an abstract syntax tree for it, comments are discarded. A text formatter processes this abstract syntax tree to produce text again. The difference between the original text and the produced text is layout and comments.

To solve this problem, the formatters produced by the formatter generator have been extended with an algorithm for restoring comments. These extended formatters consist of two parts: the function to transform an abstract syntax tree into a box expression and the functions which take the original text together with the constructed box expression and insert the comments in the text in the box expression. In this way, the comments are restored although they are not present in the abstract syntax tree. The position of a comment in the original text is used to determine its position in the formatted text. The box language has been extended with three operators, which contain the begin ($STRING_1$) and end ($STRING_2$) marker of the comment, and the comment text (B). Note that the text of the comments is formatted as well.

PAR $STRING_1$ $STRING_2$ $[B]$ A comment that starts in the first column of a new line.

VPAR $STRING_1$ $STRING_2$ $[B]$ A comment that occurred on a line which no other symbols, but not in the first column.

HPAR $STRING_1$ $STRING_2$ $[B]$ A comment which occurred after some symbol

3.6 The formatters

When we started work on this project, back-ends where available for text and TEX formatting. The essential part of the formatting tools is the translation from boxes to a specific output medium. The interpretation of boxes depends on the output medium, for example, a font operator has no effect when translating to plain ASCII text. For a complete listing of the specifications of Box and the formatters, see [BV95].

3.6.1 Box to Text

This back-end converts box to plain text. It implements all the basic semantics for the box language in ASF+SDF. There are two versions of the Box to Text back-end: A small version implemented by Eelco Visser (Chapter 5 in [BV95]) and an extended version by Mark van den Brand (Chapter 6 in [BV95]). Both versions transform a box term into lists of strings, which represent the formatted text. A string (`BOX-STRING`) is defined as a list of arbitrary characters except for a newline enclosed in double quotes. A text is simply a list of strings, where each string denotes a line on a display (`BOX-STRING* -> TEXT`).

Box to Text (Algebraic Style)

The small, algebraic version currently does not support box extensions, like alignments. The main module is small because it uses powerful text composition operators (module 'Text Operations'). These operators define a number of basic operations on strings and lists of strings.

For example, the '//' operator adds lines to a text and the '⊣' operator prefixes each line of a text with a string. This turns out to be an important operation for text formatting. For example, we can use it to prefix each line of a text with a *blank* string, thus creating an indentation.

```
context-free syntax
    TEXT   "//" TEXT      -> TEXT {right}
    BOX-STRING "-|" TEXT  -> TEXT
```

In the following equations, `a` and `b` are variables of type `BOX-STRING`, `a*` and `b*` are variables of type `BOX-STRING*` and '`*`' denotes string concatenation.

```
equations
   [1]  a* // b*  = a* b*
   [2]  a -|      =
   [3]  a -| b b* = a * b // a -| b*
```

These operators are used to construct more complex ones, like the '||' operator for horizontal composition. This a simple job when the arguments are strings, but it becomes difficult when the arguments are lists of strings. In that case, the current indentation level has to be maintained. The '||' is also defined for conditional composition. For example, the left looking horizontal composition operator '_||' only sets its second argument when the first (left) argument is not empty. Let us take a look at how these operators are used in the main module to specify the H operator.

```
context-free syntax
    BOX "(" CONTEXT "," BOX-INT ")"     -> TEXT

equations
   [B2T'H]  E(H, m) = T, hwidth(T) = s,  h = H(o*)
            ===================================================
            Ho*[E B*](O,m) = T _|| hskip(h) ||_ Ho*[B*](H, m - s - h)
```

A box term is suffixed with context (H or V) information, and an integer which keeps track of the remaining space on this line of text. If we look at the conclusion of the equation, we can see that the result of evaluating the first argument box (T) will be followed with space-options (hskip) and with the result of evaluating any remaining argument boxes. The calculated width of the first argument box and any space-options that have been applied is subtracted to obtain the remaining number of spaces on this line.

For example, suppose we evaluate the box term `H["arg1" V["arg2" "arg3"]]`. Evaluating the V box results in the text `"arg2" "arg3"`. Evaluating the H box adds `"arg1"` to the first line of the second text, resulting in `"arg1 arg2"`. The second line is prefixed with a blank string with the same length as the first argument (plus space-options), resulting in `"␣␣␣␣␣arg3"`. In this way, the indentation level is maintained.

Box to Text

This is a voluminous version that has support for extensions like alignments and fonts. It has been developed separately from the small version and is more efficiently executable. It uses a sort 'FRAME' to store all parameters, like the current (x,y) position on the page (POINT), as well as margins (left and right) and space options (DEFS). When a box term is parsed these parameters are adjusted for each box operator. When a string is encountered it is added to the output according to the values which are effective at that point.

```
context-free syntax
    "(" BOX-INT "," BOX-INT ")"                                  -> POINT
    BOX-INT "," BOX-INT "," BOX-INT "," BOX-INT "," BOX-INT -> DEFS
    POINT "&" DEFS                                               -> FRAME
```

The main work is performed by a number of 'place' functions, one for each basic box operator. The datatype TEXT is extended with POINT which holds the position information of the last character in the text built so far.

```
context-free syntax
    TEXT "&&" POINT                                          -> TEXT-STRUCT
    place BOX-LIST with FRAME and TEXT-STRUCT and CONTEXT -> TEXT-STRUCT
    hplace BOX-LIST h BOX-INT with FRAME and TEXT-STRUCT  -> TEXT-STRUCT
```

We will take a look at how H boxes with multiple arguments are handled. The following equation calls 'hplace' and also extracts the effective value for the horizontal space-option (hs) from the frame.

```
[BtT2'7]   set hs value given so* and Frame = Hs
           =====================================
           place H so* [B+ B] with Frame and Struct and 'C =
             hplace B+ B h Hs with Frame and Struct
```

The only important work for the next equation, is to call the function `hsep` which takes the proposed next coordinate (`Point1`), the effective space-option (`Hs`), the frame containing the previous coordinate and margin information (`Frame1`), and calculates the next coordinate (`Frame2`).

```
[BtT3'2]   hplace B+ h Hs with Frame1 and Struct1 = Text1 && Point1,
            hsep Point1 h Hs with Frame1 = Frame2
           =====================================
           hplace B+ B h Hs with Frame1 and Struct1 =
            place B with Frame2 and Text1 && Point1 and H
```

Single argument boxes are handed back to the function `place`, which calls the appropriate function for that box operator. Finally, only strings are left. The following equation, for example, adds a string to the text constructed so far when the previous and next coordinate are on the same line (the Y-coordinates are equal). The necessary number of layout chars is calculated from the two coordinates and also inserted, finally, the current coordinate is incremented with the length of the inserted string.

```
[BtT2'1]   deslash(String1) = String2,
           add to last line (Text1,
                        string of X1 - X2 layoutchars : String2) = Text2
           ============================================================
           place String1 with (X1,Y) & Defs and Text1 && (X2,Y) and 'C =
             Text2 && (X1 + length of String2,Y)
```

3.6.2 Box to TEX

The Box to TEX formatter [BV94] uses a different approach. Since TEX is a Box-language itself, a straightforward translation can be made from box expressions into TEX -code. For each box operator a corresponding TEX command has been defined. These macro definitions perform most of the work so the ASF+SDF specification for Box to TEX looks rather simple. For example, here is (just one more...) equation for the H operator.

```
[TeX'H]    tex(Ho*[B*])     = (\H {texs2((\HSKIP{H(o*)}), B*)})
```

We will not discuss the implementation of the `\H` macro. The function `texs2` separates each argument box with a piece of code. In this case, all argument boxes of the H operator are separated with a `\HSKIP` macro which inserts spacing. To overcome the limited nesting capability of TEX this formatter uses a number of laws to reduce the nesting depth. These laws are defined in the module 'Box-Laws' which also performs some other transformations on a box term before it is fed to the Box to TEX module.

Chapter 4

An introduction to HTML and SGML

The HyperText Markup Language (HTML) is a standard for writing electronic hypertext documents. It has been developed at CERN as an essential part of the World Wide Web (WWW) project, which has been responsible for the exploding popularity of the internet in recent years. Recently, the development of HTML has moved to the WWW consortium (W3C)[1]. This consortium is hosted by MIT, INRIA and CERN and funded by industrial members, in order to continue the development of common standards for the evolution of the WWW.

The term hypertext has been introduced as early as the 1950s. We can define hypertext as 'human-readable information linked together in an unconstrained way' [BL89]. In this way, various kinds of information can be accessed as a web of nodes in which the user can browse at will. Potentially, hypertext provides a single user-interface to many large classes of stored information such as reports, notes, databases, computer documentation and on-line systems help.

To achieve a simple document format for information exchange on virtually any platform, HTML was implemented according to the Standard Generalized Markup Language (SGML) standard. In the next section we will present a brief introduction to SGML. See [Gol90] for a complete reference.

4.1 SGML

SGML, in its present form, is the result of the efforts of many people, channeled into four major activities that occurred over the past twenty years: generic coding, the GML and SGML languages, the SGML standard, and major SGML applications.

Historically, electronic manuscripts contained control codes or macros that caused the document to be formatted in a particular way ('specific coding or procedural markup'). In contrast, generic coding, which began in the late 1960s, uses descriptive tags (for example, 'heading', rather than 'format-17').

SGML started out as GML, the Generalized Markup Language, created by Goldfarb, Mosher and Lorie at IBM in 1969. GML was based on generic coding ideas and introduced the concept of a formally-defined document type with an explicit nested element structure. In 1978, the American National Standards Institute (ANSI) established a committee for the development of

[1]http://www.w3.org/

a text description language standard. GML became the basis for this standard with aid from a project predating GML called GenCode. The first working draft of the SGML standard was published in 1980. The final text was published in 1986 [ISO86].

SGML is a meta-language. It defines a framework in which markup languages can be defined for different purposes. To distinguish particular markup languages SGML introduces the notion of a 'document type' and hence a Document Type Definition (DTD), which contains the formal specification of the syntax.

4.1.1 Markup languages

We can define markup as any means of making the interpretation of a text explicit [SMB94]. By markup language we mean a set of markup conventions used together for encoding texts. A markup language must specify what markup is allowed, what markup is required, how markup is to be distinguished from text, and what the markup means. SGML provides the means for doing the first three, it defines the *syntax* of a markup-language. The last requirement *(semantics)* is outside the scope of SGML because it is application dependent.

4.1.2 SGML structures

The technical term used in the SGML standard for a textual unit, viewed as structural component, is *element*. To distinguish elements we need to mark them in some way. The standard provides for a variety of different ways of doing this, the most commonly used method is to insert a tag at the beginning of the element *(a start-tag)* and another at its end *(an end-tag)*. Elements can contain *attributes* to describe additional information. The *content model* of each element describes what element occurrences can be contained in its context. Content is specified either in terms of other elements or using reserved words, of which the most commonly encountered is `'#PCDATA'`. This is an abbreviation for 'parsed character data', and it means that the element being defined may contain any valid character data. Last but not least, SGML provides *entities* for encoding and naming parts of a document. An entity might be a string of characters, a special character or even a whole file of text. To include it in a document, we use an *entity reference*. Figure 4.1 shows how entities are commonly used in a DTD. The example shows the definition for a heading element with an optional alignment attribute.

```
<!ENTITY % heading "H1|H2|H3|H4|H5|H6">
<!ELEMENT ( %heading ) - - (#PCDATA)>
<!ATTLIST ( %heading ) align (left|center|right|justify) #IMPLIED>>
```

Figure 4.1: Example definition of a heading element for HTML

SGML emphasizes descriptive rather than procedural markup. A descriptive markup system uses markup codes which simply provide names to categorize parts of a document, like `<H1>` for 'level 1 heading' or `<OL>` for 'start of ordered list'. By contrast, a procedural markup system defines what processing is to be carried out at a particular point like 'font size = 12 pt' or 'indent = 2 cm'. In SGML the instructions needed to process a document are kept outside the document in separate procedures or programs. With descriptive markup the same document can be processed by many different pieces of software, each of which can apply different processing instructions to those parts of it which are considered relevant.

4.1.3 SGML documents

Every SGML document consists of three parts:

- *SGML declaration*
  Specifies basic facts about the dialect of SGML being used such as the character set, the codes used for SGML delimiters and the length of identifiers. For example, the SGML declaration for HTML specifies that the character that opens a start tag is < and the maximum length of a name is 72 characters.

- *Document Type Declaration (DTD)*
  Specifies the type, relationships, attributes, and content model for all elements. For example, consider the HTML element `<A HREF="http://host/dir/file.html">this is a link</A>`, where A is the element name, HREF is an attribute name, `http://host/dir/file.html` is the attribute value and '`this is a link`' is the content.

- *Document instance*
  The document itself, which contains the data and markup of the document.

The SGML declaration and DTD are called the SGML *prologue* and specify the syntax against which the document instance is to be validated. For a major SGML application, like HTML, the prologue is 'hard-wired' in the software used and is invisible to the user of these applications.

4.2 HTML

The history of the Hypertext Markup Language (HTML) starts in 1989 with a project proposal at CERN, to be known as the World Wide Web. Goal of the project was to design a system that allowed people to work together by combining their knowledge in a web of hypertext documents. Proposer of this project was Tim Berners-Lee, who is currently chairman of the WWW consortium. He designed the first WWW server, client and hypertext browser/editor which were made available within CERN in December 1990 and on the internet at large in the summer of 1991. Through 1991 to 1993, Tim Berners-Lee continued coordinating the project and designed the initial specifications of HTML and related standards[2].

Work on HTML has been going on since then, resulting in the HTML 2.0 Proposed Standard by the HTML Working Group of the internet Engineering Task Force (IETF) in November 1995 [BLe95]. Further work resulted in the HTML 3.0 Draft at W3C, which added more sophisticated tables, text flow and mathematics. Unfortunately, this proposal proved to be too different from directions already introduced by browser manufacturers, like Netscape and Microsoft, which had already started implementing some of these features on their own. Currently, HTML is at version 3.2 which is a recommendation developed in cooperation with the industry. This path will hopefully ensure that the original intention, a platform independent system, will remain valid.

In the next part of this section, we will give a brief introduction to HTML. We will not cover the complete HTML syntax, but only those elements which are relevant for our project. The most recent drafts and proposals for the development of HTML are available from W3C[3].

[2]This includes the HTTP (HyperText Transfer Protocol) standard for transferring HTML documents over the internet and URLs (Uniform Resource Locators), a standard way of referring to resources on the internet.

[3]http://www.w3.org/MarkUp/

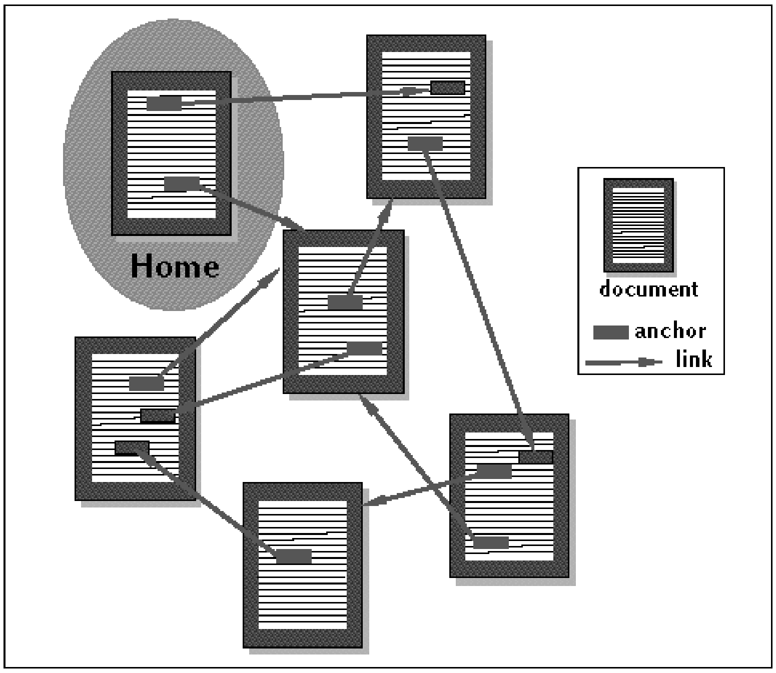


Figure 4.2: A basic hypertext structure

4.2.1 HTML elements

In HTML documents, tags define the start and end of headings, paragraphs, lists, character highlighting and links. Most HTML elements are identified in a document as a start tag, which gives the element name and attributes, followed by the content, followed by the end tag. Start tags are delimited by < and >, and end tags are delimited by < / and >. For example:

```
<H1>This is a Heading</H1>
```

Some elements appear as just a start tag. For example, to create a line break, we use
. Additionally, the end tags of some other elements (e.g. P, LI, DT, DD) can be omitted as the position of the end tag is clearly implied by the context.

```
<P>This is a paragraph.
<P>This is the next paragraph.
```

Most elements can contain attributes. For example, to center a heading on the page:

```
<H1 ALIGN=CENTER>This is a centered heading</H1>
```

Some elements require a specific content. For example, list elements (e.g. DL, UL, OL) require <LI> elements in their content to separate each list item. This example also shows how comments in HTML should be formatted.

```
<UL>      <!-- Start of unordered list -->
<LI>Item 1
<LI>Item 2
</UL>     <!-- End of unordered list -->
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<!-- Here is a good place to put a comment. -->
<HEAD>
<TITLE>Structural Example</TITLE>
</HEAD>
<BODY>
<H1>First Header</H1>
<P>Some text in the first paragraph.
</BODY>
</HTML>
```

Figure 4.3: Example HTML document

Reserved characters can be included in a HTML document by using a reference to a *text entity*. For example, to include the < sign in a text we refer to `&lt`. To include and ampersand itself we use `&amp`. More special characters can be invoked by referring to a numeric entity (e.g. `&#60`) from a standard character set, like the ISO Latin 1 character set.

4.2.2 HTML documents

Figure 4.3 shows the formal structure of a HTML document. To identify the text as an HTML document it has to start with a reference to the SGML DTD of the HTML version it complies with. For HTML version 2.0, this the `DOCTYPE` entry as shown in the example[4].

The head element contains specific elements to describe the document, which will not appear as part of the document text. The only one required is `<TITLE>` for the document title. Other allowed elements include `<META>` for providing information like author name, keywords and expiration date for the document. The body element contains the actual text and markup of the document.

4.2.3 Hypertext elements

This is where the 'H' from HTML comes in. To define a link to another HTML document (and many other kinds of information) HTML provides the anchor element `<A>`. This enables us to create a hypertext structure. Figure 4.2 shows what a basic hypertext structure looks like. The anchor element comes in a lot of flavors, depending on what we want it to do. The NAME attribute turns the anchor into the target of a link, by labeling it with a unique name. For example:

```
<A NAME="label1"> ... </A>
```

The HREF attribute turns the element into a hypertext link. The destination for the link is designated by the value of the HREF attribute, which is expressed in URL[5] notation. This is typically a WWW address, filename or target location defined by the NAME attribute. For example:

[4]However, the SGML prologue is hard-coded in current browsers so this tag does not affect the actual formatting of a document

[5]Uniform Resource Locater. A string that identifies a resource, typically on the internet. The structure of an URL may be expressed in general as: Resourcetype://host.domain/pathname

```
<A HREF="http://www.host.domain/file.html> ... </A>
<A HREF="#label1"> ... </A>
```

The first one links to a specific file as described by the WWW address. The second one links to a location in the current document as defined by the label.

4.2.4 Other HTML elements

The HTML standard includes many more elements. Relevant elements for our project are those for text-style (e.g. `<B>` for boldface, `<EM>` for emphasis and `<FONT>` for font properties) and the `<TABLE>` element, which enables us to create simple tables. To define its contents, the `<TABLE>` element accepts `<TR>` (table row) and `<TD>` (table data) elements.

There are also elements which are not part of the standard, but have been introduced by browser manufacturers. An example of this is the `<FRAME>` element, an extension introduced by Netscape in Navigator 2.0. It is used for creating multiple windows for viewing a document. Another example is the `<NOBR>` element which prevents linebreaking.

For a complete specification of HTML 3.2 elements, see [Rag97]. For a complete reference of all elements used in our specification, see Appendix B.

4.3 HTML as formatter

HTML is intended as a common medium for tying together information from widely different sources. It was also intended to be independent from a specific output medium. Because of this HTML does not offer sophisticated typesetting capabilities. Furthermore, the precise *semantics* of HTML are defined by the application that we use to process it, so we can not be sure our output will always look the same. To complicate the problem, some browser manufacturers do not completely implement the HTML DTD and some have added special tags which are not in the DTD. In this section we will give some examples of inconsistencies we have found between different browsers, as well as often requested features which are not present in the standard. Note that we only cover the elements that are relevant for our project.

- White space and line breaking
  1. A browser normally treats all white space that is not a blank (e.g. tabs, end-of-line characters etc.) as a single blank, and collapses multiple white space to a single blank (similar to the way TEX handles this). However, unlike TEX HTML currently provides no way for producing whitespace when we do need it. The HTML 3.0 proposed standard introduced a non-collapsible space (`&sp;`) but this has not been implemented. As an alternative HTML offers the `<PRE>` element for preformatted text (similar to the `verbatim` environment in TEX).
  2. Browsers are inconsistent in whether they collapse multiple line breaks (such as defined by multiple `<BR>` or `<P>` elements) into a single line break. A strict interpretation of the standard would seem to imply that they should, but most of them do not collapse multiple line breaks.
  3. To prevent linebreaking the 2.0 standard provides the NOWRAP attribute, for example `<H1 NOWRAP>`. However, many browsers do not implement this, and this attribute has been dropped in HTML 3.2 (except for table cells).
- Tabbing and indenting

1. A tabbing environment was proposed by the expired HTML 3.0 standard, but has been dropped in HTML 3.2. Currently no browser implements a tabbing environment.
2. Indenting of margins and paragraphs is not implemented.

4.3.1 HTML formatting solutions

In this section we will present a few 'solutions' which are commonly used to overcome some of the limitations of HTML from the previous section. Some of these solutions involve the 'creative' use of HTML elements in order to produce a particular layout effect. We have to be well aware of this, because this use is not according to the HTML standard. However, as long as there are no alternatives, some of these tricks are evolving into standards because of their widespread use.

- *Creating whitespace*
  The non-breaking space character `(  or  )` produces spaces which are non-breakable. Most browsers also do not collapse this 'character', and therefore it is used for creating whitespace.

- *Linebreaking*
  To implement the `H` operator, we need some way to prevent linebreaking. Netscape has introduced the `<NOBR>` element to achieve this. Unfortunately, this element is not part of the standard and it is therefore not recognized by all other browsers. However, there is no alternative solution at this moment.

- *Indenting*
  To achieve indenting in a HTML document we can use the `<BLOCKQUOTE>` element. On most browsers the contents of this element is indented from both margins. Alternatively, we can use the `<UL>` element (start of unordered list) which results in an indentation of the left margin only. We can also nest these elements to create multiple levels of indentation. Although these elements where not intended for this effect, their application for this purpose is even suggested by commercial HTML editors (e.g. Claris HomePage and Netscape Gold Editor).

4.3.2 Cascading Style Sheets

The introduction of browser-specific elements has not supported the ongoing development of the HTML standard. Hopefully, the current commitment of the industry to cooperate through the WWW consortium will ensure the future development of the standard. The HTML 3.2 reference standard [Rag97] aims to capture current recommended practice and offers a replacement for HTML 2.0, but it still does not offer solutions for most of the missing features we have described. One of the interesting proposals developed at W3C for improving this situation is the concept of Cascading Style Sheets [WLB96]. With style sheets a HTML designer can customize the presentation of the various HTML elements, for example, by defining a specific fontsize and color for each heading element. These style rules are kept in a separate document which can be invoked from a HTML document. Style sheets also promise a solution for many of the current HTML problems, for example, by introducing indentation as a style property which can be assigned to HTML elements. The current proposal also introduces properties for setting the margins of a document.

The CSS proposal introduces a formalism for specifying procedural markup. Because this information is kept in a separate file, HTML itself will remain a descriptive markup language. Another feature of CSS is that style sheets cascade; authors can attach a preferred style sheet, while the reader may have a personal style sheet to adjust for human or technological handicaps. The rules for resolving conflicts between different style sheets are defined in the specification.

For example, a simple CSS rule looks like this: `H1 {font-family: helvetica}`. This declares helvetica as the preferred font for all `<H1>` elements. The rule consists of a selector (`'H1'`) and a declaration (`'font-family: helvetica'`). All HTML tags are possible selectors. A declaration consists of a property and a value. The list of properties and possible values is defined by the CSS specification. Properties that accept a length value, like `margin` for setting margins in a document, can be set both absolute and relative to another property with very accurate measurements (e.g. cm, mm, pixels). This will introduce a new level of typographic control for HTML documents.

Some properties will introduce alternative, and possibly better, ways to implement some of the Box operators. For example, properties have been recommended for creating indented text (`P {text-indent: 2 cm}`), controlling whitespace (`P {white-space: pre}`) and spacing between words (`P {word-spacing: 4 mm}`). Properties for font control (`BODY {font-family: helvetica, font-size: 14 pt}`) and font color (`EM {color: red}`) will solve the browser-specific implementations of these futures.

Anyhow, style-sheets will improve the formatting capabilities of HTML in a dramatic way. However, the only major browser which currently implements style sheets is Microsoft's internet Explorer. Netscape has announced to implement style sheets in Navigator 4.0.

4.4 Html

We present a syntax definition for HTML. The context-free syntax part has been kept simple. This is a useful approach because we only want to generate code.

Tags and entitity references are defined as lexical syntax. They are made up of an identifier (the name of the element or entity) and delimiters. Identifiers can be constructed with both upper- and lowercase characters. Because using uppercase characters makes it easier to discriminate tags from text we will not use lowercase characters in tags we generate. With this generic syntax, we provide for both existing and future tags. To allow SGML definitions, including comments, in a HTML document we also include a generic syntax for them.

```
imports Layout Box-Strings
exports
  sorts ID STARTTAG ENDTAG ENTITYREF SGMLTAG ATTRIBUTE
        HTEXT
  lexical syntax
    [A-Za-z]+              → ID
    "<" ID ">"             → STARTTAG
    "</" ID ">"            → ENDTAG
    "&" ID ";"             → ENTITYREF
    "&#" BOX-INT+ ";"      → ENTITYREF
    "<!" ~[\n]* ">"        → SGMLTAG
```

The context-free syntax defines tags with attributes. Box-strings are used to represent the actual text. The sort HTEXT is used to represent all possible elements in a HTML document, both markup and text.

```
  context-free syntax
    ID "=" BOX-STRING      → ATTRIBUTE
    ID "=" ID              → ATTRIBUTE
```

ID "=" BOX-INT → ATTRIBUTE
"<" ID ATTRIBUTE+ ">" → STARTTAG
BOX-STRING → HTEXT
STARTTAG → HTEXT
ENDTAG → HTEXT
SGMLTAG → HTEXT
ENTITYREF → HTEXT
"(" HTEXT* ")" → HTEXT

variables

*[ab][0-9]** → BOX-STRING
*[t][0-9′]** → HTEXT
[t]""[0-9′]** → HTEXT*

equations

Lists are flattened here. Any nested list is reduced to a flat list.

[1] $(t_1^* \; (t_2^*) \; t_3^*) = (t_1^* \; t_2^* \; t_3^*)$

4.4.1 Html-Op

Several utility functions are defined for manipulating and generating hypertext.

imports Html[4.4]

exports

context-free syntax

createdoc(HTEXT) → HTEXT
rmquotes(BOX-STRING) → HTEXT
hblank(BOX-INT) → HTEXT
hblank2(BOX-INT) → HTEXT
vblank(BOX-INT) → HTEXT
specials(BOX-STRING) → BOX-STRING

hiddens

variables

int → BOX-INT
[ab] → BOX-STRING
"*chr*" → CHAR
"*chr+*" → CHAR+
"*chr**" → CHAR*

equations

The function createdoc generates the preamble for a HTML document. The DOCTYPE element implies the documents we generate are HTML 3.2 compliant (which is true except for the use of the <NOBR> element...).

[1] createdoc(t) = (<!DOCTYPE␣HTML␣PUBLIC␣"-//W3C//DTD␣HTML␣3.2//EN">␣<HTML>
<HEAD> < META NAME = "GENERATOR" CONTENT = "Box2HTML" > </HEAD>
<BODY> t </BODY> </HTML>)

This function removes the quotes which enclose a box-string before it ends up in the resulting hypertext.

[2] rmquotes("") = ()

[3] rmquotes(box-str-con(""" chr^{+} """)) = box-str-con(chr^{+})
[4] rmquotes(a) = () **otherwise**

Hblank generates non-breaking spaces ( ), which we employ as 'non collapsible spaces' until the standard provides a way of doing this. We generate a single space as space character, this will make the code easier to read. For backwards compatibility, we refer to this entity using the numerical representation ( ).

[5] hblank(1) = rmquotes("␣")
[6] hblank(*int*) = hblank2(*int*) **otherwise**

[7] $$\frac{\mathit{int} < 1 \;=\; \mathsf{true}}{\mathsf{hblank2}(\mathit{int}) \;=\; ()}$$

[8] hblank2(*int*) = (` ` hblank2(*int* − 1)) **otherwise**

Vblank generates
 elements which are used to separate lines.

[9] $$\frac{\mathit{int} \leq 1 \;=\; \mathsf{true}}{\mathsf{vblank}(\mathit{int}) \;=\; \texttt{
}}$$

[10] vblank(*int*) = (`<BR>` vblank(*int* − 1)) **otherwise**

The specials function scans for characters which have a special meaning in HTML and have not already been escaped in the box term. We only provide for characters which are likely to occur in source code, but this can easily be extended here.

[11] specials("") = ""

[12] specials(box-str-con(""" "<" chr^{*} """))
= "`<`" ∗ specials(box-str-con(""" chr^{*} """))

[13] specials(box-str-con(""" ">" chr^{*} """))
= "`>`" ∗ specials(box-str-con(""" chr^{*} """))

[14] specials(box-str-con(""" "&" chr^{*} """))
= "`&`" ∗ specials(box-str-con(""" chr^{*} """))

[15] specials(box-str-con(""" """ chr^{*} """))
= "`"`" ∗ specials(box-str-con(""" chr^{*} """))

[16] specials(box-str-con(""" *chr* chr^{*} """))
= box-str-con(""" *chr* """) ∗ specials(box-str-con(""" chr^{*} """))
otherwise

Chapter 5

Designing Box to HTML

5.1 Design choices

Translating boxes to HTML code is not as straightforward as we would like it to be. The basic box operators provide precise typesetting control, while HTML is not (yet) capable of dealing with this kind of advanced formatting. To further complicate things, the HTML semantics as implemented by popular browsers are inconsistent with eachother and the standard. However, we have shown some ways to overcome a number of the current limitations. Future developments, like the introduction of style sheets, will provide a new level of formatting control. For now, this leaves us with essentially three ways to implement Box to HTML:

1. *Use preformatted text*
   Adapt and extend the Box to Text formatter to produce text embellished with HTML elements. Embed this result in the HTML `<PRE>` element, which displays text exactly as formatted. This is the only way to achieve a full implementation of the basic box semantics according to the current HTML standard.

2. *Creative use of HTML*
   Design a translation that uses all options of the current HTML standard along with commonly accepted 'tricks' to get a result that is close to the box semantics.

3. *SCML*
   Create our own markup language for typesetting source code (Source Code Markup Language) in SGML and create a Box to SCML formatter. This is the SGML way of thinking, e.g., create a markup language for your specific need. However, no current browser application will be able to format our documents. Eventually, the introduction of an extensible markup language like XML (Section 7.4.1) could bring this kind of facilities to the web.

5.1.1 Preformatted text

Browsers render preformatted text with a fixed pitch font, preserving spacing associated with white space characters such as space and newline characters. HTML allows the use of character highlighting and links in preformatted text, but tables and lists are not supported, as well as elements that produce changes in fontsize.

If we want to generate preformatted text we need all of the semantics implemented by the Box to Text formatter. However, adapting the Box to Text formatter was not as simple as we first expected. Box to Text uses lists of strings to represent the formatted text. We can not maintain this format, because we need to somehow discriminate HTML elements from text (e.g., for length calculations). We would have to adapt Box to Text to use our HTML syntax (Section 4.3.2).

For the small (algebraic) version of Box to Text (Section 3.6.1), we would have to adapt all text composition operators. This introduces a new problem. Because we have dropped lists of strings as output format, we have also lost the notion that every string represents a line on the display. Because all text composition operators rely on this notion, we can not easily rewrite them for our more complicated HTML syntax.

The extended Box to Text version (Section 3.6.1) constructs all output at the deepest recursion level, by concatenating each string in the box term to the text constructed so far. We can not use this approach with, because we have to insert pairs of matching HTML tags in the output. Therefore, we need a recursive approach to construct the output.

5.1.2 Creative use of HTML

With this approach we are no longer bound to producing 'lines' of text. Producing output in this way is more like generating TEX code, and offers the advantage that the generated output can be dynamically resized by the browser to fit in the available window. However, we can not implement all basic box semantics with this approach. For example, the V and HV operators require that if we move to a new line, the text returns to the column where the first box started. Currently, HTML provides no way to achieve this dynamically. In order to fully implement the basic box semantics, we have to make these decisions during the translation. This implies that we have to format our output for a fixed page width. However, if we use the formatting solutions presented in Section 4.3.1 we can still construct our output without using preformatted text.

5.2 An initial approach

Because adapting the Box to Text formatter was not efficient, we decided to drop this approach and to study the use of HTML elements to format text. To get started, we designed a translation that leaves all formatting to the browser. We will use this specification to explain design choices for the implementation of the *non*-basic box operators, like alignments, links, fonts and comments. Furthermore, this implementation is a perfect framework for a future redesign, when this kind of formatting control becomes available in HTML. In Chapter 6 we will present an improved version that provides a full implementation of the basic box semantics by using calculations.

5.2.1 Box-to-Html-Simple

This is a simple version of Box to HTML. It is simple because it leaves all formatting to the HTML browser and performs no calculations. The result of this approach is that we can not fully implement the basic box semantics, because we have no information about the effective margins of the page. However, for some applications this limited approach can already produce useful results. The limitations of this implementation are:

- H This operator is fully implemented, including space-options.
- V This operator is fully implemented as long as it starts in the first column. If it starts in another column, remaining boxes will shift back to the left margin.
- I Implemented, but we can not control the *is* option. Furthermore, our implementation will introduce unwanted vertical space on some browsers.

- HV This implementation leaves the formatting of the HV operator to the browser, which will insert a new line when appropriate. The specification describes that the text should return to the column where the first argument box started. However, we can only fix a new left margin if the HV operator occurs in the context of an I operator. In other contexts the text will return to the first column.
- HOV Because we can not do any calculations, the HOV operator will *always* be rendered as a V operator.
- Alignments These are implemented using the <TABLE> element.
- Fonts Fully implemented.

imports Html-Op$^{4.4.1}$ Box String-Op Space-Options
exports
  context-free syntax
    box2html(BOX-LIST) → HTEXT
    html(BOX-LIST) → HTEXT
    place(HTEXT, BOX-LIST) → HTEXT
hiddens
  variables
    [*hiv*] → BOX-INT
    [*ab*][*0-9*]∗ → BOX-STRING
    [*B-E*][*0-9*]∗ → BOX
    [*B-E*]"+"[*0-9*]∗ → BOX+
    [*B-E*]"∗"[*0-9*]∗ → BOX∗
equations

Lists of Boxes place defines a translation on lists of boxes. Boxes are traversed recursively. Code for boxes is seperated with space-options if they apply.

[1] place(t,) = ()
[2] place(t, B) = html(B)
[3] place(t, B B^+) = (html(B) t place(t, B^+))

Strings Strings are translated by removing their doublequotes and by applying the specials function which scans for special characters which have to be replaced by a HTML entityreference.

[4] html(a) = rmquotes(specials(a))

Main This is the starting point for html translations.

[5] box2html(B^*) = createdoc(html(B^*))

H **boxes** are translated by enclosing them in a <NOBR> tag to avoid line breaking. The subboxes are separated according to the space-options by the hblank function.

[6] html(H o^*[B^*]) = (<NOBR> place(hblank(H(o^*)), B^*) </NOBR>)

V **boxes** are translated by separating them with a
 tag.

[7] $\text{html}(\text{V } o^*[B^*]) = \text{place}(\text{vblank}(\text{V}(o^*)), B^*)$

I **boxes** are implemented by using an <UL> element (without LI items). This causes a correct indentation, but the *is* option cannot be set and some browsers automatically insert a newline before the start of an indentation.

[8] html(I $o^*[B]$) = (<UL> html(B) </UL>)

WD **boxes** are implemented straightforward, but we can only process a single string as argument.

[9] $\text{html}(\text{WD}[a]) = \text{hblank}(\text{width}(a))$

HOV **boxes** are mapped to a V operator, because we can not determine the right margin at this point. Therefore we can never garantuee that all subboxes of a HOV operator can be placed horizontally.

[10] $\text{html}(\text{HOV } o^*[B^*]) = \text{html}(\text{V } o^*[B^*])$

HV **boxes.** The formatting of the HV operator is left to the browser, which will insert a new line when appropriate. However, the *is* option can not be set, and in some contexts (e.g. HV in H) the text will return to the first column

[11] $\text{html}(\text{HV } o^*[B^*]) = \text{place}(\text{hblank}(\text{H}(o^*)), B^*)$

5.2.2 Alignments-to-Html

Alignments are implemented using the <TABLE> element. We can translate alignments in a straightforward way to a HTML table. Each row is mapped to a <TR> (table row) element. All boxes in a row operator are mapped to <TD> (table data) element. The ALIGN attribute is set according to the values in A-OPTIONS. This has to be done for each cell in the table, because in HTML the alignment can not be set for columns but only for rows. Currently, this implementation does not support nested alignments. This is a design decision and not a limitation of HTML. Space options for the A and R elements are also not supported in this version.

imports Alignments Box-to-Html-Simple[5.2.1]
hiddens
  context-free syntax
    rows(A-OPTIONS, BOX-LIST) $\to$ HTEXT
    elements(A-OPTIONS, BOX-LIST) $\to$ HTEXT
hiddens
  variables
    [B-E][0-9]∗ $\to$ BOX
    [B-E]"+"*[0-9]∗* $\to$ BOX+
    [B-E]"∗"*[0-9]∗* $\to$ BOX∗
equations

This is the starting point for translating alignments.

[1] html(A (ao^*) $so^*[B^*]$) = (<TABLE> rows((ao^*) , B^*) </TABLE>)

For each row a <TR> element is inserted.

[2] rows(() ,) = ()

$$
\begin{array}{lll}
[3] & \mathsf{rows}((\mathit{ao}^*)\ ,\ \mathsf{R}[B^+]) & = (\texttt{<TR>}\ \mathsf{elements}((\mathit{ao}^*)\ ,\ B^+)) \\
[4] & \mathsf{rows}((\mathit{ao}^*)\ ,\ \mathsf{R}[B^+]\ C^+) & = (\texttt{<TR>}\ \mathsf{elements}((\mathit{ao}^*)\ ,\ B^+)\ \mathsf{rows}((\mathit{ao}^*)\ ,\ C^+))
\end{array}
$$

For each box in a row a <TD> element is inserted.

$$
\begin{array}{lll}
[5] & \mathsf{elements}(()\ ,\ ) & = () \\
[6] & \mathsf{elements}((\mathsf{l}\ ,\ \mathit{ao}^*)\ ,\ B\ B^*) & = (<\texttt{TD ALIGN} = \texttt{LEFT} > \mathsf{html}(B)\ \mathsf{elements}((\mathit{ao}^*)\ ,\ B^*)) \\
[7] & \mathsf{elements}((\mathsf{c}\ ,\ \mathit{ao}^*)\ ,\ B\ B^*) & = (<\texttt{TD ALIGN} = \texttt{CENTER} > \mathsf{html}(B)\ \mathsf{elements}((\mathit{ao}^*)\ ,\ B^*)) \\
[8] & \mathsf{elements}((\mathsf{r}\ ,\ \mathit{ao}^*)\ ,\ B\ B^*) & = (<\texttt{TD ALIGN} = \texttt{RIGHT} > \mathsf{html}(B)\ \mathsf{elements}((\mathit{ao}^*)\ ,\ B^*))
\end{array}
$$

5.2.3 Fonts-to-Html

This module defines a translation for the font operator. The font operator F can be followed by one or more font options. We provide a translation for the parameters font name (*fn*), font size (*sz*) and color (*cl*). Font operator abbreviations can be used to abstract from actual typesetting, by providing logical styles for mathematics (*MATH*), variables (*VAR*), keywords (*KW*) and numbers (*NUM*).

imports Box-to-Html-Simple$^{5.2.1}$ Fonts
hiddens
variables

$$
\begin{array}{ll}
[B\text{-}E][0\text{-}9]* & \to \mathsf{BOX} \\
[B\text{-}E]\text{"+"}[0\text{-}9]* & \to \mathsf{BOX+} \\
[B\text{-}E]\text{"}*\text{"}[0\text{-}9]* & \to \mathsf{BOX}* \\
\text{"}chr*\text{"}[0\text{-}9']* & \to \mathsf{CHAR}* \\
Int\ [0\text{-}9']* & \to \mathsf{BOX\text{-}INT}
\end{array}
$$

equations

Font properties are translated to the appropriate HTML elements. The FACE attribute specifies a specific font, like courier or times. If the appropriate font is not installed on the computer which is used to view the document, the browser will use the default font.

$$
\begin{array}{llll}
[1] & \mathsf{html}(\mathsf{F\ fn} = \texttt{KWf}[B]) & = (\texttt{<B>}\ \mathsf{html}(B)\ \texttt{</B>}) & \\
[2] & \mathsf{html}(\mathsf{F\ fn} = \mathsf{fid}(chr^*)[B]) & = (<\texttt{FONT FACE} = \mathsf{id}(chr^*) > \mathsf{html}(B)\ \texttt{</FONT>}) & \textbf{otherwise}
\end{array}
$$

The SIZE attribute specifies font size.

$$[3] \quad \mathsf{html}(\mathsf{F\ sz} = Int[B]) = (<\texttt{FONT SIZE} = Int > \mathsf{html}(B)\ \texttt{</FONT>})$$

The COLOR attribute can be a RGB triplet (e.g. "FF0000") or a color name

$$[4] \quad \mathsf{html}(\mathsf{F\ cl} = \mathsf{fid}(chr^*)[B]) = (<\texttt{FONT COLOR} = \mathsf{id}(chr^*) > \mathsf{html}(B)\ \texttt{</FONT>})$$

We define the abbreviations as follows:

$$
\begin{array}{lll}
[5] & \mathsf{html}(\mathsf{MATH}[a]) & = \mathsf{html}(a) \\
[6] & \mathsf{html}(\mathsf{VAR}[a]) & = (\texttt{<EM>}\ \mathsf{html}(a)\ \texttt{</EM>}) \\
[7] & \mathsf{html}(\mathsf{KW}[a]) & = (\texttt{<B>}\ \mathsf{html}(a)\ \texttt{</B>}) \\
[8] & \mathsf{html}(\mathsf{NUM}[a]) & = \mathsf{html}(a)
\end{array}
$$

The escape operator can be used to include special characters or mathematical symbols which are mapped to HTML entity references. However, the characters mentioned below are also handled by the 'specials' function if they are not escaped. This list can be extended to include characters for a specific purpose.

[9] html(ESC[">"]) = `>`
[10] html(ESC["<"]) = `<`
[11] html(ESC["&"]) = `&`
[12] html(ESC[*a*]) = html(*a*) **otherwise**

5.3 Formatting comments

Formatting comments seems a trivial issue for our specification. The box operators specify where a comment should be restored in the formatted text. However, we are designing Box to HTML to be used in combination with the other back-ends, and therefore it is desired to introduce back-end specific comments. With such a mechanism, we can distinguish comments which are only relevant for TEX (e.g. text containing TEX commands) from, for example, comments containing HTML markup. In this way, we can create documentation in different formats from a single source file.

The current algorithm for restoring comments inserts a box-term containing the comment begin- and end marker and the comment text. A straightforward way to do create 'back-end-specific' comments would be to use different comment markers for different kinds of comments. For example:

Marker	Text	HTML	TEX
%%	yes	no	no
%%T%	yes	no	yes
%%H%	yes	yes	no
%%A%	yes	yes	yes

With this approach, the person writing comments is forced to be specific about where comments should appear. If the comments are not edited, they will only show up when formatted with the text formatter. There is no way to exclude comments from the text formatter, because the text formatter is also used for pretty printing code within the ASF+SDF term editor. The comment marker reverts to '`%%`' in the formatted output, but this decision could easily be overruled. This mechanism is only effective when implemented by *all* back-ends, so the corresponding modules of the text and TEX back-ends have to be adapted as well.

5.3.1 Separating comments from code

Another interesting HTML capability is to use a different color and/or text style for comment text. This will make a listing much easier to read when viewed on a display. We suggest this should be achieved by enclosing the generated comment operators in the front-end formatter in a font operator. Another option is to implement this in the module below, but this is less flexible. As an example, we have chosen to layout comments using the `<EM>` element.

5.3.2 Comments-to-Html

This module formats the comment-operators. A comment of type PAR is prefixed with a `<BR>` tag to force it to start on a new line. All comment text is formatted as H box, because we can not determine where to insert comment begin- and/or endmarkers if the comment is broken over multiple lines. Comment text is formatted emphasized to seperate it from source text.

imports Box-to-Html-Simple[5.2.1] Comments
hiddens
  context-free syntax
    checkcomment(BOX-STRING) → BOX-BOOL
  variables
    *[ab][0-9]** → BOX-STRING
    *[B-E][0-9]** → BOX
    [B-E]"+"*[0-9]** → BOX+
    [B-E]"*"*[0-9]** → BOX*
equations

This function checks its argument to see if it is a recognized comment-marker. If we want to format comments using another marker, it can easily be changed here.

[1] checkcomment("%%H%") = true
[2] checkcomment("%%A%") = true
[3] checkcomment(*a*) = false **otherwise**

These rules format comment text that starts with the required marker for this back-end.

[4]
$$\frac{\mathsf{checkcomment}(a) \;=\; \mathsf{true}}{\mathsf{html}(\mathsf{PAR}\ a\ b[B^*]) \;=\; (\texttt{
 <EM>}\ \mathsf{rmquotes}(\texttt{"\%\%"})\ \mathsf{html}(\mathsf{H}\ [B^*])\ \texttt{</EM>})}$$

[5]
$$\frac{\mathsf{checkcomment}(a) \;=\; \mathsf{true}}{\mathsf{html}(\mathsf{VPAR}\ a\ b[B^*]) \;=\; (\texttt{<EM>}\ \mathsf{rmquotes}(\texttt{"\%\%"})\ \mathsf{html}(\mathsf{H}\ [B^*])\ \texttt{</EM>})}$$

[6]
$$\frac{\mathsf{checkcomment}(a) \;=\; \mathsf{true}}{\mathsf{html}(\mathsf{HPAR}\ a\ b[B^*]) \;=\; (\texttt{<EM>}\ \mathsf{rmquotes}(\texttt{"\%\%"})\ \mathsf{html}(\mathsf{H}\ [B^*])\ \texttt{</EM>})}$$

Comment text that is not formatted using a required marker for this back-end is discarded.

[7]
$$\frac{\mathsf{checkcomment}(a) \;=\; \mathsf{false}}{\mathsf{html}(\mathsf{PAR}\ a\ b[B^*]) \;=\; ()}$$

[8]
$$\frac{\mathsf{checkcomment}(a) \;=\; \mathsf{false}}{\mathsf{html}(\mathsf{VPAR}\ a\ b[B^*]) \;=\; ()}$$

[9]
$$\frac{\mathsf{checkcomment}(a) \;=\; \mathsf{false}}{\mathsf{html}(\mathsf{HPAR}\ a\ b[B^*]) \;=\; ()}$$

5.4 Hypertext links

Hypertext is a powerful format for documenting computer programs, because of its ability to provide all kinds of links. For example, we can use hypertext to provide a link from the use of a function to its definition. When typesetting ASF+SDF specifications, we can use hypertext to link names of imported modules to their respective definitions. In this way, hypertext is a powerful tool for comprehending program source code.

Basic hypertext documentation might consist of a *table of contents*, which links to each source file, along with buttons for navigating from one document to the next. Setting up this basic hypertext structure is outside the scope of Box to HTML, but can easily be coded by hand or by a tool. For now, we are interested to see what links can be provided automatically by the formatter generator using the information present in the abstract syntax tree. For a particular application, a variety of links may be added to enable non-linear browsing. For example:

- Links from uses of functions, types and/or variables to their respective definitions (Figure 5.1).
- Links from included and imported files/modules to their definition (Figure 5.2).
- Links from language primitives and library functions to documentation, like manual pages (Figure 5.3).
- Links from definitions of functions, types and variables to *backtrace lists*, which provide links to all places where that item is used.

This list is just an indication, as a 'linking strategy' depends heavily on the syntax of the used language and of the intended use of the documentation.

```
imports Layout
exports
  sorts BOX-BOOL BOX-BOOL-LIST

...
```

$\longrightarrow$

```
exports
  lexical syntax
    [ \t\n]             -> LAYOUT
    "%%" ~[\n]* "\n" -> LAYOUT
    "%" ~[%\n]+ "%"  -> LAYOUT
```

Figure 5.1: ASF+SDF: Imported module linked to its definition

```
while counter do
  ...
  counter := counter - 1
  ...
od;
```

$\longrightarrow$

```
begin
  declare
    counter: natural,

...
```

Figure 5.2: Pico: Variable use linked to its declaration

5.4.1 Crossreference operators

To include links in the box term, we can use the box crossreference operators LBL and REF. These operators have been part of the box specification for some time, but are not yet generally used by the formatter generator or back-ends. The crossreferencing operators have two arguments. The first argument is a string which is used to label a reference. The second argument is a box term, which contains the actual text.

Variables, functions, and other names that occur in program source code are usually unique. Because of this property we can use the name of an object as its label. Another approach would

```
context-free syntax
  true     -> BOX-BOOL
  false    -> BOX-BOOL
  ...
```

→

```
KEYWORD: Context-free syntax
Describes concrete and abstract syntax.
It contains a set of declarations for
functions, each consisting of zero or
more literals, sort names or lists
followed by '->' and a result sort.

...
```

Figure 5.3: ASF+SDF: Keyword linked to documentation

be to generate incremental labels, like `ref1`, `ref2` and so on, but this would require a complex extension to the generated formatters. To discriminate labels generated for Box to HTML from other labels, we will prefix each label by the string "`A:`" (for hypertext anchor).

The front-end formatters have to be adapted to include crossreference operators in the generated box term. This formatter also has to create labels for each reference. We will give some examples of how to do this as part of the case studies in the next chapter.

5.4.2 Crossreferences-to-Html

This module describes the mapping of box crossreference operators to hypertext links and anchors. The LBL operator is used to designate a box term as the destination of a link (hypertext anchor). The REF operator is mapped to a hypertext link. The first argument of the LBL or REF operator contains a string which is used to set the value of the appropriate anchor or link attribute. This string must start with "`A:`" to identify it as a hypertext label.

imports Crossreferences Box-to-Html-Simple[5.2.1]
hiddens
  context-free syntax
    checklabel(BOX-STRING) → BOX-STRING
  variables
    "*chr**" → CHAR*
    [*B-E*][*0-9*]* → BOX
    [*B-E*]"+"[*0-9*]* → BOX+
    [*B-E*]"*"[*0-9*]* → BOX*
equations

This function checks a label to see if it starts with "`A:`". If so, the remainder of the string is returned.

[1] checklabel(box-str-con("\"" "A" ":" *chr** "\"")) = box-str-con("\"" *chr** "\"")
[2] checklabel(*a*) = "" **otherwise**

Links The LBL operator is translated to a target location, by setting the NAME attribute of the <A> element with the relevant part of the string argument of the operator.

$$[3] \quad \frac{\mathsf{checklabel}(a) = b, \quad b \neq \texttt{""}}{\mathsf{html}(\mathsf{LBL}[a, B]) = (\texttt{<}\ \texttt{A NAME} = b\ \texttt{>}\ \mathsf{html}(B)\ \texttt{</A>})}$$

[4] html(LBL[*a*, *B*]) = html(*B*) **otherwise**

The REF operator is translated to a hypertext link, by setting the HREF attribute of the <A> element. The relevant part of the string argument is treated as an URL. This implies that the front-end formatter has to decide if it generates local (e.g. `"#label"`) or external (e.g. `"filename"` or `"www address"`) references. This is a design choice.

$$[5] \quad \frac{\begin{array}{rcl}\mathsf{checklabel}(a) & = & b, \\ b & \neq & \texttt{""}\end{array}}{\mathsf{html}(\mathsf{REF}[a,\ B]) \ = \ (<\texttt{A HREF} = b > \mathsf{html}(B)\ \texttt{</A>})}$$

$$[6] \quad \mathsf{html}(\mathsf{REF}[a,\ B]) = \mathsf{html}(B) \quad \textbf{otherwise}$$

Chapter 6

A full prototype of Box to HTML

In the previous chapter, we have presented an initial translation for Box to HTML and have proposed a standard for creating back-end specific comments. This translation does not fully implement the basic box semantics, because HTML does not (yet) provide the required formatting control.

We can only achieve a full implementation if we use calculations to format the text for a given fixed page-width. In Appendix A we present a prototype of Box to HTML which uses this approach to fully implement the basic box semantics. It keeps track of the remaining space on the current line during the translation, and takes formatting decisions accordingly. It is based on the initial translation and the Box to Text formatters. In fact, if we replace `<BR>` elements with newline characters and ` ` elements with space characters in the output, we obtain preformatted text (except for alignments).

Output is formatted for a given fixed page-width, by keeping track of the remaining space on the current line during the translation. It also provides a translation for the `O` operator for typesetting conditional ASF+SDF equations. We have used this improved version to typeset the examples for the case studies in this chapter. In Appendix C you will find the generated HTML document source for the shown examples.

6.1 Known problems

Formatting the generated code Box to HTML generates a continuous stream of text and HTML elements. To view the source code properly, we have applied some manual formatting, like putting newlines after every `<BR>` element. Fortunately, the Box tools are also capable of generating a HTML formatter to do this for us.

Spaces introduced by the ASF+SDF term prettyprinter The term prettyprinter separates each context-free syntax element with a space character, which has to be edited out. A generated HTML formatter will also enable us to solve this problem.

Tables The `<TABLE>` element introduces a newline when invoked, which inserts an empty line in the document when the alignment occurs in V context. This can be corrected by introducing a specialized `place` function in the alignments module which handles this case, but is also simple to edit out manually.

Preventing linebreaks The `<NOBR>` element which we use to implement the `H` operator is the only element that is specific to Netscape Navigator, which means we can not expect it to be handled correctly on other browsers.

6.2 Case Study: Typesetting ASF+SDF

We have adapted the AsFix to Box formatter and have used the output with Box to HTML to create a typeset version of the module `Box-Booleans`.

6.2.1 Adapting the AsFix formatter

The AsFix formatter translates AsFix to Box. We can use it to format ASF+SDF specifications. The AsFix formalism has been updated recently, but the formatter has not yet been rewritten so we have used the old version for this example.

Suppose we want to create links from imported modules to their respective definitions, like we have shown in Figure 5.1. This is the relevant equation from the AsFix pretty printer:

```
[2'5]    ppATerm(T) = B*
         ===============
         ppATerm(["Imports" T]) = H [ KW ["imports"] HV is=0 [B*]]
```

It takes an AsFix 'import' section, which consists of a list of modulenames in AsFix format (T). A box term is created which contains 'imports' as keywords and the modulenames after they have been processed by ppATerm to obtain a list of plain modulenames (B*). We add a function **crefGen** to this specification which takes such a list and adds REF operators.

```
context-free syntax
  crefGen(BOX-LIST)             -> BOX-LIST
equations
  [2'5'1] crefGen() =
  [2'5'2] crefGen(a B*) = REF["A:" * a * ".html",a] ++ crefGen(B*)
```

Note that we have not defined LBL operators anywhere. Since we generate separate files for each ASF+SDF module, we do not need them because we can refer to the filename. To achieve this, the suffix **.html** is added to the name of the module. Finally, we adapt the original equation:

```
[2'5]    ppATerm(T) = B*,
         crefGen(B*) = C*
         ===============
         ppATerm(["Imports" T]) = H [ KW ["imports"] HV is=0 [C*]]
```

Now for each modulename in an import section, the generated HTML code will contain an entry like: `<A HREF = "Modulename.html"> Modulename </A>`

6.2.2 A HTML documentation tool for ASF+SDF

The AsFix formatter produces two box terms. One for the SDF part of the specification and one for the ASF part. For each box term, Box to HTML produces a HTML file which we have named **Modulename.sdf.html** and **Modulename.asf.html**. To combine these files into a single window when viewed with a browser, we have used the HTML `<FRAME>` element (Figure 6.1). For each module, we would have to create such a HTML document named `Modulename.html` which invokes the appropriate files. To support older browsers that do not implement frames, we include a `<NOFRAMES>` element that contains links to both modules. The result is shown in Figure 6.2. Links are underlined. When the name of a linked module is clicked, the browser will look for file named `Modulename.html`.

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 3.2//EN">
<HTML> <HEAD>
<TITLE>Box-Booleans</TITLE>
</HEAD> <BODY>
<FRAMESET ROWS="50%,50%">
<FRAME SRC="Box-Booleans.sdf.html" NAME="SYNTAX">
<FRAME SRC="Box-Booleans.asf.html" NAME="EQUATIONS">
</FRAMESET>
<NOFRAMES>
<A HREF="Box-Booleans.sdf.html"> Box-Booleans.sdf </A>
<A HREF="Box-Booleans.asf.html"> Box-Booleans.asf </A>
</NOFRAMES>
</BODY> </HTML>
```

Figure 6.1: Defining a frameset

The ToLaTeX tool for typesetting Asf+Sdf specifications [Vis95] generates a file for each module in the specification. It can also generate crossreferences between modules and an index for the entire specification. To obtain HTML documentation for a specification, including crossreferences and an index, we present the following procedure:

1. All modules are converted to AsFix using `Modules2Asfix`. This produces a file with suffix '`.asfix`' for each file and a file '`modules`', which contains the relative pathnames for all converted files.
2. The adapted AsFix formatter is called for each file in `modules` and generates a box term for the SDF part. Box to HTML converts this to HTML and the result is put into a file '`modulename.sdf.html`'.
3. The adapted AsFix formatter is called for each file in `modules` and generates a box term for the ASF part. Box to HTML converts this to HTML and the result is put into a file '`modulename.asf.html`'.
4. For each module, a file '`modulename.html`' is created according to Figure 6.1 with the appropriate modulenames instantiated. To prevent that the frameset is loaded *inside* an existing frame, all modules containing links to this file must have a `<BASE TARGET="_TOP">` element in the `<HEAD>` section.
5. The file `modules` is extended with a hypertext link for each entry and renamed to `index.html`.
6. The template for the file '`modulename.html`' as shown in Figure 6.1 can be adapted to achieve improved browsing capabilities, for example by introducing a link 'TOP' to `index.html`.

6.3 Case Study: Typesetting Pico

We have used the generated formatter for the toy language PICO to demonstrate another use of hypertext links, linking the use of variables to their respective definitions, as proposed in Figure 5.2.

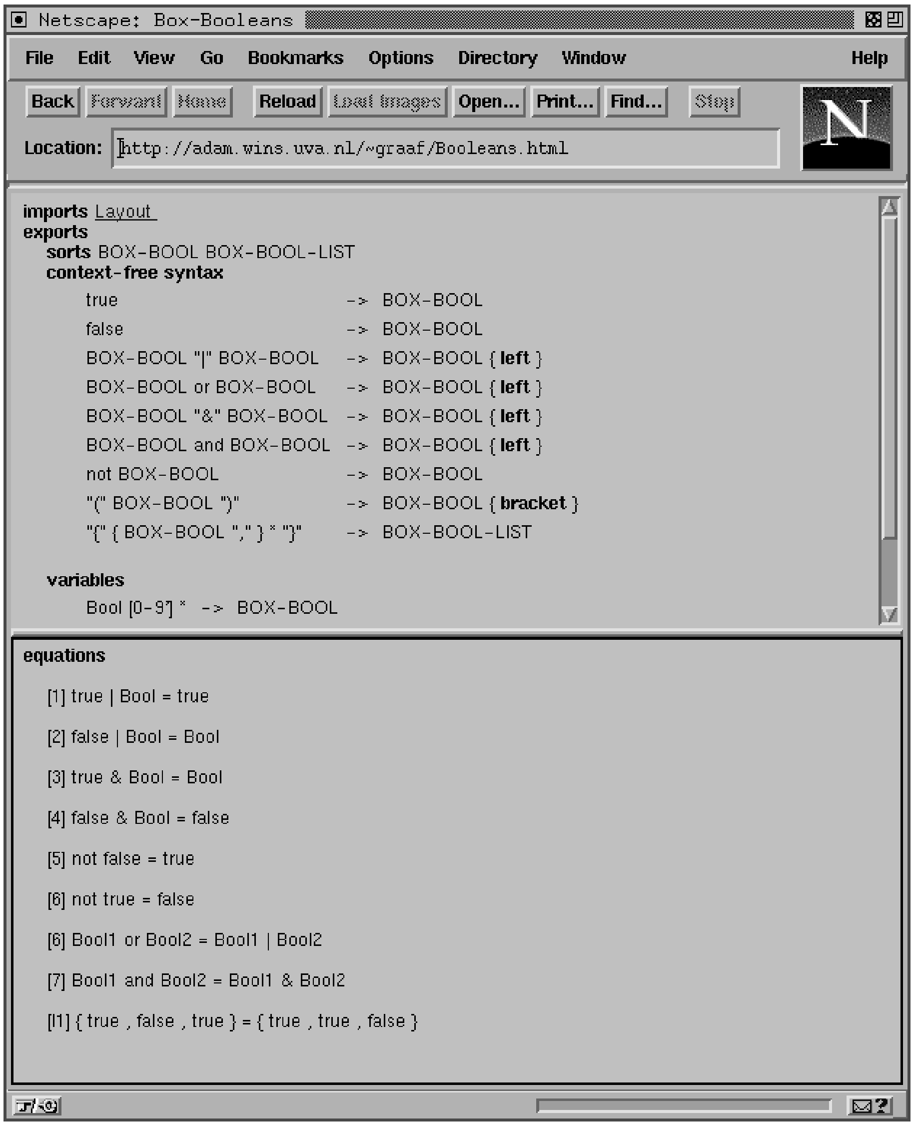


Figure 6.2: Module Box-Booleans in Netscape

Figure 6.3 shows an example of a Pico program typeset with the adapted Pico formatter and Box to HTML. When we click on a variable name, the browser will jump to its definition, which is especially useful when typesetting a large file. We have also included some comments using the HTML only comment marker (`%%H%`).

6.3.1 Adapting the PICO formatter

First, we have to attach a label to every variable declaration. Here is the appropriate equation from the formatter:

```
[default-3] ppID-TYPE(var_ID:0 : var_TYPE:2) =
              HV [ ppID(var_ID:0) ":" ppTYPE(var_TYPE:2) ]
```

This equation formats statements of the form `"id" : "type"`, which is only used in variable declarations. The function `ppID` transforms a Pico ID into a box-term. Because the result sort is always of type BOX-STRING, we have changed the result sort. Finally, we adapt the rule for typesetting variable declarations to insert a LBL operator with a label that has the same name as the variable.

```
context-free syntax
   ppID ( ID ) -> BOX-STRING
equations
[default-3] ppID-TYPE(var_ID:0 : var_TYPE:2) =
               HV [ LBL[ "A:":ppID(var_ID:0),ppID(var_ID:0)] ":" ppTYPE(var_TYPE:2) ]
```

To insert references, we adapt the rules that format variable assignments, and variables that occur in expressions:

```
[default-5] ppSTATEMENT(var_ID:0 := var_EXP:2) =
            HV [ REF["A:#":ppID(var_ID:0),ppID(var_ID:0)] ":=" ppEXP(var_EXP:2) ]
[default-11] ppEXP(var_ID:0) = REF["#":ppID(var_ID:0),ppID(var_ID:0)]
```

As a result, each variable definition is enclosed in a hypertext anchor, e.g. `<A NAME="varname">varname</A>`. Each occurence of that variable is enclosed in a hypertext link with a reference local to the document, e.g. `<A HREF="#varname">varname</A>`.

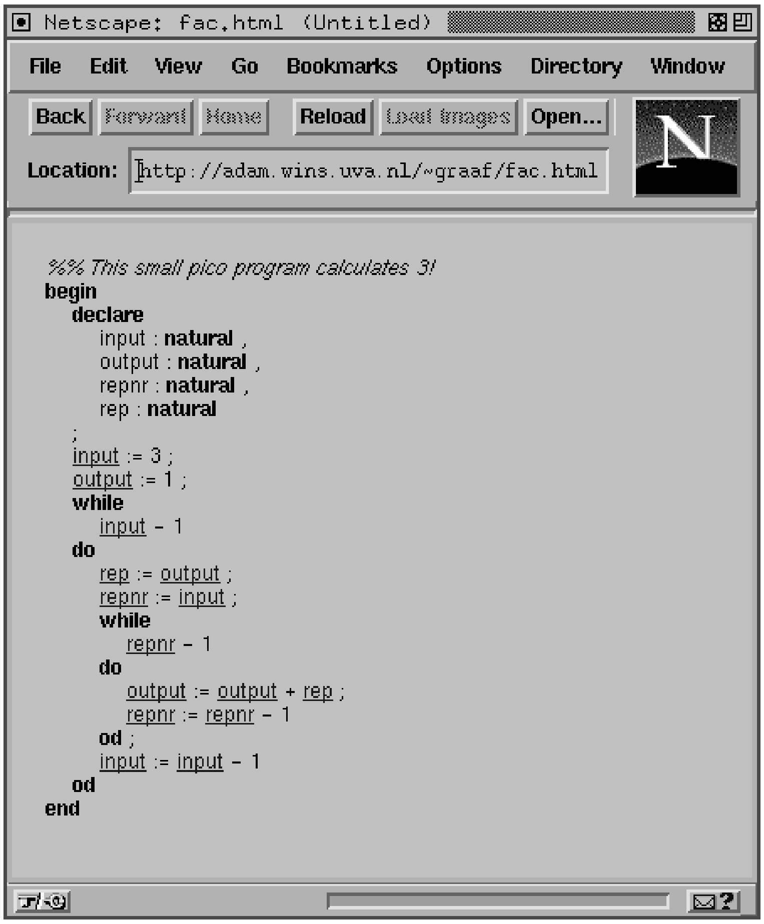


Figure 6.3: Pico program in Netscape

Chapter 7

Related Work

7.1 Program language filters

To obtain HTML formatted listings of source code many people have developed language specific filters. A list of available program language filters for HTML is available from the WWW consortium[1]. Besides being limited to a single language, most of them require a lot of additional tuning of the generated HTML code.

An interesting project is the Hypercode project at MIT [BHK$^+$94]. This project aimed at setting a standard for hypertext representation of program source code, which includes links from uses of functions, types, variables, and macros to their respective definitions. It also aimed at creating tools that generate function dependency graphs, backtrace lists, and indexes from hyperized source code. However, the project seems to be abandoned and only an implementation for C has been realized, relying heavily on C specific issues.

7.2 From TEX to HTML

Another approach is to generate HTML code from documentation already available in TEX format. HTML and TEX are both generic markup systems, and at first approximation translating between these two systems should not prove too difficult. A list of translators is also available from W3C[2].

At a basic level HTML provides equivalents for most TEX commands, like for example sectioning and lists. At a more advanced level, HTML falls short of providing equivalents. Some converters solve this by converting too complex parts into picture files. These converters do not operate very well on files with complex TEX commands, like the TEX files produced by ToLATEX and Box to TEX, and therefore we do not consider them a useful alternative.

7.2.1 Texinfo

Texinfo is a documentation system that uses a single source file to produce both on-line information and printed output. It uses a limited set of commands to express document markup and structure. It is typically used to create manuals, and program source code has to be included as verbatim text. It is part of Info, a hypertext system developed as part of the GNU project[3]. A

[1] http://www.w3.org/Tools/Prog_lang_filters.html
[2] http://www.w3.org/Tools/Word_proc_filters.html
[3] http://www.gnu.ai.mit.edu/

texinfo file can be typeset using TEX and the texinfo macro package. When converted to a set of 'info' files, the documentation can also be browsed using the program 'info'. With the texinfo to HTML converter `texi2html`[4] texinfo files can be used to create on-line documentation in hypertext format. Texinfo creates hypertext links for cross-references and footnotes. It creates several files with an index page and includes links for navigation at the top and bottom of each page. Source code will be included as preformatted text. This approach has been used to create on-line documentation for the OPAL system[5]. OPAL is a programming language which integrates both concepts of algebraic specification and functional programming. Since the documentation was already available in texinfo format, this path was chosen to create the HTML documents.

7.3 Applications

The combination of the formatter generator with Box to TEX and Box to HTML results in a potentially powerful documentation tool. We can produce both typeset and on-line documentation for any language given its context-free grammar in SDF. This tool enables us to create documentation for a system very quickly, like printed manuals and on-line source code library. This documentation can be updated automatically while a language or program is under development.

Related tools, like Texinfo (Section 7.2.1), force us to manually insert our code in the output, which is likely to cause errors. More importantly, the extra work involved will not encourage a programmer to create documentation right from the start of a project. However, the formatters generated by the formatter generator and the back-ends have to be tuned to satisfy the needs for a specific project. The amount of time needed to do this depends on how sophisticated the resulting tool has to be.

7.3.1 Teaching and analyzing purposes

Hypertext representation of program source code is very useful for teaching and analyzing purposes. With a proper use of hypertext links, it is easier to comprehend program control flow, datastructures and other aspects of code organization.

7.3.2 Creating program libraries

On-line documentation for source code can also be used to setup reference libraries for a particular project or programming environment, which can be quickly browsed to find a particular module. Potentially, hypertext offers the possibility to trace all dependencies (like included modules, functions and datastructure) of a piece of code. With a proper tool, a piece of code could be extracted *together* with all pieces of code it depends on. In this way, finding and re-using code from existing libraries could become very easy.

7.3.3 Literate programming

Another application is the construction of literate programming tools from language definitions. The concept of literate programming is that each program should be its own documentation, so that both the documentation and executable can be obtained from a single source file. Using chunk definitions and quotations is preferred in order to obtain documentation that has a highly modular structure. Such documentation is also very suitable for browsing with hypertext.

[4] http://wwwcn.cern.ch/dci/texi2html/
[5] http://uebb.cs.tu-berlin.de/~opal/

We could create an extended specification of, for example, PICO to allow a literate programming style where subprocedures are indicated with their name and specified later in the document. Because we use a literate programming syntax on top of the PICO specification, we can adapt the generated formatter to include crossreferences for chunk definitions which can be processed by the back-ends to create documentation.

7.4 Future Work

The HTML language is constantly evolving, and many additions are proposed and under development, like the Cascading Style Sheets proposal (Section 4.3.2). We expect that it is easy to incorporate these new elements in the Box to HTML specification, both to improve its capabilities and to keep up with the standards used by the Internet community.

7.4.1 XML

The Extensible Markup Language (XML)[6] is a recent proposal of the WWW consortium. XML aims to bring key benefits of generic SGML to the web. It consists of a simplified subset of SGML that retains the key SGML advantages of extensibility, structure, and validation in a language that is designed to be easier to learn, use, and implement than full SGML. XML differs from HTML in three major aspects:

1. New tag and attribute names can be defined at will
2. Document structures can be nested to any level of complexity
3. Any XML document can contain an optional description of its grammar for use by applications that need to perform structural validation

When implemented by web browsers, XML potentially provides a way to create a markup language for a specific need.

7.4.2 ASF+SDF documentation tool

The combination of ToBox (AsFix to Box) with Box to TeX and Box to HTML provides the basis for a tool that could replace ToLaTeX with a system that produces both TeX and HTML code from the same source. Additional integration and post-processing is necessary to create a tool that seamlessly generates all these output formats, without the need for heavy manual editing of the result. This requires, for example, extending Box to TeX with the crossreferencing operator and creating a post-processing tool which takes care of indexing and combining multiple HTML documents as in Section 6.2.2.

7.4.3 JAVA

JAVA is a programming language, loosely based on C++, and supported by popular browsers. It has been developed in an attempt to create a simple but powerful, robust, platform-independent language. Platform independent JAVA programs are called *applets*. JAVA applets can be incorporated in HTML documents and are interpreted by a JAVA-capable browser. This offers interesting possibilities. We can imagine a JAVA applet that has a Box term as input and produces a HTML document. This applet could be created by compiling the Box to HTML

[6] http://www.w3.org/XML/

specification to C and adapting the result. However, this application would only be interesting if we also have a compiled version of the generated formatter for a certain language, so we could do the entire transformation outside of the ASF+SDF Meta-Environment. Another approach is to implement a JAVA applet that has a box term as input an produces a typeset result in a JAVA GUI (Graphical User Interface). However, we might as well incorporate a screen-shot of Box to TeX output in a HTML document with the same result. Furthermore, using JAVA excludes users with text-oriented browsers and demands a lot of processing power on other systems. Currently, we do not think JAVA is a serious alternative of implementing Box to HTML.

Chapter 8

Conclusions

In this thesis we have presented a specification for a system that translates Box expressions into HTML code. Box expressions are declarative, language-independent descriptions of the intended layout of a program. HTML is a platform-independent language for creating hypertext documents.

This specification extends the functionality of the Box tools, a system that is capable of generating formatters for arbitrary languages from their syntax. These formatters map the constructs of a language to Box expressions, which are subsequently converted to human-readable output by back-ends that produce text, TEX code, or HTML code. We have shown that the system can be extended to provide automatic generation of hypertext links and we have proposed a standard for creating back-end specific comments.

The combination of back-end formatters forms a basis for building powerful documentation tools that assist programmers and documentation writers. From a single source program, we can create a prettyprinted version for display in an editor, a typeset version for printing and a browsable on-line version. Because of the modular design of the box tools and formatters, adapting the system for the specific needs of a language requires only small adaptions to the generated formatters and back-ends.

8.1 Formatting with HTML

The Box to HTML formatter is capable of formatting program texts without producing preformatted text. We think this is a useful approach considering future HTML enhancements, which will further improve formatting capabilities for web publishing. Other ways of including typeset representations of programs in HTML, like screenshots or a JAVA applet, will currently exclude the users of older and non-graphical browsers from accessing our documentation.

- **HTML markup**
  Are the markup capabilities of HTML powerful enough to produce typographically effective presentations of computer programs?

  HTML is a language for conceptual markup, not for typesetting. Formatting decisions are made by a HTML browser at run-time, and the semantics vary among browsers. The current HTML 3.2 standard offers very limited control over actual formatting. If we stick to this standard, we can only create a full implementation of the basic box semantics by generating preformatted text.

However, many of these limitations have been tackled by HTML writers and even browser manufacturers by creating layout effects with elements initially designed for another purpose. We have used all features from the current HTML version, both 'standard' and 'widely accepted' to be able to produce effective presentations of computer programs.

- **Hypertext links**
  Hypertext is a powerful concept for documenting computer programs. For example, we can link uses of functions and variables to to their respective definitions. Can we generate these links from an abstract syntax tree?

  We have shown that hypertext links can be included with simple adaptions to the generated formatter, using the standard crossreferencing operators proposed for Box. For more sophisticated applications, like building a library of ASF+SDF modules, post-processing is necessary using HTML template documents.

- **Comments**
  When creating both typeset and on-line documentation, we do not want them to be completely similar. For example, we want to have more elaborate comments in the typeset version while we want clickable links as comments in the hypertext version. How are we going to support this?

  We have introduced a mechanism for discriminating between comments for specific back-ends. To achieve this, we have introduced a set of comment-markers with fixed functions. The other back-ends have to be adapted to recognize these markers. To visually separate comments from source code, we have suggested the use of different colors and/or text styles.

Appendix A

A full prototype of Box to HTML

A.1 Box-to-Html

In this module we describe the transformation of boxes into HTML code. The box term is traversed only once, except for box terms starting with a HV or HOV operator which might need to reformat their subboxes. The function box2html takes a box term and a value for the right margin and transforms it into HTML. If no right margin is provided, a default value is used. We introduce a frame datastructure (HTEXT-FRAME) that holds all important information during the transformation of boxes to HTML. The first component (HTML-INFO) holds an integer value m which represents the number of characters used on the current line and a boolean *Vmode* which is true if we have placed any subboxes of the current box term in vertical context. The second component is an integer *Rm* which holds the maximum number of characters allowed on a line (the rightmargin). The auxiliary function html does the actual job by mapping each box operator to HTML. In this module it is only defined for the basic box operators.

```
imports Html-Op^4.4.1 Box String-Op Outstrings Contexts Space-Options
exports
  sorts HTEXT-STRUCT HTEXT-INFO HTEXT-FRAME
  context-free syntax
    "(" BOX-INT "," BOX-BOOL ")"                 → HTEXT-INFO
    HTEXT-INFO "&" BOX-INT                       → HTEXT-FRAME
    HTEXT "&&" HTEXT-INFO                        → HTEXT-STRUCT

    box2html(BOX-LIST)                           → HTEXT
    box2html(BOX-LIST, BOX-INT)                  → HTEXT
    html(BOX-LIST, HTEXT-FRAME, CONTEXT)         → HTEXT-STRUCT
  variables
    [ab][0-9]*          → BOX-STRING
    Htext [0-9]*        → HTEXT
    Hstruct [0-9]*      → HTEXT-STRUCT
    Info [0-9]*         → HTEXT-INFO
    Frame [0-9]*        → HTEXT-FRAME
    [mshvi][0-9]*       → BOX-INT
    Rm [0-9]*           → BOX-INT
    Vmode [0-9]*        → BOX-BOOL
    "X"                 → CONTEXT
```

The auxiliary function html uses the following hidden functions which place boxes in a certain context and separate them with space-options. During the transformation of the HV operator, we have to keep track of extra information like all space-options and the starting position of the first formatted box in case we have to do reformatting. The datastructure HV-FRAME is introduced to hold this info.

hiddens
  sorts HV-INFO HV-FRAME
  context-free syntax
    place(BOX-LIST, HTEXT-FRAME, BOX-INT, CONTEXT) → HTEXT-STRUCT

    "(" BOX-INT "," BOX-INT "," BOX-INT "," BOX-INT ")" → HV-INFO
    HV-INFO "&" BOX-INT → HV-FRAME

    hvplace(HTEXT-STRUCT, BOX-LIST, HV-FRAME) → HTEXT-STRUCT
    hvplace2(HTEXT-STRUCT, HTEXT-STRUCT, BOX-LIST, HV-FRAME) → HTEXT-STRUCT
  variables
    [B-E][0-9]∗ → BOX
    [B-E]"+"*[0-9]*∗ → BOX+
    [B-E]"∗"*[0-9]*∗ → BOX∗
    HVInfo [0-9]∗ → HV-INFO
    HVFrame [0-9]∗ → HV-FRAME
equations

This is the starting point for HTML translations. The function box2html takes care of the initialization of the frame datastructure. The second equation performs this in case the right margin is explicitly given. The resulting hypertext is passed to createdoc which adds the HTML preamble and closing elements.

$$[1] \quad \frac{\mathsf{html}(B^*,\ (0,\ \mathsf{false})\ \&\ 8\ 0,\ \mathsf{H})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{box2html}(B^*)\ =\ \mathsf{createdoc}(\mathit{Htext})}$$

$$[2] \quad \frac{\mathsf{html}(B^*,\ (0,\ \mathsf{false})\ \&\ \mathit{Rm},\ \mathsf{H})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{box2html}(B^*,\ \mathit{Rm})\ =\ \mathsf{createdoc}(\mathit{Htext})}$$

Strings are translated by removing their double quotes and by applying the 'specials' function which scans for special characters which have to be replaced by a html entity. Furthermore the length of the string is calculated, added to the number of chars used on the current line and passed back along with the resulting hypertext.

$$[3] \quad \frac{\mathsf{length\ of}\ a\ =\ m_2}{\mathsf{html}(a,\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ \mathsf{rmquotes}(\mathsf{specials}(a))\ \&\&\ (m + m_2,\ \mathit{Vmode})}$$

H **boxes** are translated by enclosing them in <NOBR> to avoid line breaking The horizontal space option is calculated only once and passed to the place function.

$$[4] \quad \frac{h\ =\ \mathsf{H}(o^*),\quad \mathsf{place}(B^*,\ \mathit{Frame},\ h,\ \mathsf{H})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{html}(\mathsf{H}\ o^*[B^*],\ \mathit{Frame},\ \mathsf{X})\ =\ (\texttt{<NOBR>}\ \mathit{Htext}\ \texttt{</NOBR>})\ \&\&\ \mathit{Info}}$$

V boxes. For the transformation of V boxes we calculate the vertical space option.

$$[5] \quad \frac{v \;=\; \mathsf{V}(o^*)}{\mathsf{html}(\mathsf{V}\ o^*[B^*],\ \mathit{Frame},\ \mathsf{X}) \;=\; \mathsf{place}(B^*,\ \mathit{Frame},\ v,\ \mathsf{V})}$$

The function place takes care of placing boxes in a certain context and separates them with space-options. The first two equations handle this in case of an empty box list or a single box.

$$[6] \quad \mathsf{place}(,\ \mathit{Info}\ \&\ \mathit{Rm},\ s,\ \mathsf{X}) = ()\ \&\&\ \mathit{Info}$$

$$[7] \quad \mathsf{place}(B,\ \mathit{Frame},\ s,\ \mathsf{X}) = \mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X})$$

H context. This equation deals with horizontal context. The result of formatting the first box is concatenated with any horizontal space and the result of formatting the remaining boxes. The used space variable m is passed through and incremented with the applied horizontal space option.

$$[8] \quad \frac{\begin{array}{rcl}\mathsf{html}(B,\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{H}) &=& \mathit{Htext}_1\ \&\&\ (m,\ \mathit{Vmode}),\\ \mathsf{place}(B^+,\ (m+h,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ h,\ \mathsf{H}) &=& \mathit{Htext}_2\ \&\&\ (m_2,\ \mathit{Vmode}_2)\end{array}}{\mathsf{place}(B\ B^+,\ \mathit{Info}\ \&\ \mathit{Rm},\ h,\ \mathsf{H}) \;=\; (\mathit{Htext}_1\ \mathsf{hblank}(h)\ \mathit{Htext}_2)\ \&\&\ (m_2,\ \mathit{Vmode}_2)}$$

V context This equation deals with vertical context. The result of formatting the first box is concatenated with
 elements according to the vertical space option. The used space variable m is *not* passed through so during the formatting of V boxes it represents the horizontal offset at which the first box started. To maintain this indentation, horizontal space is added and concatenated with with the formatting of the remaining boxes This result is passed back along with the used space variable m of the formatted remaining boxes (which will equal the used space after the last remaining box has been formatted) .

$$[9] \quad \frac{\begin{array}{rcl}\mathsf{html}(B,\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{V}) &=& \mathit{Htext}_1\ \&\&\ \mathit{Info},\\ \mathsf{place}(B^+,\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ v,\ \mathsf{V}) &=& \mathit{Htext}_2\ \&\&\ (m_2,\ \mathit{Vmode}_2)\end{array}}{\mathsf{place}(B\ B^+,\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ v,\ \mathsf{V}) \;=\; (\mathit{Htext}_1\ \mathsf{vblank}(v)\ \mathsf{hblank}(m)\ \mathit{Htext}_2)\ \&\&\ (m_2,\ \mathsf{true})}$$

I boxes are formatted by incrementing the used space variable m with the space-option for indentation and adding horizontal space to indent the first line.

$$[10] \quad \frac{\begin{array}{rcl}i &=& \mathsf{I}(o^*),\\ \mathsf{html}(B,\ (m+i,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) &=& \mathit{Htext}\ \&\&\ \mathit{Info}\end{array}}{\mathsf{html}(\mathsf{I}\ o^*[B],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) \;=\; (\mathsf{hblank}(i)\ \mathit{Htext})\ \&\&\ \mathit{Info}}$$

WD boxes. The argument box is placed, the resulting text will be ignored except for the last line. The number of characters used by the last line is calculated and horizontal space is yielded with exactly the same length.

$$[11] \quad \frac{\begin{array}{rcl}\mathsf{html}(B,\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) &=& \mathit{Htext}\ \&\&\ (m_2,\ \mathit{Vmode}_2),\\ h &=& m_2 - m\end{array}}{\mathsf{html}(\mathsf{WD}[B],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) \;=\; \mathsf{hblank}(h)\ \&\&\ (m+h,\ \mathit{Vmode})}$$

HOV boxes are first formatted as a H box. If the result doesn't fit in the remaining space on the current line the box is reformatted as a V box.

$$[12]\quad \frac{\begin{array}{rcl}\mathsf{html}(\mathsf{H}\ o^*[B^*],\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{X}) & = & \mathit{Htext}\ \&\&\ (m,\ \mathit{Vmode}),\\ m \leq \mathit{Rm} & = & \mathsf{true},\\ \mathit{Vmode} & = & \mathsf{false}\end{array}}{\mathsf{html}(\mathsf{HOV}\ o^*[B^*],\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ \mathit{Htext}\ \&\&\ (m,\ \mathit{Vmode})}$$

$$[13]\quad \mathsf{html}(\mathsf{HOV}\ o^*[B^*],\ \mathit{Frame},\ \mathsf{X}) = \mathsf{html}(\mathsf{V}\ o^*[B^*],\ \mathit{Frame},\ \mathsf{X})\quad \textbf{otherwise}$$

HV boxes are the most complicated to format, because for each subbox we have to check whether it fits on the remaining line or should be placed at the start of a new line. The first two equations handle no argument boxes or a single argument box

$$[14]\quad \mathsf{html}(\mathsf{HV}\ o^*[],\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{X}) = ()\ \&\&\ \mathit{Info}$$
$$[15]\quad \mathsf{html}(\mathsf{HV}\ o^*[B],\ \mathit{Frame},\ \mathsf{X}) = \mathsf{html}(B,\ \mathit{Frame},\ \mathsf{H})$$

The first HV box is formatted without paying attention to the amount of space it uses. Space options are calculated only once and passed as HVInfo along with the current value of the used space.

$$[16]\quad \frac{\begin{array}{rcl}\mathit{HVInfo} & = & (\mathsf{H}(o^*),\ \mathsf{V}(o^*),\ \mathsf{I}(o^*),\ m),\\ \mathsf{html}(B,\ (m,\ \mathsf{false})\ \&\ \mathit{Rm},\ \mathsf{H}) & = & \mathit{Hstruct}\end{array}}{\mathsf{html}(\mathsf{HV}\ o^*[B\ B^+],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ \mathsf{hvplace}(\mathit{Hstruct},\ B^+,\ \mathit{HVInfo}\ \&\ \mathit{Rm})}$$

The formatting of the remaining boxes depends on the amount of space left on the current line. The boxes are formatted one by one but as soon as a box runs over the right margin its formatted text will be shifted to the next line A term hvplace(Htext & (m, Vmode),B+,(h,v,i,m2) & Rm) declares that on this line we have already set Htext and have used m characters of space. We still have to set boxes B+. The first box started at position *m2*, so if we shift to a new line we have (Rm - m2) characters left. If there are no more boxes left we are ready.

$$[17]\quad \mathsf{hvplace}(\mathit{Hstruct},\ ,\ \mathit{HVFrame}) = \mathit{Hstruct}$$

If there is at least one box left we set that box in a horizontal context on the current line. A term hvplace(Hstruct1, Hstruct2,B B*, HVInfo, Rm) declares that on this line we have already set Hstruct1. We have mapped box B to Hstruct2 (note that we remember box B in case we have to reformat it) and now we have to decide where we are going to put the formatted text.

$$[18]\quad \frac{\mathsf{html}(B,\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{H})\ =\ \mathit{Hstruct}}{\begin{array}{l}\mathsf{hvplace}(\mathit{Htext}\ \&\&\ \mathit{Info},\ B\ B^*,\ \mathit{HVInfo}\ \&\ \mathit{Rm})\ =\\ \mathsf{hvplace2}(\mathit{Htext}\ \&\&\ \mathit{Info},\ \mathit{Hstruct},\ B\ B^*,\ \mathit{HVInfo}\ \&\ \mathit{Rm})\end{array}}$$

Box B was mapped to the single line of text Htext2 and fits on the current line. We yield the previously formatted box (Htext) and Htext2 on the current line separated by space options.

$$[19]\quad \frac{\begin{array}{rcl}m + h \leq \mathit{Rm} & = & \mathsf{true},\\ \mathit{Hstruct} & = & (\mathit{Htext}\ \mathsf{hblank}(h)\ \mathit{Htext}_2)\ \&\&\ (m + h,\ \mathsf{false})\end{array}}{\begin{array}{l}\mathsf{hvplace2}(\mathit{Htext}\ \&\&\ \mathit{Info},\ \mathit{Htext}_2\ \&\&\ (m,\ \mathsf{false}),\ B\ B^*,\ (h,\ v,\ i,\ m_2)\ \&\ \mathit{Rm})\ =\\ \mathsf{hvplace}(\mathit{Hstruct},\ B^*,\ (h,\ v,\ i,\ m_2)\ \&\ \mathit{Rm})\end{array}}$$

Box B was mapped to the single line of text Htext2 but does not fit on the current line. We reformat box B for the next line and yield the previously formatted box (Htext) and Htext2 on different lines separated by space options.

$$[20]\quad \frac{\begin{array}{rcl} m + h > Rm & = & \mathsf{true}, \\ \mathsf{html}(B,\ (m_2 + i,\ \mathsf{false})\ \&\ Rm,\ \mathsf{H}) & = & Htext_3\ \&\&\ (m_3,\ Vmode_2), \\ Hstruct & = & (Htext\ \mathsf{vblank}(v)\ \mathsf{hblank}(m_2 + i)\ Htext_3)\ \&\&\ (m_3 + i,\ \mathsf{false}) \end{array}}{\begin{array}{l} \mathsf{hvplace2}(Htext\ \&\&\ Info,\ Htext_2\ \&\&\ (m,\ Vmode),\ B\ B^*,\ (h,\ v,\ i,\ m_2)\ \&\ Rm)\ = \\ \mathsf{hvplace}(Hstruct,\ B^*,\ (h,\ v,\ i,\ m_2)\ \&\ Rm) \end{array}}$$

Box B was mapped to multiple lines of text. We reformat box B for the next line and yield the previously formatted box (Htext) and Htext2 on different lines separated by space options.

$$[21]\quad \frac{\begin{array}{rcl} \mathsf{html}(B,\ (m_2 + i,\ \mathsf{false})\ \&\ Rm,\ \mathsf{H}) & = & Htext_3\ \&\&\ (m_3,\ Vmode), \\ Hstruct & = & (Htext\ \mathsf{vblank}(v)\ \mathsf{hblank}(m_2 + i)\ Htext_3)\ \&\&\ (m_3 + i,\ \mathsf{false}) \end{array}}{\begin{array}{l} \mathsf{hvplace2}(Htext\ \&\&\ Info,\ Htext_2\ \&\&\ (m,\ \mathsf{true}),\ B\ B^*,\ (h,\ v,\ i,\ m_2)\ \&\ Rm)\ = \\ \mathsf{hvplace}(Hstruct,\ B^*,\ (h,\ v,\ i,\ m_2)\ \&\ Rm) \end{array}}$$

A.2 Alignments-to-Html

Alignments are implemented using the <TABLE> element. We can translate alignments in a straightforward way to a HTML table. Each row is mapped to a <TR> (table row) element. All boxes in a row operator are mapped to <TD> (table data) element. The ALIGN attribute is set according to the values in A-OPTIONS. This has to be done for each cell in the table, because in HTML the alignment can not be set for columns but only for rows. This implementation does not support nested alignments. This is a design decision and not a limitation of HTML. We also do not evaluate not any space-options of the A and R operator as they are not yet employed by any of the generated formatters. The <TABLE> element features attributes for row- and columnspan and cellspacing, but there are no exact equivalents for the space options.

imports Alignments Box-to-Html$^{A.1}$
hiddens
  context-free syntax
    rows(A-OPTIONS, BOX-LIST, HTEXT-FRAME) → HTEXT
    elements(A-OPTIONS, BOX-LIST, BOX-INT) → HTEXT
hiddens
  variables
    [B-E][0-9]∗ → BOX
    [B-E]"+"*[0-9]*∗ → BOX+
    [B-E]"∗"*[0-9]*∗ → BOX∗
equations

This is the starting point for translating alignments. The values for used space and right margin are passed to the function rows which evaluates all R boxes inside the A box.

[1] $\mathsf{html}(\mathsf{A}\ (ao^*)\ so^*[B^*],\ Info\ \&\ Rm,\ \mathsf{X}) = (\texttt{<TABLE>}\ \mathsf{rows}((ao^*)\ ,\ B^*,\ Info\ \&\ Rm)\ \texttt{</TABLE>})\ \&\&\ Info$

For each row a <TR> element is inserted. A blank column is inserted with the exact width of the current position to maintain the proper indentation.

[2] $\mathsf{rows}((ao^*)\ ,\ ,\ Frame) = ()$

[3] $\text{rows}((ao^*)\ ,\ \text{R}[B^+],\ (m,\ Vmode)\ \&\ Rm) = (\texttt{<TR>}\ \texttt{<TD>}\ \text{hblank}(m)\ \text{elements}((ao^*)\ ,\ B^+,\ Rm))$

[4] $\text{rows}((ao^*)\ ,\ \text{R}[B^+]\ C^+,\ (m,\ Vmode)\ \&\ Rm) = (\texttt{<TR>}\ \texttt{<TD>}\ \text{hblank}(m)\ \text{elements}((ao^*)\ ,\ B^+,\ Rm)\ \text{rows}((ao^*)\ ,\ C^+,\ (m,\ Vmode)\ \&\ Rm))$

For each box in a row a <TD> element is inserted.

[5] $\text{elements}(()\ ,\ ,\ Rm) = ()$

[6] $$\frac{\text{html}(B,\ (0,\ \text{false})\ \&\ Rm,\ \text{H})\ =\ Htext\ \&\&\ (m,\ Bool)}{\text{elements}((\text{l}\ ,\ ao^*)\ ,\ B\ B^*,\ Rm)\ =\ (<\texttt{TD ALIGN} = \texttt{LEFT}>\ Htext\ \text{elements}((ao^*)\ ,\ B^*,\ Rm))}$$

[7] $$\frac{\text{html}(B,\ (0,\ \text{false})\ \&\ Rm,\ \text{H})\ =\ Htext\ \&\&\ (m,\ Bool)}{\text{elements}((\text{c}\ ,\ ao^*)\ ,\ B\ B^*,\ Rm)\ =\ (<\texttt{TD ALIGN} = \texttt{CENTER}>\ Htext\ \text{elements}((ao^*)\ ,\ B^*,\ Rm))}$$

[8] $$\frac{\text{html}(B,\ (0,\ \text{false})\ \&\ Rm,\ \text{H})\ =\ Htext\ \&\&\ (m,\ Bool)}{\text{elements}((\text{r}\ ,\ ao^*)\ ,\ B\ B^*,\ Rm)\ =\ (<\texttt{TD ALIGN} = \texttt{RIGHT}>\ Htext\ \text{elements}((ao^*)\ ,\ B^*,\ Rm))}$$

A.3 Fonts-to-Html

This module defines a translation for the font operator. The font operator F can be followed by one or more font options. We provide a translation for the parameters font name (*fn*), font size (*sz*) and color (*cl*). Font operator abbreviations can be used to abstract from actual typesetting, by providing logical styles for mathematics (*MATH*), variables (*VAR*), keywords (*KW*) and numbers (*NUM*).

imports Box-to-Html$^{A.1}$ Fonts
hiddens
variables
[B-E][0-9]∗ → BOX
[B-E]"+"[0-9]∗ → BOX+
[B-E]"∗"[0-9]∗ → BOX∗
"chr∗"[0-9']∗ → CHAR∗
equations

Font Operator Font properties are translated to the appropriate HTML elements. The FACE attribute specifies a specific font, like courier or times. If the appropriate font is not installed on the computer which is used to view the document, the browser will use the default font.

[1] $$\frac{\text{html}(B,\ Frame,\ \text{X})\ =\ Htext\ \&\&\ Info}{\text{html}(\text{F fn} = \texttt{KWf}[B],\ Frame,\ \text{X})\ =\ (\texttt{<B>}\ Htext\ \texttt{</B>})\ \&\&\ Info}$$

[2] $$\frac{\text{html}(B,\ Frame,\ \text{X})\ =\ Htext\ \&\&\ Info}{\text{html}(\text{F fn} = \text{fid}(chr^*)[B],\ Frame,\ \text{X})\ =\ (<\texttt{FONT FACE} = \text{id}(chr^*)>\ Htext\ \texttt{</FONT>})\ \&\&\ Info} \quad \textbf{otherwise}$$

The SIZE attribute specifies font size.

$$[3] \quad \frac{\mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{html}(\mathsf{F\ sz} = \mathit{Int}[B],\ \mathit{Frame},\ \mathsf{X})\ =\ (<\texttt{FONT SIZE} = \mathit{Int} > \mathit{Htext}\ \texttt{</FONT>})\ \&\&\ \mathit{Info}}$$

The COLOR attribute can be a RGB triplet (e.g. "FF0000") or a color name

$$[4] \quad \frac{\mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{html}(\mathsf{F\ cl} = \mathsf{fid}(\mathit{chr}^*)[B],\ \mathit{Frame},\ \mathsf{X})\ =\ (<\texttt{FONT COLOR} = \mathsf{id}(\mathit{chr}^*) > \mathit{Htext}\ \texttt{</FONT>})\ \&\&\ \mathit{Info}}$$

We define the abbreviations as follows:

$$[5] \quad \mathsf{html}(\mathsf{MATH}[a],\ \mathit{Frame},\ \mathsf{X}) = \mathsf{html}(a,\ \mathit{Frame},\ \mathsf{X})$$

$$[6] \quad \frac{\mathsf{html}(a,\ \mathit{Frame},\ \mathsf{X})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{html}(\mathsf{VAR}[a],\ \mathit{Frame},\ \mathsf{X})\ =\ (\texttt{<EM>}\ \mathit{Htext}\ \texttt{</EM>})\ \&\&\ \mathit{Info}}$$

$$[7] \quad \frac{\mathsf{html}(a,\ \mathit{Frame},\ \mathsf{X})\ =\ \mathit{Htext}\ \&\&\ \mathit{Info}}{\mathsf{html}(\mathsf{KW}[a],\ \mathit{Frame},\ \mathsf{X})\ =\ (\texttt{<B>}\ \mathit{Htext}\ \texttt{</B>})\ \&\&\ \mathit{Info}}$$

$$[8] \quad \mathsf{html}(\mathsf{NUM}[a],\ \mathit{Frame},\ \mathsf{X}) = \mathsf{html}(a,\ \mathit{Frame},\ \mathsf{X})$$

The escape operator can be used to include special characters or mathematical symbols which are mapped to HTML entity references. However, the characters mentioned below are also handled by the 'specials' function if they are not escaped. This list can be extended to include characters for a specific purpose.

$$\begin{array}{ll}
[9] & \mathsf{html}(\mathsf{ESC}[\texttt{">"}],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) = \texttt{\>}\ \&\&\ (m+1,\ \mathit{Vmode}) \\
[10] & \mathsf{html}(\mathsf{ESC}[\texttt{"<"}],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) = \texttt{\<}\ \&\&\ (m+1,\ \mathit{Vmode}) \\
[11] & \mathsf{html}(\mathsf{ESC}[\texttt{"\&"}],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) = \texttt{\&}\ \&\&\ (m+1,\ \mathit{Vmode}) \\
[12] & \mathsf{html}(\mathsf{ESC}[a],\ \mathit{Frame},\ \mathsf{X}) \qquad\qquad\qquad = \mathsf{html}(a,\ \mathit{Frame},\ \mathsf{X}) \quad \textbf{otherwise}
\end{array}$$

A.4 Comments-to-Html

This module formats the comment-operators. To format the comment text we use a special version of the equations for translating HV boxes. Because of the convention we have introduced for the comment markers, we do not (yet) support end-markers. This implies that a HPAR operator that appears *between* symbols is not allowed.

imports Box-to-Html$^{A.1}$ Comments
hiddens
  context-free syntax
    checkcomment(BOX-STRING) → BOX-BOOL
    parplace(BOX-LIST, HTEXT-FRAME, CONTEXT) → HTEXT-STRUCT
    parplace2(HTEXT-STRUCT, BOX-LIST, BOX-INT, BOX-INT) → HTEXT-STRUCT
    parplace3(HTEXT-STRUCT, HTEXT-STRUCT, BOX-LIST, BOX-INT, BOX-INT) → HTEXT-STRUCT
  variables

$[B\text{-}E][0\text{-}9]* \rightarrow$ BOX
$[B\text{-}E]$ "+" $[0\text{-}9]* \rightarrow$ BOX+
$[B\text{-}E]$ "*" $[0\text{-}9]* \rightarrow$ BOX∗

equations

This function checks the comment marker to see if we should process the comment.

[1] checkcomment("%%H%") = true
[2] checkcomment("%%A%") = true
[3] checkcomment(a) = false **otherwise**

The function parplace typesets comment text. A single line of comment text is prefixed with the default comment marker.

[4] parplace(, $Info$ & Rm, X) = () && $Info$

[5] $$\frac{\mathsf{html}(B,\ (m+3,\ Vmode)\ \&\ Rm,\ \mathrm{X})\ =\ Htext\ \&\&\ Info}{\mathsf{parplace}(B,\ (m,\ Vmode)\ \&\ Rm,\ \mathrm{X})\ =\ (\mathsf{rmquotes}(\texttt{"\%\%"})\ Htext)\ \&\&\ Info}$$

Subsequent lines of comment text are handed to the function parplace2.

[6] $$\frac{\begin{array}{r}\mathsf{html}(B,\ (m+2,\ Vmode)\ \&\ Rm,\ \mathrm{H})\ =\ Hstruct,\\ \mathsf{parplace2}(Hstruct,\ B^{+},\ m,\ Rm)\ =\ Htext\ \&\&\ Info\end{array}}{\mathsf{parplace}(B\ B^{+},\ (m,\ Vmode)\ \&\ Rm,\ \mathrm{X})\ =\ (\mathsf{rmquotes}(\texttt{"\%\%"})\ Htext)\ \&\&\ Info}$$

A term parplace2(Htext & (m,Vmode),B+,m2,Rm) declares that on this line we have already set Htext and have used m characters of space. If there are no more boxes left we are ready and yield Htext. We remember the starting point $m2$ of the first line in case we have to insert a new line.

[7] parplace2($Hstruct$, , m, Rm) = $Hstruct$

If there is at least one line of comment text we process this line. A term parplace3(Hstruct1, Hstruct2, B*, m, Rm) declares that on this line we have already set Hstruct1 and have mapped the next line to Hstruct2.

[8] $$\frac{\mathsf{html}(B,\ Info\ \&\ Rm,\ \mathrm{H})\ =\ Hstruct}{\mathsf{parplace2}(Htext\ \&\&\ Info,\ B\ B^{*},\ m,\ Rm)\ =\ \mathsf{parplace3}(Htext\ \&\&\ Info,\ Hstruct,\ B^{*},\ m,\ Rm)}$$

Now we have to decide where we put the current line. If the current comment line still fits on this line we yield the previous and current comment text horizontally.

[9] $$\frac{\begin{array}{r}m \le Rm\ =\ \mathsf{true},\\ Hstruct\ =\ (Htext\ Htext_2)\ \&\&\ (m+1,\ Vmode)\end{array}}{\mathsf{parplace3}(Htext\ \&\&\ Info,\ Htext_2\ \&\&\ (m,\ Vmode),\ B^{*},\ m_2,\ Rm)\ =\ \mathsf{parplace2}(Hstruct,\ B^{*},\ m_2,\ Rm)}$$

If the current comment line does not fit on this line, we move it to the next line and prefix it with a default comment marker.

[10] $$\frac{\begin{array}{r}m_2 > Rm\ =\ \mathsf{true},\\ Hstruct\ =\ (Htext\ \texttt{
}\ \mathsf{hblank}(m_3)\ \mathsf{rmquotes}(\texttt{"\%\%"})\ Htext_2)\ \&\&\ (m_3+(m_2-m),\ Vmode_2)\end{array}}{\mathsf{parplace3}(Htext\ \&\&\ (m,\ Vmode),\ Htext_2\ \&\&\ (m_2,\ Vmode_2),\ B^{*},\ m_3,\ Rm)\ =\ \mathsf{parplace2}(Hstruct,\ B^{*},\ m_3,\ Rm)}$$

Comment operators These equations handle the different comment operators. A PAR comment is started in the first column of a new line.

$$[11]\quad \frac{\begin{array}{rcl}\mathsf{checkcomment}(a) &=& \mathsf{true},\\ \mathsf{parplace}(B^*,\ (0,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X}) &=& \mathit{Htext}\ \&\&\ \mathit{Info}\end{array}}{\mathsf{html}(\mathsf{PAR}\ a\ b[B^*],\ (m,\ \mathit{Vmode})\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ (\texttt{
}\ \texttt{<EM>}\ \mathit{Htext}\ \texttt{</EM>})\ \&\&\ \mathit{Info}}$$

A VPAR comment is formatted at the current position.

$$[12]\quad \frac{\begin{array}{rcl}\mathsf{checkcomment}(a) &=& \mathsf{true},\\ \mathsf{parplace}(B^*,\ \mathit{Frame},\ \mathsf{X}) &=& \mathit{Htext}\ \&\&\ \mathit{Info}\end{array}}{\mathsf{html}(\mathsf{VPAR}\ a\ b[B^*],\ \mathit{Frame},\ \mathsf{X})\ =\ (\texttt{<EM>}\ \mathit{Htext}\ \texttt{</EM>})\ \&\&\ \mathit{Info}}$$

A HPAR is also formatted at the current position

$$[13]\quad \frac{\begin{array}{rcl}\mathsf{checkcomment}(a) &=& \mathsf{true},\\ \mathsf{parplace}(B^*,\ \mathit{Frame},\ \mathsf{X}) &=& \mathit{Htext}\ \&\&\ \mathit{Info}\end{array}}{\mathsf{html}(\mathsf{HPAR}\ a\ b[B^*],\ \mathit{Frame},\ \mathsf{X})\ =\ (\texttt{<EM>}\ \mathit{Htext}\ \texttt{</EM>})\ \&\&\ \mathit{Info}}$$

If the comment marker is not recognized, the comments are skipped.

$$[14]\quad \frac{\mathsf{checkcomment}(a)\ =\ \mathsf{false}}{\mathsf{html}(\mathsf{PAR}\ a\ b[B^*],\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ ()\ \&\&\ \mathit{Info}}$$

$$[15]\quad \frac{\mathsf{checkcomment}(a)\ =\ \mathsf{false}}{\mathsf{html}(\mathsf{VPAR}\ a\ b[B^*],\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ ()\ \&\&\ \mathit{Info}}$$

$$[16]\quad \frac{\mathsf{checkcomment}(a)\ =\ \mathsf{false}}{\mathsf{html}(\mathsf{HPAR}\ a\ b[B^*],\ \mathit{Info}\ \&\ \mathit{Rm},\ \mathsf{X})\ =\ ()\ \&\&\ \mathit{Info}}$$

A.5 Crossreferences-to-Html

This module describes the mapping of box crossreference operators to hypertext links and anchors. The LBL operator is used to designate a box term as the destination of a link (hypertext anchor). The REF operator is mapped to a hypertext link. The first argument of the LBL or REF operator contains a string which is used to set the value of the appropriate anchor or link attribute. This string must start with "`A:`" to identify it as a hypertext label.

imports Crossreferences Box-to-Html$^{A.1}$
hiddens
  context-free syntax
    checklabel(BOX-STRING) → BOX-STRING
  variables
    "*chr**" → CHAR*
    *[B-E][0-9]** → BOX
    [B-E]"+"*[0-9]** → BOX+
    [B-E]"*"*[0-9]** → BOX*
equations

Links The LBL operator is translated to a target location, by setting the NAME attribute of the <A> element with the relevant part of the string argument of the operator.

$$[1] \quad \frac{\mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X}) = \mathit{Htext}\ \&\&\ \mathit{Info},\quad \mathsf{checklabel}(a) = b,\quad b \neq \texttt{""}}{\mathsf{html}(\mathsf{LBL}[a,\ B],\ \mathit{Frame},\ \mathsf{X}) = (<\texttt{A NAME} = b > \mathit{Htext}\ \texttt{</A>})\ \&\&\ \mathit{Info}}$$

[2] $\mathsf{html}(\mathsf{LBL}[a,\ B],\ \mathit{Frame},\ \mathsf{X}) = \mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X})$ **otherwise**

The REF operator is translated to a hypertext link, by setting the HREF attribute of the <A> element. The relevant part of the string argument is treated as an URL. This implies that the front-end formatter has to decide if it generates local (e.g. `"#label"`) or external (e.g. `"filename"` or `"www address"`) references. This is a design choice.

$$[3] \quad \frac{\mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X}) = \mathit{Htext}\ \&\&\ \mathit{Info},\quad \mathsf{checklabel}(a) = b,\quad b \neq \texttt{""}}{\mathsf{html}(\mathsf{REF}[a,\ B],\ \mathit{Frame},\ \mathsf{X}) = (<\texttt{A HREF} = b > \mathit{Htext}\ \texttt{</A>})\ \&\&\ \mathit{Info}}$$

[4] $\mathsf{html}(\mathsf{REF}[a,\ B],\ \mathit{Frame},\ \mathsf{X}) = \mathsf{html}(B,\ \mathit{Frame},\ \mathsf{X})$ **otherwise**

This function checks a label to see if it starts with "`A:`". If so, the remainder of the string is returned.

[5] $\mathsf{checklabel}(\mathsf{box\text{-}str\text{-}con}(\texttt{"""}\ \texttt{"A"}\ \texttt{":"}\ \mathit{chr}^*\ \texttt{"""})) = \mathsf{box\text{-}str\text{-}con}(\texttt{"""}\ \mathit{chr}^*\ \texttt{"""})$
[6] $\mathsf{checklabel}(a) = \texttt{""}$ **otherwise**

A.6 Over-to-Html

This module implements the box operator O which was added for the specific purpose of typesetting conditional equations in ASF+SDF specifications. This operator accepts two argument boxes and a separation character.

imports Over Box-Int-Op Box-to-Html$^{A.1}$
hiddens
variables
*[B-E][0-9]** → BOX
[B-E]"+"*[0-9]** → BOX+
[B-E]"*"*[0-9]** → BOX*
*chr[0-9]** → CHAR
chr"*"*[0-9]** → CHAR*
equations

First, box B1 is formatted. The length of the last line of the resulting hypertext is used to construct a string of separation characters with exactly this length. Finally, the result of formatting

B1, the separation string and B2 is yielded in vertical context.

$$[1]\quad \frac{\begin{array}{rcl} \mathsf{html}(B_1, \mathit{Frame}, \mathsf{X}) & = & \mathit{Htext}\ \&\&\ (m, \mathit{Vmode}), \\ \mathsf{html}(B_2, \mathit{Frame}, \mathsf{X}) & = & \mathit{Htext}_2\ \&\&\ (m_2, \mathit{Vmode}_2), \\ \text{string of } \mathsf{max}(m, 3)\ \text{box-str-con}(\texttt{"""}\ \mathit{chr}\ \texttt{"""})\ \text{chars} & = & a, \\ v & = & \mathsf{V}(o^*) \end{array}}{\begin{array}{l} \mathsf{html}(\mathsf{O}\ o^*[B_1\ \text{box-str-con}(\texttt{"""}\ \mathit{chr}\ \mathit{chr}^*\ \texttt{"""})\ B_2], \mathit{Frame}, \mathsf{X}) = \\ (\mathit{Htext}\ \mathsf{vblank}(v)\ \mathsf{rmquotes}(a)\ \mathsf{vblank}(v)\ \mathit{Htext}_2)\ \&\&\ (m_2, \mathit{Vmode}_2) \end{array}}$$

Appendix B

HTML element list

This is an elementary reference for all HTML elements generated by Box to HTML. Elements shown in italics are *not* part of the HTML 3.2 standard.

Element name	**Description**
`<!DOCTYPE>`	SGML prologue identifying the document
`<!-->`	SGML comment
`<HTML>`	Indicates start of HTML document
`<HEAD>`	Indicates head section, which describes properties of the document
`<META>`	Used to include name/value pairs describing properties
`<TITLE>`	Identifies the document title
`<BODY>`	Indicates the body section, which contains the actual markup
`<A>`	Anchor or hypertext link. Attributes NAME and HREF
`<B>`	Boldface
` `	Break. Inserts a newline
`<EM>`	Emphasize. Usually rendered in italics
`<FONT>`	Sets various font properties. Attributes are FACE, SIZE and COLOR
<FRAME>	Invokes a document in a frameset. Attributes are SRC, NAME
<FRAMESET>	Indicates start of frame layout. Attribute ROWS
`<I>`	Italic text
<NOFRAMES>	Alternative text for browsers not supporting frames
<NOBR>	No break. Prevents automatic linebreaking by the browser
`<TABLE>`	Indicates start of table environment
`<TD>`	Table data. Defines a table cell. Attribute ALIGN
`<TR>`	Table row. Defines a row
`<UL>`	Unordered list. Used for indenting
	non-breaking space (also #&160;)
"	quotation mark (also #&34;)
<	less than sign (also #&60;)
>	greater than sign (also #&62;)
&	ampersand (also #&38;)

Appendix C

HTML document source

C.1 Box Booleans

This is the generated HTML code for the Box Booleans module as shown in Figure 6.2. The space characters between HTML elements are introduced by the ASF+SDF term prettyprinter.

C.1.1 SDF part

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD> <META NAME = "GENERATOR" CONTENT = "Box2HTML"> </HEAD>
<BODY>
<NOBR> <B> imports </B> <A HREF = "Layout.html"> Layout </A>
</NOBR> <BR> <B> exports </B> <BR>
    <NOBR> <B> sorts </B> BOX-BOOL BOX-BOOL-LIST </NOBR> <BR>
    <NOBR> <B> context-free </B> <B> syntax </B> </NOBR>
<TABLE>
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> true </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> BOX-BOOL
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> false </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> BOX-BOOL
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> BOX-BOOL &quot;|&quot; BOX-BOOL </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> <NOBR> BOX-BOOL <NOBR> { <B> left </B> } </NOBR> </NOBR>
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> BOX-BOOL or BOX-BOOL </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> <NOBR> BOX-BOOL <NOBR> { <B> left </B> } </NOBR> </NOBR>
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> BOX-BOOL &quot;&amp;&quot; BOX-BOOL </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> <NOBR> BOX-BOOL <NOBR> { <B> left </B> } </NOBR> </NOBR>
<TR> <TD>        
```

```
<TD ALIGN = LEFT> <NOBR> BOX-BOOL and BOX-BOOL </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> <NOBR> BOX-BOOL <NOBR> { <B> left </B> } </NOBR> </NOBR>
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> not BOX-BOOL </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> BOX-BOOL
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> &quot;(&quot; BOX-BOOL &quot;)&quot; </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> <NOBR> BOX-BOOL <NOBR> { <B> bracket </B> } </NOBR> </NOBR>
<TR> <TD>        
<TD ALIGN = LEFT> <NOBR> &quot;{&quot; <NOBR> { <NOBR> BOX-BOOL &quot;,&quot;
</NOBR> } * </NOBR> &quot;}&quot; </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> BOX-BOOL-LIST
</TABLE> <BR>
    <B> variables </B>
<TABLE>
 <TR> <TD>        
<TD ALIGN = LEFT> <NOBR> Bool <NOBR> [0-9'] * </NOBR> </NOBR>
<TD ALIGN = CENTER> -&gt;
<TD ALIGN = LEFT> BOX-BOOL
</TABLE>
    <B> priorities </B> <BR>
        <NOBR> { <B> left </B> <NOBR> : <NOBR>
&quot;|&quot; , </NOBR> or </NOBR> } </NOBR> <BR>
        <NOBR> &lt; <NOBR> { <B> left </B> <NOBR> : <NOBR>
&quot;&amp;&quot; , </NOBR> and </NOBR> } </NOBR> </NOBR> <BR>
        <NOBR> &lt; not </NOBR>
</BODY> </HTML>
```

C.1.2 ASF part

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD> <META NAME = "GENERATOR" CONTENT = "Box2HTML"> </HEAD>
<BODY>
<B> equations </B> <BR> <BR>
    <NOBR> [1] <NOBR> true | Bool = </NOBR> true </NOBR> <BR> <BR>
    <NOBR> [2] <NOBR> false  | Bool = </NOBR> Bool </NOBR> <BR> <BR>
    <NOBR> [3] <NOBR> true &amp; Bool = </NOBR> Bool </NOBR> <BR> <BR>
    <NOBR> [4] <NOBR> false &amp; Bool = </NOBR> false </NOBR> <BR> <BR>
    <NOBR> [5] <NOBR> not false = </NOBR> true </NOBR> <BR> <BR>
    <NOBR> [6] <NOBR> not true = </NOBR> false </NOBR> <BR> <BR>
    <NOBR> [6] <NOBR> Bool1 or Bool2 = </NOBR> Bool1 | Bool2 </NOBR> <BR> <BR>
    <NOBR> [7] <NOBR> Bool1 and Bool2 = </NOBR> Bool1 &amp; Bool2 </NOBR> <BR> <BR>
    <NOBR> [11] <NOBR> { <NOBR> true , </NOBR> <NOBR> false , </NOBR> true } = </NOBR>
{ <NOBR> true , </NOBR> <NOBR> true , </NOBR> false } </NOBR>
</BODY> </HTML>
```

C.2 Pico program

This is the generated HTML code for the Pico program shown in Figure 6.3.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD> <META NAME = "GENERATOR" CONTENT = "Box2HTML"> </HEAD>
<BODY>
<BR>
<EM> %% <NOBR> This small pico program calculates 3! </NOBR> </EM><BR>
<B> begin </B> <BR>
    <B> declare </B> <BR>
        <NOBR> <A NAME = "input"> input </A> : <B> natural
</B> , </NOBR> <BR>
        <NOBR> <A NAME = "output"> output </A> : <B> natural
</B> , </NOBR> <BR>
        <NOBR> <A NAME = "repnr"> repnr </A> : <B> natural
</B> , </NOBR> <BR>
        <A NAME = "rep"> rep </A> : <B> natural </B> <BR>
    ; <BR>
    <NOBR> <A HREF = "#input"> input </A>  :=  3 ; </NOBR> <BR>
    <NOBR> <A HREF = "#output"> output </A> :=  1 ; </NOBR> <BR>
    <B> while </B> <BR>
        <NOBR> <A HREF = "#input"> input </A> - 1 </NOBR> <BR>
    <B> do </B> <BR>
        <NOBR> <A HREF = "#rep"> rep </A> := <A HREF = "#output">
output </A> ; </NOBR> <BR>
        <NOBR> <A HREF = "#repnr"> repnr </A> := <A HREF = "#input">
input </A> ; </NOBR> <BR>
        <NOBR> <B> while </B><BR>
            <NOBR> <A HREF = "#repnr"> repnr </A> - 1 </NOBR> <BR>
        <B> do </B> <BR>
            <NOBR>
<A HREF = "#output"> output </A> := <NOBR> <A HREF = "#output">
output </A> + <A HREF = "#rep"> rep </A> </NOBR> ; </NOBR> <BR>
            <A HREF = "#repnr"> repnr </A> := <NOBR>
<A HREF = "#repnr"> repnr </A> - 1 </NOBR> <BR>
        <B> od </B> ; </NOBR> <BR>
        <A HREF = "#input"> input </A> := <NOBR> <A HREF = "#input">
input </A> - 1 </NOBR> <BR>
    <B> od </B> <BR>
<B> end </B>
</BODY> </HTML>
```

Bibliography

[Bar96] S.L. Barth. A rule-driven front-end for prettyprinting asfix. Master's thesis, Programming Research Group, University of Amsterdam, 1996.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BHK+94] J. Brown, J. Harris, L. Karbiner, et al. Hypercode. Technical report, MIT Artificial Intelligence Laboratory, september 1994.

[BL89] T. Berners-Lee. Information management: A proposal, March 1989. CERN, available from http://www.w3.org/History/1989/proposal.html.

[BLe95] T. Berners-Lee and D. Connolly ed. Html 2.0 specification, november 1995. Produced by the HTML Working Group of the IETF, available from ftp://ds.internic.net/rfc/rfc1866.txt.

[Bru95] J.J. Brunekreef. *On Modular Algebraic Protocol Specification*. PhD thesis, University of Amsterdam, 1995.

[BV94] M.G.J. van den Brand and E. Visser. From box to tex: An algebraic approach to the construction of documentation tools. Technical report p9420, Programming Research Group, University of Amsterdam, 1994.

[BV95] M.G.J. van den Brand and E. Visser. Box: Language, laws and formatters (version 1.4). Technical report, Programming Research Group, University of Amsterdam, June 1995. Technical documentation.

[BV96] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[Gol90] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual,* version 6 December. 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. ftp://ftp.cwi.nl/pub/gipe/reports/SDFmanual.ps.Z.

[Hil96] J.A. Hillebrand. A small language for the specification of grid protocols. Technical report, Programming Research Group, University of Amsterdam, 1996.

[ISO86] Information processing – text and office systems - standard generalized markup language (sgml), 1986. Available from http://www.iso.ch/cate/d16387.html.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[MCC86] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

[Rag97] D. Raggett. Html 3.2 reference specification, january 1997. W3C, available from http://www.w3.org/TR/REC-html32.html.

[Res94] M. Res. A generated programming environment RISLA, a specification language for defining financial products. Master's thesis, Programming Research Group, University of Amsterdam, 1994.

[SMB94] C.M. Sperberg-McQueen and Lou Burnhard. Guidelines for electronic text encoding and interchange. 1994. http://www-tei.uic.edu/orgs/tei/.

[Vis95] E. Visser. *ASF+SDF to LaTeX*. Programming Research Group, University of Amsterdam, 1995. http://adam.fwi.uva.nl/˜visser/tolatex/ (WWW page with references to software and manuals).

[Vos90] K.J. Vos. Pretty for an easy touch of beauty. Master's thesis, Programming Research Group, University of Amsterdam, 1990.

[WLB96] H Wium Lie and B. Bos. Cascading style sheets, level 1 (w3c recommendation), 1996. Available from http://www.w3.org/TR/REC-CSS1.

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.wins.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.wins.uva.nl`,
directory `pub/programming-research/reports/`.

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*