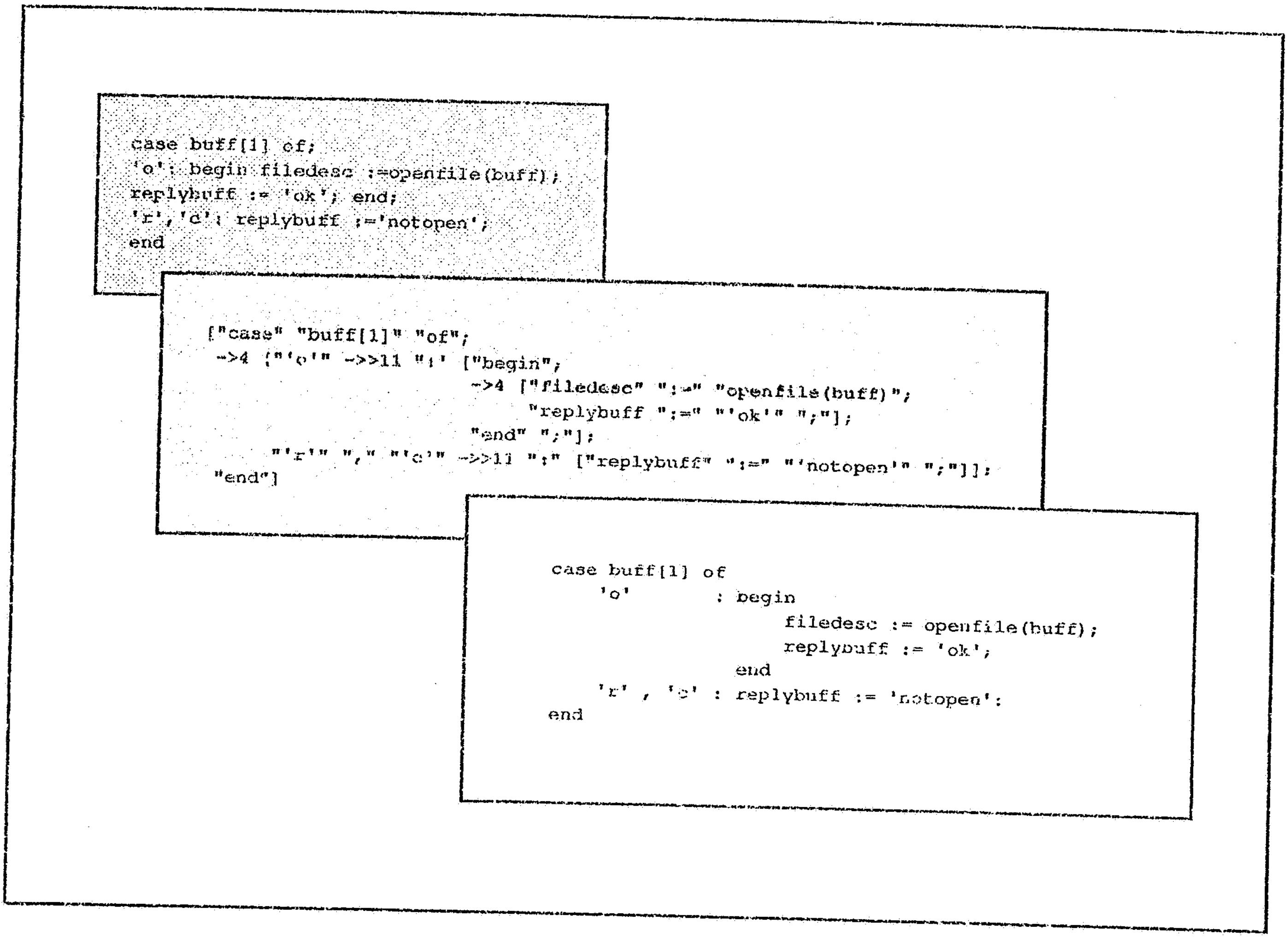
Pretty

for an easy touch of beauty



K.J.Vos

january 1990

Pretty

for an easy touch of beauty

K.J. Vos

Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Pretty is developed for the GIPE-project (Generation of Interactive Programming Environments). The aim of this project is to create a language-independent generator of programming environments. When specifications of the properties of a language are given to this generator, it can create a programming environment especially for this language. Pretty is used for specifying how the language structures must be printed on the screen. Pretty is based on the box-language of PPML, a prettyprint-language also developed in the GIPE-project. In this thesis we will look at prettyprinting in general, the prettyprint-language PPML, the arguments for designing yet another prettyprinting language, and the new language Pretty itself. We will give example specifications for the prettyprinting of PICO, a small toy language, and for the prettyprinting of Pascal. We will also discuss the box-interpreter, which translates terms written in the box-language Pretty into real text.

Table of contents

Introduction

GIPE (Generation of Interactive Programming Environments) is a research-project of the European research program ESPRIT, in which the CWI (Centre for Mathematics and Computer Science) and the University of Amsterdam participate. The goal of the project is to find a way to create solely on basis of a definition of a language a programming environment for this language. A programming environment is a set of programs that supports the user when he wants to write programs in a specific language. It consists of an editor, a parser, an typechecker, a compiler, a debugger and a prettyprinter for this language. Using a programming environment has great advantages. All parts of it are specialised for one language so this extra knowledge can be used to help the user better. For example, the editor can be completely syntax-directed, which means that it tells the user continually what language structures he can use at the point he's working on. He can only choose from those language structures, so he can not choose wrongly. This will guarantee that his program will stay syntactically correct all the time.

It is very costly to build a programming environment, let alone for every programming language a specific one. This raises the question whether it is possible to identify the similarities and differences between various language-specific programming environments. Then one will need to design and implement the language-independent aspects only once. This leads to a generator of programming environments which is parameterised by a definition of the desired language. Consequently, not only a generator must be created, but also new formalisms in which a specificator of a programming environment for a specific language can define the properties of this language.

The aim of the GIPE project is to develop such a generator of interactive programming environments. Interactive means that the environment gives direct responses, so for example a user can stop a command while it is running. It also means that all parts of the environment are integrated to form one environment with which the user can create, manipulate and compile structured objects. He doesn't have to be aware which part of the environment he is using during his actions. It is obvious that an interactive environment requires a good user-interface.

This thesis describes the development of a prettyprinting formalism and a supporting prettyprinter for the GIPE generator. Prettyprinting is placing the text that is hidden in an internal representation on the screen, and if possible, in a readable way. The internal representations used in most programming environments are tree structures. The parser creates an abstract syntax tree for the text the user created and all other parts of the environment will refer to this tree and manipulate it or create a new tree. The prettyprinter must put those trees back into readable text for the user.

In this way a prettyprinter can also be used to transform a text the user made into a text that is more beautiful and more readable. The user can type in his text rather carelessly, no indentation, no worry if the form of the text resembles the structure of the program. The parser will create a tree for this text and then the prettyprint program will work on it and put in the necessary layout and indentation to make the program readable.

A generator has no knowledge of properties of particular languages, so it also does not know how to prettyprint a text in a specific language. The prettyprinter of the generator can only deal with some language-independent aspects, like determining when a word must be placed on the next line. Most of these things have to do with the fact that in the end the prettyprinted text must be placed on a screen with limited dimensions. So this prettyprinter will deal with the low-level, but sometimes difficult and time-consuming, calculations to put every string of the text on the right place.

A specificator of a language will have to specify how he wants each language structure to be printed on the screen globally. This is usually done by giving for every language structure a format written in a box-language. A box-language is a very important concept for language-independent prettyprinting. It is a small language that describes atomic boxes, usually only strings, and operators on boxes. An operator can indicate for example that the boxes must be placed next to each other or, a very difficult one, that the boxes must be placed under each other with the "="- signs forming one column. Here we abstract from the contents of the boxes, whether it are atomic boxes or composite boxes, we are only concerned with the position of boxes to each other.

The box-language is an intermediate language. On one side the specificator can define the global indentation for every language structure by giving a box-term for each one. Now one box-term can be created for a whole text by recursively matching the language structures in the text to these rules and thus creating one great composite box-term. On the other side the language-independent part of the prettyprinter, in fact the box-interpreter, can take this box-term and determine the real position on the screen of the subboxes according to the operators used in the box-term. When all strings in the subboxes have their new position, the complete prettyprinted text is formed.

We will encounter the following existing formalisms:
- SDF (Syntax Definition Formalism) for defining concrete and abstract syntax;
- PPML (Pretty Print Meta-Language) for defining prettyprinting rules;
- ASF (Algebraic Specification Formalism) for defining the semantics of languages;
- ASF + SDF : the amalgamation of ASF and SDF.

PPML is a prettyprint language that not only consists of a box-language for describing the formats, but also offers ways to describe pattern-matching, simple functions and control structures. However in this thesis we did not want to design a whole new language for prettyprinting, we only wanted to specify a box-language in SDF, because ASF+SDF already offers ways to do pattern-matching and to make very complex functions and control structures. So we did not use PPML plain, but we only specified the box-language of PPML in SDF. We also made some changes to improve the readability and to make use of the complex functions that can be described in ASF+SDF.

We can now summarize the goals of this project as follows
- Create a prettyprinting formalism on top of ASF + SDF that profits from all features that are already available in that formalism
- Give a formal description of the semantics of the new prettyprint formalism
- Apply the prettyprint formalisms in some case studies.

In chapter one we will explain more about prettyprinting, how it is usually done by a one-language prettyprinter and how the generator uses a box-language to handle prettyprinting. In this chapter we will also give a short description of PPML and show some parts of a prettyprint-specification in PPML for the small language PICO. Chapter two will contain a full description of the new box-language Pretty and two example prettyprint-specifications, one also for the small language PICO and one for the more serious language Pascal. At the end of this chapter an evaluation of Pretty is given. In chapter three we will show how the box-interpreter works by explaining the global set-up of the specification of the box-interpreter. Finally we will evaluate the results so far and indicate what still must be done.

Chapter One: Prettyprinting

In this chapter we will first explain what prettyprinting is about and how it usually is done. Then we will say something about box-languages, languages in which the global layout of a language structure can be described, and next we will see how a box-language can be used in a generator to separate the language-specific and language-independent prettyprinting. Finally, we will discuss PPML, the language on which the new box-language is based.

1.1 common prettyprinting

First, we will explain how a common prettyprinter works. Let's see how a prettyprinter for the language Pascal works. Usually, a parser for Pascal comes first. It looks at the strings of the program that must be prettyprinted, one by one, and examines which language structure (i.e. if-then-else, assignment) the string belongs to. So the parser knows all the language structures of Pascal, formally said the syntax of Pascal. When the parser has found the language structures that are used to construct the text, it creates a tree that shows the text and the language structures in it. Here is an example.

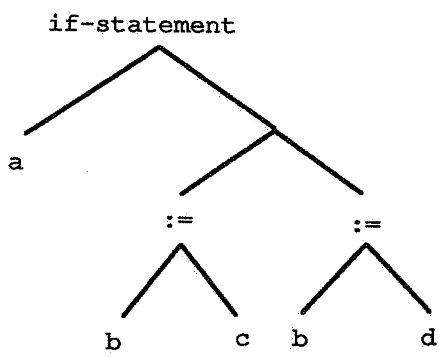
If the syntax of Pascal tells us that

- every sequence of characters is an identifier (`Id`)
- an identifier is also an expression (`Expr`)
- an assignment-statement looks like `Id := Expr`
- an if-statement looks like `if Expr then Stat else Stat`

then for the text

```
if   a   then b:=c else b:=d
```

the following tree is created



This is called an *abstract syntax tree*, it shows the text together with its structure according to the syntax. Now the prettyprinter receives this tree as input. Because it prettyprints only Pascal programs, it has simply coded somewhere how to prettyprint each language structure of Pascal. For this simple piece of text the prettyprinter must have rules encoded for three language structures:

For an if-statement there might be this rule:

> print `if` literally, print a blank, print the `Expr` according to a rule for Expr's, print a blank, print `then` literally, print a blank, print the first statement according to a rule for statements, print a blank, print `else` literally, print a blank and print the second statement according to a rule for statements.

The rule for an assignment might be:

> print the `Id` according to a rule for Id's, print a blank, print := literally, print a blank and print the `Expr` according to a rule for Expr's.

If the rule for Id's say to print Id's as they are, then this is the result:

```
if a then b := c else b := d
```

Here the prettyprinter has only one rule for every language structure. Often it is useful to have another rule for a language structure that will be used in a special situation. For example,what must be done when a long if-statement must be printed on a small page? We would like the prettyprinter to cut the long statement on the right places. The prettyprinter needs an alternative second rule for if-statements.

For example this one:

> if the if-statement does not fit completely on one line, then print the if-statement like this: print `if` literally, print a blank, print the `Expr` according to a rule for Expr's, print a newline, print `then` literally, print a blank, print the first statement according to a rule for statements, print a newline, print `else` literally and print the second statement.

Now a long if-statement will be printed like this

```
if listempty
then number := elements
else number := big
```

Notice that the condition in the second rule ("if it does not fit on one line") is a difficult one. It can not be tested easily beforehand. To avoid much extra work it is best to check the condition of the second rule while prettyprinting according to the first rule. The first rule must now change, because every time a part of the if-statement is printed the prettyprinter must check if it did not reach the end of the line yet, though there is still text to be printed. If the last part of the if-statement is printed without reaching the end of the line, the first rule turned out to be the right rule to use and the if-statement is prettyprinted. If however the end was reached, the prettyprinter must start all over again now following the second rule. Unfortunately, in this case, one line will be prettyprinted two times.

The first rules will now look something like this

> print `if` literally, if end of line start again with rule 2, print a blank, if end of line goto rule 2, print the `Expr` according to a rule for Expr's, if end of line goto rule 2, etc.

This is awkward, but at least the whole if-statement is never prettyprinted two times. Most simple prettyprinters have only one rule for the language structures to avoid these complex rules.

Prettyprinting without any exceptional situations is straightforward, but when the prettyprinter has to deal with the limited dimensions of the screen and with different rules for different situations, prettyprinting turns out to be rather complicated. One important task of the prettyprinter is to decide which rule must be used without doing too much extra work.

1.2 Box-languages

In the previous section we saw that the prettyprint rules were encoded in the prettyprinter. There we described them in long English sentences. It is also possible to describe them formally. Two major advantages are that the meaning is unambiguous and that the rules are easier to read.

Often a *box-language* is used for this purpose. A box-language describes atomic boxes and different operators for combining boxes. Usually an atomic box is a string. Atomic boxes can be combined with an operator to get composite boxes which can be combined again. Each operator stands for a particular way of positioning the boxes relative to each other. In this way it is possible to describe the layout of a language structure in a box-language.

Now we will explain four operators or separators that are common in box-languages. These operators and their arguments allow us to describe all prettyprinting rules that a normal, not too sophisticated, prettyprinter uses. We will give examples of them using the syntax of PPML. The box-language of PPML has exactly these four operators.

- The <h> separator, h of *horizontal*, places the subboxes after each other. It has one argument, the number of blanks between the subboxes. Normally the words in a text are separated by one blank, so the default is one. When the separator does not have an number in it, the default will be used.

  example

  ```
  [<h 4> "we" "are" "placed" "horizontally"]
  ```

  leads to the text

  ```
  we    are    placed    horizontally
  ```

- The <v> separator, v of *vertical*, places each subbox on a new line. The first argument says how many blanks must be indented when a subbox is placed on a new line. So all but the first box have an indentation before them. The second argument does not specify how many newlines, but how many open lines, must be placed between subboxes. So the default is 0.

  example

  ```
  [<v 5,1> "we" "are" "placed" "vertically"]
  ```

leads to the text

```
we
      are
      placed
      vertically
```

- The <hv> separator, *horizontal <u>and</u> vertical*, acts just like a typewriter, it places the subboxes next to each other, in the way <h> does, until the line is full, then it inserts a newline and puts the rest of the subboxes on the next line. <hv> needs three arguments, the first is the number of blanks between subboxes, see separator <h>, the second and third operator have the same purpose as the first and second argument of the <v> separator, they are needed in case the subboxes do not fit on one line.

  example

  ```
  [<hv 1, 2, 0>  "we" "are" "placed" "hor" "and" "vert"]
  ```

  leads to this text on a wide page

  ```
  we are placed hor and vert
  ```

  and to this text on a small page

  ```
  we are placed hor
    and vert
  ```

- The <hov> separator, horizontal <u>or</u> vertical, tries to place the subboxes on one line, like the <h> separator, but if it finds out they do not fit on one line, it will place all subboxes vertically like the <v> operator. <hov> needs the same three arguments as <hv>, namely number of blanks, indent on new line and number of open lines.

  example

  ```
  [<hov 1, 5, 0>  "we" "are" "placed" "hor" "or" "vert"]
  ```

  leads to this text on a wide page

  ```
  we are placed hor or vert
  ```

and to this text on a small page

```
we
are
placed
hor
or
vert
```

Let's go back to the previous section where we described two complicated rules for the short and the long if-statement in English sentences. Both rules can be described in one simple box-term in PPML:

```
[<hov 1,0,0> [<h 1>  "if" *Expr]
             [<h 1>  "then" *Statement1]
             [<h 1>  "else" *Statement2]]
```

`*Expr` is a recursive box-call, it will be replaced by a box according to the format for expression. `*Statement1` and `*Statement2` are also recursive box-calls. The square brackets mark the limits of each non-atomic box. For example here we see that the operator `<hov>` has three subboxes. Try to understand that this box-term in PPML has the same meaning as the two rules for the if-statement in the previous section.

By introducing the box-language two aspects of prettyprinting, that were mixed up in the long English rules, are now separated, describing the possible formats of a language structure and determining in an economic way which format must be chosen. A box-term shows only the possible formats. A box-interpreter, that will form a real page of text out of a box-term, will have fast algorithms for each operator in the box-term. This separation is very useful when developing of a prettyprinter for the generator as we will show in the next section.

1.3 Prettyprinting in the generator.

In the one-language prettyprinter we described in section 1.1 the rules for the prettyprinting of each language structure are encoded in the prettyprinter. However, the generator has no knowledge about any particular language, so it also has no specific prettyprint-rules encoded in it. Because there are so many different languages, it is not even possible to formulate useful standard prettyprinting, that the generator could work with. So the generator can only create a prettyprinter for a language, if it receives the prettyprint-rules for it.

In order to write a specification of the prettyprinting rules, we need a language in which we can describe the format of each language structure. For this purpose we often use a box-language as described in the previous section. A box-language is very suitable, because the specificator of the prettyprint-rules has to define only the global layout for every language structure. He does not have to be concerned with details like what to do when the line is too long or keeping track of the real position each string is getting.

When we give these prettyprint-rules to the generator it can create a environment with a prettyprinter. After the parser has created a syntax tree for the text that must be prettyprinted, the prettyprinter will translate this tree into one box-term. It will recursively apply to every language structure in the tree one suitable prettyprint-rule and thus build a box-term that shows the global layout for the whole text.

Now this box-term still has to be translated into a new piece of text. A box-interpreter inside the generator will take care of this. It translates the box-term into real text. It not only has to deal with the global layout of the text described by the box-term, but also with the real dimensions of the screen. The box-interpreter will have fast algorithms for the operators of the box-language and it can prettyprint strings, the atomic boxes of the box-language. It also handles exceptional situations. When the text can't be printed on the screen according to the global layout, the box-interpreter must print the text slightly different. For example, when a sentence is too long the box-interpreter will put the last part of the sentence on the next line. Notice that the only language the box-interpreter has to understand is the box-language.

Now we see that the box-language is an intermediate language, it separates the language-specific and language-independent aspects of prettyprinting. The specificator of an environment specifies the language-specific part and the language-independent part is implemented in the box-interpreter. The prettyprinter in the generated environment only has to translate the text into a box-term. This is straightforward. The most complicated and time-consuming part of prettyprinting is done by the general box-interpreter.

Let us give an example in simplified Pascal of the whole path.

the text: `repeat a:=b until      true ;`

the input for the parser: the syntax in SDF

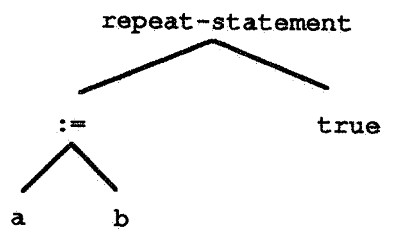
```
lexical syntax
        a                   -> Id          {a, b and true are identifiers}
        b                   -> Id
        true                -> Id
context-free syntax
        Id                  -> Expr  {an identifier is also an expression}
        Id := Expr          -> Statement
        repeat Statement until Expr -> Statement
```

the input for the prettyprinter: prettyprint rules in a box-language

for every language structure, thus for every left-hand side of the rules above, there has to be a rule that says how to prettyprint that structure.

```
Id                  :   string(Id)  {print identifiers as they are}
Id := Expr          :   [<h 1> *Id ":=" *Expr]
repeat Statement
until Expr;         :   [<v 0,0> "repeat"
                                  *Statement
                                  [<h 1> "until" *Expr ";"]
```

In the generated programming environment the parser parses the text using the language definition in SDF and creates this abstract syntax tree for it.



The prettyprinter receives this tree and recursively matches every structure of it with the prettyprint rules formulated in the box-language, trying to make one box-term for the whole text. In this case this would be

```
[<v 0,0> "repeat"
          [<h 1>  string(a)  ":="  string(b)]
          [<h 1>  "until"  string(true)  ";"]]
```

Then the prettyprinter handles this box-term over to the box-interpreter, that will translate the box-term into text and print it on the screen.

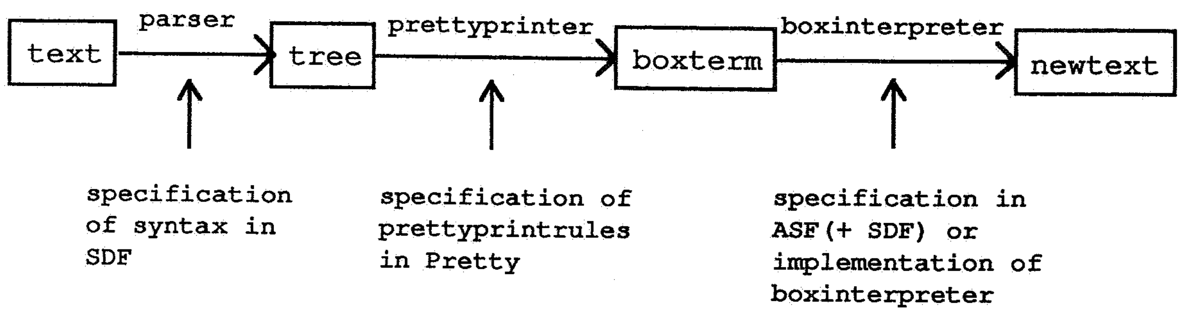
```
repeat
a := b
until true;
```

If the page width is 7, the last line is too long, so the prettyprinter would print the box-term differently using his own way to deal with exceptional situations. One possibility could be:

```
repeat
a := b
until
true;
```

A good box-interpreter handles exceptional cases as much the same as normal cases to assure that the new text does not look completely unexpected. Here the conversion of box-term into text was rather simple, but when the box-language offers more complicated operators the task of the box-interpreter becomes also more difficult and time-consuming.

It is important throughout the next chapters to know which steps there are and when they are taken. We will summarize the steps in a picture also showing where which language is used.



1.4 PPML

We have already explained the operators and the arguments of the box-language of PPML in section 1.2. In this section we will first give some idea of what a prettyprint-specification in PPML looks like, then we will explain some features of PPML by showing some rules from the prettyprint-specification for the toy-language PICO. Finally we will give an evaluation of PPML.

1.4.1 Rules in PPML

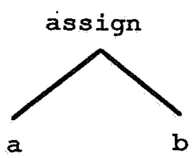
The box-language which we already have explained is only a part of the prettyprint language PPML. It is also possible to describe patterns for each language structure, simple auxiliary functions and control structures. A prettyprint-specification in PPML is a sequence of rules, that look like this

```
<pattern>   ->   <format>;
```

A pattern expresses an abstract syntax tree that may contain variables. A format is usually written in the box-language. Here is an example of a rule.

```
assign(*var, *exp)   ->  [<h 1>  *var ":=" *exp];
```

*var and *exp are variables. The pattern matches a binary syntax with the assign operator on top. Remember that a parser first creates an abstract syntax tree of the text, that must be prettyprinted. Then the prettyprinter will try to match this tree to a pattern of one of the rules. It will continue to match subtrees, that are instantiated to the variables, with the patterns until all subtrees are matched. Then the final box-term for the tree is formed. The prettyprinter will, for example, match the following tree with the rule above.



After recursive calls to rules for identifiers the final box-expression will be

```
[<h 1>  "a" := "b"].
```

Now follows an example of a complete prettyprint-specification. Note that it is also possible to define constants and default arguments.

```
prettyprinter of FILLINNAME is
constant
   tab = 3;
default
   <h 1>; <v 0, 0>; <hv 1, 0, 0>; <hov 1, 0, 0>;
rules
   assign(*var, *exp)         -> [<h>  *var ":=" *exp];
   ifthenelse(*exp, *series1,
              *series2)       -> [<hov> [<hv> "if" *expr ]
                                        [<hv> "then" *series1]
                                        [<hv> "else" *series2]
                                              "fi" ];
end prettyprinter
```

We already explained the meaning of the separators and their arguments in section 1.2. However one thing is still not clear. What does the <h> separator do with the rest of its subboxes when the line is full? Does it give an error message of does it place the rest of the subboxes on the next line just like the <hv> separator? If the latter is true, then what is the difference between the <h> and the <hv> operator? Maybe we can do away with the <h> separator and use the <hv> operator instead. We didn't t find an answer in the documentation, but of course we could have tested this with the current PPML-implementation. Unfortunately, we didn't have time to do so. So just for safety reasons we will use the <hv> separator whenever we think a sentence may become too long.

Now we will describe a larger example, a complete prettyprint specification for a small language. Maybe then we can say more about the expressiveness of the box-language of PPML and about the readability of specifications in PPML.

1.4.2 A prettyprinter for PICO in PPML

PICO is a small Pascal-like toy language. A program starts with the keyword "begin", followed by a declaration section, where only variables of type natural number or string can be declared. Then follows the statement-section, in which only the assignment-statement, the if-statement and the while-statement can be used. The expressions used in these statements may be an identifier, an addition or subtraction of natural numbers or a concatenation of strings. A PICO-program concludes with the keyword "end". In appendix A a specification of PICO in SDF is given. It is a simple SDF-specification, that will help to understand the syntax of PICO better.

Appendix B is a specification of prettyprint-rules for PICO in PPML. We will not discuss the whole specification. We will only explain the rules where a yet unknown feature of PPML is used. This will lead to a better understanding of the specification and of PPML.

Three unknown features are local separators, list patterns and elision (a mechanism to see a structure with different levels of detail). We will now give an example of each feature.

The second rule in the specification is

```
pico_program(*decls, *series) ->
                                   [<v> "begin"
                                        <v tab,0> [<v 0,1> *decls
                                                          *series]
                                        "end"];
```

This rule shows the use of local separators. `<v tab,0>` is a local separator, it specifies how its left and right neighbour subbox should be combined and overrules the global separator <v> at the beginning of the box. The rule says that a PICO-program must look like this.

```
begin
    *decls
    *series
end
```

We also see here that if another value than the default-value for one argument must be used, the default-values for all other arguments of that operator must be repeated. Also note that the use of constants like tab make it easier to remember where each argument stands for.

The third rule specifies how the declaration section, which consists of the keyword `declare` followed by a list of declarations, must be prettyprinted.

```
decls(*decl, **decls) ->
             [<v tab,0> "declare"
                        [<hv> *decl (<h 0> "," **decls) ";"]];
```

The pattern of this rule is a list pattern. `decls(*decl, **decls)` matches a tree with on top the `decls`-operator and with at least one son. The first son, i.e. the first decl, is instantiated with `*decl` and the rest of the sons with `**decls`. The parentheses in the format of the rule indicate that the pattern must be repeated for the rest of the list, if the list has more than one element. A comma must be placed before each element of the list and immediately after the previous element. So the list of decls is printed with a comma inserted between each two elements, thus like this

```
declare
    decl, decl, ..., decl;
```

The last feature we will discuss is *elision*, also called outlining. This allows to print an overview of the program in which, for example, only the procedure headings of the procedures are shown instead of the complete text. This is done by controlling the depth of recursive calls during the pattern-matching. If not

not all variables in a box-term are substituted by a subbox, but if they are simply replaced by 3 dots, some details will not be seen any more. The depth of a recursive call is also called the holophraste level. Automatically for each deeper recursive call the holophraste level is decreased by one.

example

```
op1(*x, op2(*y))  ->    [<h>  *x *y];
```

if h is the current holophraste value, `*x` will be prettyprinted with a holophraste value h - 1 and `*y` with holophraste value h - 2. So the pattern of the rule determines what holophraste level the subtrees will have.

The user can make an outline of his program by specifying

```
program(*x)        ->    *x ! 10;
*x ! 0             ->    "...";
```

The abstract syntax tree of the complete program will now be prettyprinted with holophraste level 10 and a subtree with holophraste level 0 will be printed as ... literally.

It is also possible to specify what details the user does want to see. The rule for the plus-expression in the PICO-specification says, for example, that if a part of a plus-expression is printed, then the complete plus-expression must be shown, because the holophraste values of the recursive calls stay the same as the holophraste value in the beginning.

```
plus(*expr1, *expr2)    ->  [<hv> *expr1 !+1  + *expr2 !+1]
```

1.4.3 Evaluation of PPML

Here we will focus on two properties of prettyprint languages, expressiveness and ease of use.

What does expressiveness mean? In a more expressive language the user can specify more complex actions, he has more possibilities, or he can describe the same actions as in less expressive languages, but in an easier way. In an ideal prettyprint language the specificator can express all his wishes for all languages.

We think the possibilities offered in PPML are enough to describe a normal, not too sophisticated prettyprinter. PPML offers simple patterns, list patterns and even conditional patterns based on the result value of a simple function. For differentiating between alternative layouts there are four structures that can be used, context-dependent rules, if-statements, repeat-statements and simple functions in the format of a rule. The box-language offers enough possibilities for describing the prettyprint formats of a normal prettyprinter. Also when we wrote a specification for PICO, we did not really clash with any restraints on these matters. We only came across one layout format we would like to be able to specify.

It is not possible to specify that a short if-statement must be printed like this

```
if a then b else c
```

and a long if-statement like this

```
if listpointer = nil then
  writeln("nothing left")
else
  writeln("still something left")
```

This is because of the tabs in the long if-statement. The following format seems right, but isn't.

```
[<hov 1,tab,0>
   [<hov>
       [<hov 1,tab,0> [<hv> "if" *Expr "then"]
                      *Stat1]
                      "else"]
                      *Stat2]
```

According to this rule, this if-statement is also possible.

```
if listpointer = nil then nil else
    bufferpointer
```

In the box-language of PPML complicated things, like alignment and columns can not be specified. Is it not possible to put more separators in the box-language that can handle these things? Maybe it is, but we must take something else into account.

Depending on its use, the response time of the PPML-prettyprinter is very important. For instance, when only the tree-representation of a program is maintained during editing and the prettyprinter is used for recreating the textual representation of the program. This means that every time the user makes a change, it is translated in a change in the tree and the whole tree must be prettyprinted again to show the changed text. So even for a small change the whole text must be prettyprinted. In this case the prettyprinter must of course give quick responses. This constraint on response-time can have influence on the box-language of PPML. Complex operators in the box-language will require a complicated and thus most likely a slower box-interpreter. As we already mentioned most of the work during prettyprinting is done by the box-interpreter, so a fast prettyprinter needs a fast box-interpreter. Maybe it is not possible to find a fast interpretation for the complex operators. Then the box-language must be kept simple.

Let's look at another aspect of prettyprint languages, readability. It is a nice property of a prettyprint-language, when the specifications written in this language are easy to read and to write. Probably the language will then be easier to learn and errors will be less likely to occur. One way to improve the readability of prettyprint specifications is to give the operators and arguments in the box-language clear names or symbols, so it is obvious what they mean. Having many operators and arguments can be confusing, but when they have clear names, this does not have to be so.

We find subjectively specifications in PPML not very easy to read and to write. It takes some time before the user of PPML really knows what action belongs to each operator name and before he can read the operators quickly. The arguments are just plain numbers, so he will have to look first at what position the number is placed, before he knows what the argument stands for. It is rather clumsy that the default-values of all other arguments must be repeated, when one argument of a separator is changed. A format of a language structure with both indentation and exdentation, like in a long if-statement (use `<hv>` instead of `<hov>` in the rule above), has got a lot of separators and brackets. Using local separators will reduce the number of brackets, but not the number of separators.

In summary, we can say that in PPML the specificator can express all the things he wants for a normal prettyprinter. Only the readability of the prettyprint-specification can still be improved.

Chapter Two: Pretty, a new box-language

In this chapter we will introduce Pretty, a box-language. We will first explain the syntax of Pretty, then we will give a prettyprint-specification for PICO in Pretty and say something about the prettyprint-specification for Pascal. Finally we will evaluate Pretty.

2.1 A definition of Pretty

2.1.1 the syntax of Pretty

Contrarily to the prettyprint-language PPML, which has besides a box-language also some other features, Pretty is only a box-language. For specifying the other features we will use ASF+SDF.

This is an example of a format written in Pretty and in PPML.

```
[ "case" *Exp;                  [<v> [<hv> "case" *Exp]

   -> *Caselist ;                        <v tab, 0> *Caselist

  "end" ]                               "end"]
```

In Pretty all operators are written in infix notation, so an operator says how the two boxes on each side of it will be combined together. There are three operators. The *nothing* operator places the boxes next to each other, just like the <hv> operator in PPML. It is called like this because nothing will be placed between the boxes. The /-operator acts just like the <hov> operator, only now this infix operator is associative, this means that a sequence of boxes combined with /, can be seen as if all these boxes were combined with one /-operator. So the following formats have the same meaning.

```
[ "if" *Expr /                  [ <hov> [<hv> "if" *Expr]

  "then" *Stat1 /                       [<hv> "then" *Stat1]

  "else" *Stat2 ]                       [<hv> "else" *Stat2]]
```

The ;-operator acts just like the <v> operator in PPML.

We have chosen to use the operators in this form, because we think it will improve the readability of the formats. The symbols for the operators are short and clear. Because we use the infix notation, there is always an operator between each two subboxes and the reader of the specification can see quickly how subboxes are combined with each other. Furthermore, he can now simply think of / as of a possible cut in the sentence and of ; as a newline. It is easier to think in these terms than to think for example about the complicated action of the hov operator.

We introduced four new arguments. Now we can give an absolute tab, so the next box will start on the column number of the value of the absolute tab. We also have three new arguments to change the values of the left margin and the right margin.

An argument is written like a symbol, that indicates which argument is meant, followed by a value for that argument. The arguments have no strict ordering. The specificator just gives a list of arguments, in any ordering he likes, containing only the arguments for which he does not want to use the default value. This argument list will be placed logically with the boxes. This is different in the box-language of PPML, where most operators are placed with the separators.

The new box-language now looks very simple. A string is an atomic box. Compound boxes can have the following forms, a list of arguments followed by a box or two boxes combined with nothing, `;` or `/`.To reduce the number of brackets we specified priorities on the operators. Except for the symbols that are used for the arguments, these three sentences describe the whole language!

Here is the SDF-definition for the language. The arguments are now called changes, because it is better to see them as temporarily changing the default-values of the prettyprinter, when the box, that follows the changes, must be printed.

```
module Box-syntax
   import Strings
   export
      sorts SYMBOL CHANGE CHANGELIST BOX
      lexical syntax
         [ \t\n\r]             -> LAYOUT
         "<->"                 -> SYMBOL
         "|"                   -> SYMBOL
         "->"                  -> SYMBOL
         "->>"                 -> SYMBOL
         "|<<-"                -> SYMBOL
         "->>|"                -> SYMBOL
         "|<+"                 -> SYMBOL
         "|d|"                 -> SYMBOL
         "d+"                  -> SYMBOL

      context-free syntax
         STRING                -> BOX
         SYMBOL INT            -> CHANGE
         SYMBOL                -> CHANGE
         {CHANGE ","}+         -> CHANGELIST
         "(" CHANGELIST ")"    -> CHANGELIST  {bracket}
         CHANGELIST BOX        -> BOX
         BOX  BOX              -> BOX  {right}
         BOX "/" BOX           -> BOX  {right}
         BOX ";" BOX           -> BOX  {right}
         "[" BOX "]"           -> BOX
         "(" BOX ")"           -> BOX  {bracket}
         defaults              -> CHANGELIST
         layoutchar            -> STRING
```

```
priorities

     CHANGELIST BOX      -> BOX  >
     BOX  BOX            -> BOX  >
     BOX "/" BOX         -> BOX  >
     BOX ";" BOX         -> BOX

equations

[1]  defaults    = <-> 1,   | 1,   -> 4,   ->> 8,   |<- 1,
                   ->>| 60,   |<-+ 30,   |d| 100,   d+ 1

[2]  layoutchar = " "
```

The extension `{right}` means that the operators are right-associative. When brackets must be placed in a box-term, where all boxes are combined by the same operator (op), the box-term would look like this.

```
Box1 op (Box2 op (Box3 op ....)))
```

The extension `{bracket}` means that the brackets in this function are not really needed in the syntax, they will only keep parts of the language together.

As we see in the second and third rule of the context free syntax an argument can be a symbol or a symbol followed by an integer. When there is no integer behind the symbol, the value for this symbol in the defaults-equation will be used, just like in PPML. The layoutchar specifies what string must be placed on the open space between the boxes. Usually, a blank is placed between two boxes, but now it is also possible to place for example dots on open spaces. Normally the equations for the defaults and the layoutchar will be placed in the prettyprint-specification.

All arguments are now placed before a box, even the ones that are associated with operators in PPML. So if the specificator wants a tab before a box or two blanks instead of one blank, he must write this down in the changelist of the box. Let's see what changes can be given in a changelist.

`<-> Int`	width	Int blanks between the end of the previous box and the start of the following box.
`| Int`	height	Int newlines between the end of the previous box and the start of the following box.
`-> Int`	relative tab	Int blanks between the end of the previous box and the start of the following box.
`->> Int`	absolute tab	the following box must start at column Int. The open space before it must be filled with blanks.
`|<-+ Int`	relative left margin	the left margin for the following box must be set to the old left margin + Int
`|<<- Int`	absolute left margin	the left margin for the following box must be set to Int
`->>| Int`	absolute right margin	the right margin for the following box must be set to Int
`|d| Int`	absolute depth	the depth, the holophraste level, of the following box must be set to Int
`d+ Int`	relative depth	the depth of the following box must be set to the old depth + Int

The four new symbols are `->>`, `|<-+`, `|<<-`, `->>|`. With these operators the right and left margin can be changed and together with the absolute tab it is now possible to make columns.

2.1.1 combining operators and arguments

Here we want to draw attention to four properties of Pretty with respect to combining operators and arguments.

When a relative tab is used in combination with the operator `/`, the tab will only be printed when the boxes do not fit on one line and are thus placed above each other. This is logical, as shown in this example

```
[ "if" *Expr "then /
   -> *Stat1 /
   "else /
   -> *Stat2 ]
```

Now in a small if-statement the tabs will not be printed:

```
if a then b else c
```

but in a long if-statement they will appear:

```
if listpointer = nil then
  writeln("nothing left")
else
  writeln("still something left")
```

Here follow two other small aspects we have to keep in mind. Because the arguments are now placed before a box, the operators must retrieve their arguments from the box next to the operator.

example

When the box-interpreter must prettyprint the box-term `"next" "to"`, containing two boxes combined by the nothing-operator, it will see that the box-term `"to"` has no list of arguments, so it will look in the defaults-equation for the default number of blanks between two boxes. When it prettyprints `"further" <-> 5 "apart"`, it will set the number or blanks to 5.

We can also see from the definition of Pretty, that it is possible to combine all arguments with all operators. Some combinations are very common, like the nothing operator and <->, we saw in the example above, and the operator `;` and `|`, but combinations that were not possible in PPML can now also be made, like this one

```
"funny" "but" |2 "possible"
```

The box-interpreter has to be able to do something sensible with all box-terms. This term might be printed like

```
funny but


           possible
```

In chapter three we will explain the working of the box-interpreter in detail.

Finally, we want to draw attention to two situations that cause us trouble in Pretty. We could not implement the arguments of the `<hv>`-operator in Pretty the same way as in PPML, because in PPML the arguments are associated with the separator and in Pretty they are associated with the boxes.

The `<hv>`-operator has three arguments in PPML. It is simple to write down the first argument of `<hv>` in Pretty, the one that says how many layout characters must be placed between the boxes. We just write <-> Int between every two boxes. However the second argument of `<hv>`, the tab, is not so easily implemented in Pretty. The tab must only be placed before the box that is placed first on a new line. But we can not know beforehand which box this will be, so in Pretty we can't place the tab before the right box. We must find another way to get the same effect. We now group the boxes with normal brackets and place `|<-+ tab` before this composite box. This means that the left margin of this composite box will become one tab greater, but the box will still start on the column of the old left margin. In the end the text will look the same as if a tab was placed on the new line. So when this is written in PPML

```
[<hv 1, tab, 0> *Expr1 "+" *Expr2 ]
```

this is the format in Pretty

```
[|<-+ tab (*Expr1 "+" *Expr2) ]
```

However we did not find a solution for the other situation. The third argument of `<hv>` says how many open lines there must be placed before the first box that does not fit on the line. In Pretty it is not possible to write this down. Again, it is not possible to determine before which box the newlines must be placed.

We don't think it will be a real drawback of Pretty. The trick with the tab-options doesn't look very nice, but it is possible to describe the tab-option and it will not be used very often. The newline-option can't be described in Pretty, but we think it will hardly ever be needed in a prettyprint-specification.

2.1.2 priorities

In Pretty we specified priorities on the operators, so the specificator does not have to place brackets around every combination of boxes with an operator. The boxes will now be grouped according to the priorities. Putting a changelist before a box has a higher priority than using the nothing operator between two boxes. The nothing operator has a higher priority than the / operator, which on its turn has a higher priority than the ; operator. The three operators are right-associative. These priorities are just as expected. Here is an example.

According to the priorities, the prettyprinter would see this example

```
["if" *Expr1 "then" ;
  -> *Expr2 ;
 "else" / *Expr3 ]
```

like this

```
[("if" (*Expr1"then")) ;
 ((-> *Expr2) ;
 ("else" / *Expr3)) ]
```

Now the brackets are placed just as was intended with the if-format.

2.1.3 rules

When a specificator writes a prettyprint specification in Pretty, he must not only use Pretty but also ASF + SDF. What does a specification look like? How must one write down the prettyprint rules?

We can write the prettyprint rules down as equations in the equation-section of a ASF + SDF module. In the context-free syntax of the module we will declare the main function, that will translate the language structures of the language into a box-term. In the example below, this function has the name *. In the equations the specificator will describe how the function exactly works. He will have to describe what box-term the *-function will return for every language structure. The form of the equations is very much the same as the form of the rules in PPML. The equations are written in the simple syntax of ASF, a tag, a term constructed from the functions and the variables defined earlier, a "=" sign and another term constructed from those functions and variables. An equation can also have conditions. The equation will then only be used for a particular input when the conditions are satisfied.

Here is a part of an example module

```
module Language-prettyspec
  import  Language-syntax, Box-syntax
  export
    context-free syntax
      "*" STAREXPR        -> BOX            {the main function}
       PROGRAM            -> STAREXPR       {every language structure
       STATEMENT          -> STAREXPR        is of type
       ...                                   STAREXPR}
    variables
       Exp                -> EXPR
       ...

    equations

    [11] * if Exp then Series1
           else Series2 fi       =  [ "if" *Exp ;
                                      "then" *Series1 ;
                                      "else" *Series2 ;
                                      "fi" ]
    [12] * while Exp do Series do =  [ "while" *Exp ;
                                      "do" *Series ;
                                      "od" ]
```

We have given some idea of what a prettyprint-specification in Pretty plus ASF+SDF looks like. More information on ASF+SDF can be found in the references [1], [2].

2.1.4 elision

In section 1.4.3 we saw that in PPML elision is handled during the translation of the text into a box-term. When the text is matched with the left-hand sides of the prettyprint-rules, the pattern-matcher refers to the information about the holophraste level of each subtree. Only when a subtree has a holophraste level of 0, it will match the subtree with this rule

```
x !0                  ->    "..."
```

Instead of a whole box-term for this subtree, only this small box, a string of three dots, is returned. So when elision is used to create an outline of the text, the text is translated into a box-term that is smaller than the box-term for the whole text. This smaller box-term is then translated into a new text by the box-interpreter.

In Pretty it is not possible to handle elision during the translation of the text into a box-term, because the pattern-matching will be done in the standard way for all equations written in ASF + SDF. We don't have access to the information about the level of each subtree. Therefore, when an outline has to be made of the text, we will translate the text into a box-term, that holds information about the holophraste level of each language structure, and the box-interpreter will create an outline of this box-term during the translation of the box-term into the new text.

How do we put information about the holophraste levels into the box-term? Well, we simply group all boxes that must have the same holophraste level together by placing square brackets around them. When the box-interpreter recursively translates all subboxes into strings, it simply decrements the holophraste level of each subbox surrounded by square brackets.

In PPML one would write these two rules in the prettyprint-specification to get an outline of the text onto a depth of 10.

```
program(*x)          ->     *x !10
x ! 0                ->     "..."
```

In Pretty the specificator can set the value of the symbol |d| (absolute depth) to the holophraste level he wants for the whole tree. This is how he specifies the information of the first PPML-rule above. The second rule, which says what should be printed when a box has holophraste level 0, can be specified by associating a string with depthstring, a predefined constant. Here are two equations that hold the same information for the elision as the two PPML-rules above.

```
[1] defaults     =  <-> 1,   | 1,   -> 4,   ->> 40,   |<<- 1,   ->>| 80,
                    |<-+  5,   |d| 10,   d+ 1
[2] depthstring  =  "..."
```

Now the box-interpreter can handle elision in a simple way. It will set the holophraste level for the whole box-term to the default-value of |d| in the defaults-equation. Then it will recursively translate the subboxes into real text. Every time it comes across a box surrounded by square brackets, it checks if the holophraste level of this box is zero. If so, it returns the depthstring as the real text for this box. If the level is greater than zero, it will go on translating the subboxes, but with a holophraste level that is set one lower.

The final text will be the same as the text the PPML-prettyprinter will create. Notice however, that the PPML-prettyprinter does less work, when it creates an outline, because is immediately translates the text into a small box-term, whereas in Pretty first the box-term for the whole text is constructed and only later an outline of this box-term is make.

Let us now look at what the specificator must do to assure that every subbox gets the correct holophraste level. Because usually every format begins and ends with a square bracket, most of the time the specificator does not have to be aware of the holophraste levels, but just places square brackets around each format. However, there is one situation where the brackets must be placed differently. If a list is printed, each element must have the same holophraste level. In this case once a list is printed all its elements are printed and not just a few of them. Usually one would like this to happen. If the specificator always puts brackets at the begin and end of a format, this will not work correctly.

Here are two rules in Pretty

```
*begin Series end       = [ "begin" ;
                             -> *Series ;
                            "end" ]

*Stat ; Stats           = [ *Stat ";" ;
                            *Stats ]
```

When Series is defined as a list of Statements, the following box-term can be created:

```
[ "begin" ;
   -> [ [Stat1] ";" ; [ [Stat2] ";" ; [...]]] ;
  "end"]
```

Here we see that each next element in the statement list will be printed with a holophraste level that is one smaller than the holophraste level of the box before it. Now when the whole list has a low holophraste level the last elements in the list will be left out. One way to solve this is to change the two rules into

```
*begin Series end       = [ "begin" ;
                            -> [*Series] ;
                            "end" ]

*Stat ; Stats           =   *Stat ";" ;
                            *Stats
```

We moved the brackets in the second rule up one level to the first rule. Now the box-term will look like this

```
[ "begin" ;
   -> [ [Stat1] ";"  [Stat2] ";" .....] ;
  "end"]
```

Here all elements of the list have the same holophraste level.

2.2 PICO in Pretty

We will now show what a real prettyprint-specification in ASF+SDF plus Pretty looks like by presenting a prettyprint-specification for PICO. The equations will express the same rules we described for PICO in PPML. Next we will give an extension of these rules. These extra rules will make it possible to do some alignment. We will also say something about alignment in Pascal. Finally, we will give a comparison of the specification for PICO in PPML and the one in Pretty.

2.2.1 simple Pretty-specification of PICO

In the specification below the context-free syntax and the variable-section are abbreviated. Appendix C contains the complete text for the extended module. More information about ASF + SDF can be found in the references.

```
module Pico-box
  import  Pico-syntax, Box-syntax
  export
    context-free syntax
      "*" STAREXPR              -> BOX            {the main function}
      PROGRAM                   -> STAREXPR       {every language structure
      DECLS                     -> STAREXPR        of PICO is a
      ...                                          starexpression}

      ID-LIST "^" INT           -> STAREXPR       {language structures
      ...                                          with an argument}
    variables
      Exp Exp1 Exp2             -> EXP
      Stat                      -> STATEMENT
      ...
```

equations

```
[1]   defaults                  =  <-> 1,   | 1,   -> 4,   ->> 40,   |<<- 1,
                                   ->>| 80,   |<-+ 5,  |d| 100,   d+ 1
[2]   layoutchar                =  " "
[3]   depthstring               =  "..."
[4]   * begin Decls Series end  =  [ "begin" ;
                                      -> *Decls ; |2,
                                      -> [*Series] ;
                                     "end" ]
[5]   * declare ;               =  [ "declare" ";" ]
[6]   * declare Id-list ;       =  [ "declare" ;
                                     -> [*Id-list] ";" ]
[7]   * Id : Type, Id-list      =   *Id <->0 ":" *Type "," ;
                                    *Id-list
[8]   * Id : Type               =   *Id <->0 ":" *Type

[9]   * Stat; Stats             =    *Stat <->0 ";" ;
                                     *Stats
[10]  *                         =              {for the empty Stats-list}
[11]  * Id := Exp               =  [ *Id <->0 ":=" d+ *Exp ]
[12]  * if Exp then Series1
        else Series2 fi         =  [ "if" *Exp "then ;
                                     -> *Series1 ;
                                     "else" ;
                                     -> *Series2 ;
                                     "fi" ]
[13]  * while Exp do Series do  =  [ "while" *Exp "do" ;
                                     -> *Series ;
                                     "od" ]
[14]  * Exp1 + Exp2             =  [ d+ *Exp1 "+" d+ *Exp2 ]
[15]  * Exp1 - Exp2             =  [ d+ *Exp1 "-" d+ *Exp2 ]
[16]  * Exp1 ||  Exp2)          =  [ d+ *Exp1 "||"d+ *Exp2 ]
[17]  * id(Chars)               =  [ string(Chars) ]
[18]  * nat(Chars)              =  [ string(Chars) ]
[19]  * string(Chars)           =  [ string(Chars) ]
```

Here we see that the SDF-syntax definition of PICO and that of the box-language Pretty are imported. In the context-free syntax part all language structures of PICO are defined to be of the same type STAREXPR. We also defined the * - function, the main prettyprint function, it will translate all STAREXPR's, so all language structures, in a box-term. The rules in the equation section describe how this function does its job.

The defaults-equation has two functions. The start values for the prettyprinter are declared, |<<- and ->>| determine the page width and |d| determines the normal depth to which a tree must be printed, and also the default-arguments for the operators are declared, <->, |, ->, ->>, |<-+ and d+.

The layoutchar specifies what string the hv and hov operators must place between the separate boxes. Usually a blank is placed between each two boxes, but it is also possible to specify for example that two dots must be placed in between, by writing down layoutchar = "..".

In equation number [11], [14], [15] and [16] we used the d+ change, because we would like an expression to be printed completely, once a part of it has been printed.

In rules [17] to [19] a special feature of ASF+SDF is used. If all the characters of a specific identifier must be collected in order to create a string of them, the predefined sort CHAR and the following functions can be used.

```
id "(" CHAR ")"          ->    ID

string "(" CHAR ")"      ->    STRING
```

These functions are automatically created for each lexical function that groups characters into a new type. The term id(Chars) matches now with all identifiers and the characters in the identifier are automatically placed in the list Chars. Now by the term string(Chars) this list of characters is converted to a normal string. See the references for more details.

2.2.2 an extension of the simple specification

We would like to align the identifier-list in PICO. Instead of this text

```
number: natural,

amountoffish: natural,

big: natural
```

we would like the prettyprinter to produce this text

```
number       : natural,

amountoffish : natural,

big          : natural
```

In order to do this we must declare two auxiliary functions in the context-free syntax. One that computes the length of an identifier, another that computes the maximum length of the identifiers in an Id-list. When the *-function must now translate the language structures ID-LIST and ID-TYPE, it must have an extra argument, namely the length of the longest identifier plus 1. This is exactly the column number where the colon will be placed. So the two STAREXPR's for ID-LIST and ID-TYPE must be expanded. Finally the rules [6], [7] and[8] must be changed, because they must pass or use the argument. Here are the changes.

```
context-free syntax
   length "(" IDCHARS ")"  -> INT
   maxlength of ID-LIST    -> INT
   ID-LIST "^" INT         -> STAREXPR
   ID-TYPE "^" INT         -> STAREXPR
variables
     Maxl  Spaces          -> INT
equations
[6] * declare Id-list ;          = ["declare" ;
                                    -> [*Id-list ^
                                   (maxlength of Id-list + 1)] ";" ]
[7] * (Id : Type, Id-list
                  ^ Maxl)        = *Id ->> Maxl ":" *Type "," ;
                                    *(Id-list  ^ Maxl)
[8] *(Id : Type ^ Maxl)          = *Id ->> Maxl ":" *Type
```

Here we see that when the Id-list is going to be printed, the maximum length of the Id's plus 1 is computed and passed on. By using the absolute tab we make sure that the colon is placed on the value of the absolute tab and all places between an identifier and a colon are filled with blanks.

Appendix C contains the complete extended specification for PICO.

2.3 Pascal in Pretty

We wanted to write a specification for a serious programming language to find out if a specificator could specify all he needed in Pretty. So we made a prettyprint-specification for Pascal in Pretty. This is placed in appendix E. The Pascal-specification looks a lot like the PICO-specification, so we won't go over it in details. We will only mention the things we especially noticed.

Just as in PICO all language structures must be defined to be of type STAREXPR and for every language structure a variable must be defined of the same type as this language structure. Because Pascal has a lot more language structures than PICO, this is rather awkward. Now two pages are filled with these definitions. Maybe in the future these functions and variables can be derived directly from the syntax definition in SDF of a language.

In the Pascal-specification we did something special. We defined a new change, the _ change. In the equations we wrote down that this change is equal to the change <->0. In this way we defined an abbreviation for the rather long and disturbing change <->0. The change <->0 is used very often in Pascal, because usually symbols like , and ; and brackets start right after the last box. We think that the abbreviation for this change makes the rules easier to read and to write, but every specificator can decide for himself if he wants to use such abbreviations or not. Of course it is not possible to define a real new change in a specification, because the box-interpreter wouldn't know how to handle it.

In Pascal there are a lot of places where alignment can be used. Here we will show a difficult case of alignment, namely the alignment of the variable declarations in Pascal as we specified it in appendix E. Look at this text.

```
var
   Number, Int1, Int2, Length,Value, Idnum, Buffnum, Height, Width,
   Depth, Wheelnumber, Carnumber : integer;
   Name : array[1..Namelength] of char;
   Character, Char1 : char;
```

We would like to change this text into the far more readable text

```
var
   Number, Int1, Int2, Length, Value,
   Idnum, Buffnum, Height, Width,
   Depth, Wheelnumber, Carnumber      : integer;
   Name                               : array[1..Namelength] of char;
   Character, Char1                   : char;
```

In the Pascal-specification we do this by carefully determining where the : must be placed. We also set the right margin for the list of integer variables just before the column of the :. Now the nothing operator will automatically place the integers on three lines between the left margin and the changed right margin.

How do we determine the right place for the colon ? First we calculate the length of the longest list of identifiers in the whole variable section. Then we calculate the length of the longest type name. If the addition of both lengths is smaller than the page width, both longest parts will fit on one line, so the colon can be placed right after the longest identifierlist. If the addition is greater than the page width and the longest type name is not very long, the type name will be placed at the end of the line and the colon will be placed right before this type name. Now the identifier-lists will be spread over more lines. If the longest type name is very long the colon will be placed in the middle of the line and both the identifierlists and the type names will be spread over more than one line.

Hopefully, the following equations for the variable declarations will explain the actions for the alignment better.

```
[9'1]    *(var Var-Decls;)          = ["var" [*(Var-Decls
                                              ^ maxids of Var-Decls
                                              ^ maxtype of Var-Decls)] ";"]

[10'1] *(Var1-Decl; Var-Decls ^MaxIds ^MaxT) =
                                       *(Var1-Decl ^ MaxIds ^ MaxT) ";" ;
                                       *(Var-Decls ^ MaxIds ^ MaxT)

[11'1] MaxIds + MaxT < Line
       --------------------
       *(Ids : Type ^ MaxIds ^ MaxT) = [*(Ids) ->>MaxIds ":" *(Type)]
```

```
[11'2] MaxIds + MaxT > Line, MaxT <= 40,
       Tab = Line - MaxT - 3
       --------------------------------
       *(Ids : Type ^ MaxIds ^ MaxT) =
                                    [->>| Tab *(Ids) ->>Tab ":" *(Type)]

[11'3] MaxIds + MaxT > Line, MaxT > 40,
       Halfline3 = Halfline + 3
       --------------------------------
       *(Ids : Type ^ MaxIds ^ MaxT) =
           [->>| Halfline *(Ids) ->> Halfline "=" |<<- Halfline3 *(Type) ]
```

Unfortunately, we see here that this mixture of computing things and specifying the global layout for a language structure makes the equations less easier to read. The formats in the box-language are still not very difficult, but the arguments in the starexpressions and the calculations in the conditions of the equations make the equations less clear. We would rather like to have a special operator for alignment, so we can say for example "align this list on the ":" character". However it is very difficult to solve alignment in general.

When writing the specification for Pascal we came across a slight problem. We can not specify, that we don't want a new line to start with a comma. If the comma doesn't fit on the previous line, we would rather have the string before the comma also on the next line. The text

```
var
   Number, Length, Value, Idnum, Int1
   , Int2, Wheelnumber                : integer;
```

is clearly not right. We rather want the text:

```
var
   Number, Length, Value, Idnum,
   Int1, Int2, Wheelnumber            : integer;
```

One solution to this problem is to simply implement in the box-interpreter that it never must place punctuation marks at the beginning of a new line. Another solution is to add a new operator to the box-language that would allow the specificator to say "the last string of this box must be on the same line as the first string in the next box, or easily said, no newline between these boxes. Neither PPML nor Pretty can handle this situation.

We would have liked to test the prettyprint-specification for Pascal and prettyprint real Pascal programs with it. Then we could see if our prettyprint-specification is correct and whether there are things we wanted to specify, but we could not express in Pretty. If the new version of the generator would have been finished, we would just put in the syntax definition of Pascal, the prettyprint-specification for Pascal and the specification in ASF + SDF of the box-interpreter and we would have a programming environment for Pascal with a prettyprinter. Unfortunately, the new version is not finished yet.

So far we can only say that we were able to specify all we wanted in Pretty to describe a reasonable prettyprinter for Pascal.

2.4 Evaluation of Pretty

Just as with the evaluation of PPML, we will focus on the readability of the specifications written in Pretty and on the expressiveness of Pretty .

It is not very difficult to learn to read or to write Pretty. All operators and arguments are written down by logical small symbols. Because all operators are written in infix notation, the same layout as the rule itself describes can now be placed on the subboxes. So at first sight, the reader can see by the indentation in the rule, what the rule means. This really improves the readability.

Pretty is based on PPML. Except for the new arguments that were introduced, Pretty offers more or less the same expressiveness. PPML and Pretty differ only slightly. In PPML the <hov>-operator can not be combined with the tab well. In Pretty it is not possible to give the newline argument to the <hv> operator.

However by introducing new arguments for alignment and by relating Pretty to ASF + SDF we really improved the expressiveness. Alignment is very important in prettyprinting. Because the specificator can specify all the auxiliary functions he needs in ASF + SDF, he can make the alignment as sophisticated as he wants. But alas, it is still a lot of work to specify sophisticated alignment and the equations get less readable.

One consequence of the fact that more complex actions can be described in Pretty is that the prettyprinter must do a lot more work. The part of the prettyprinter that translates a text into a box-term will have more to do. It has to deal with more conditions in equations and more calculations. The box-terms will not be more difficult, because one can only use four new arguments, so the box-interpreter will do the same amount of work. It isn't a problem that prettyprinting will take more time now, because in the dutch generator the text will not be prettyprinted very often. The real text, the user is working on, is saved, so the editor will deal with the changes, the user makes, in the text itself. The tree-representation of the text will also be altered, but the tree has not to be prettyprinted. Remember that in the french generator the real text is not saved and the changed tree-representation of the text must be prettyprinted every time before the changed text can be printed. In the dutch generator the text will only be prettyprinted when the user says so. This means that prettyprinting will not slow down the editing actions of the user very much. Of course the prettyprinter for Pretty must have a reasonable response time, but now the response-time isn't critical any more.

We must realise that we can't compare the response-times of both prettyprinters yet. The prettyprinter for PPML is finished and it works, but we could not test Pretty plus ASF + SDF yet, so we don't really know if the box-interpreter is correct and how long it takes before a text is prettyprinted. The only thing we can say here is that the response-time of the Pretty-prettyprinter will have a less greater impact on the overall performance of the programming environment than the PPML-prettyprinter.

In summary we can say that prettyprint-specifications in Pretty are well readable, only when auxiliary functions and conditions are used the equations get less readable. Besides the layout formats that can be expressed in PPML, it is also possible to specify alignment in Pretty. It is still possible to put more operators and arguments in Pretty. This will make the box-interpreter more complicated and time-consuming, but the response-time of the prettyprinter will not easily put constraints on the speed at which the user can work with the programming environment.

Chapter Three : The Box-interpreter

The box-interpreter takes care of the second, language-independent, step of the prettyprinting process: the translation of a box-term into real text. First, we will discuss the specification of the box-interpreter we wrote in ASF + SDF. We will look at some of the algorithms and datastructures defined in this specification. In the next section we will show some test results. Because the new system that would support ASF + SDF was not yet ready, when we wanted to test our specification, we had to translate our specification in one that uses only ASF. We used this specification in some tests. The last test shows the prettyprinting of a small part of a Pascal-program.

3.1 A specification of the box-interpreter in ASF + SDF

The complete specification can be found in Appendix F. It is very long, so here we will only look at the main issues.

3.1.1 Frames

When we specified an implementation of the box-interpreter, we introduced one important new datastructure, the frame. In this section we will say something about the general outline of the specification and about the function of frames.

The specification consists of 8 modules with the following names, Integers, Booleans, Chars, Strings, Boxsyntax, Changelists, Frames and BoxtoString. The most important module is BoxtoString.

This module contains the main function

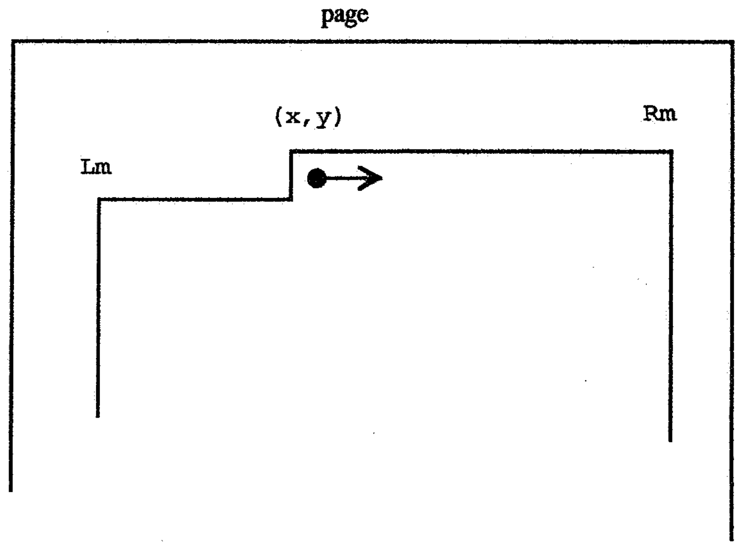
```
makestring of BOX       ->  STRING
```

However, this function doesn't do very much. It only initializes the first frame. A frame is a list of information that says how the box must be printed. It contains the following values, the startpoint of the frame (the x- and y- coordinate), the left margin, the right margin and the depth. So the frame holds the information about the sizes of the space in which the box-term must be printed and the depth of the recursive calls for this box-term. When the first frame is initialized, the startpoint will be set to column 1, line 1 and the other values will be set to the values of the symbols |<<-, ->>| and |d| in the defaults-equation. After this is done, most of the real work is done by the function

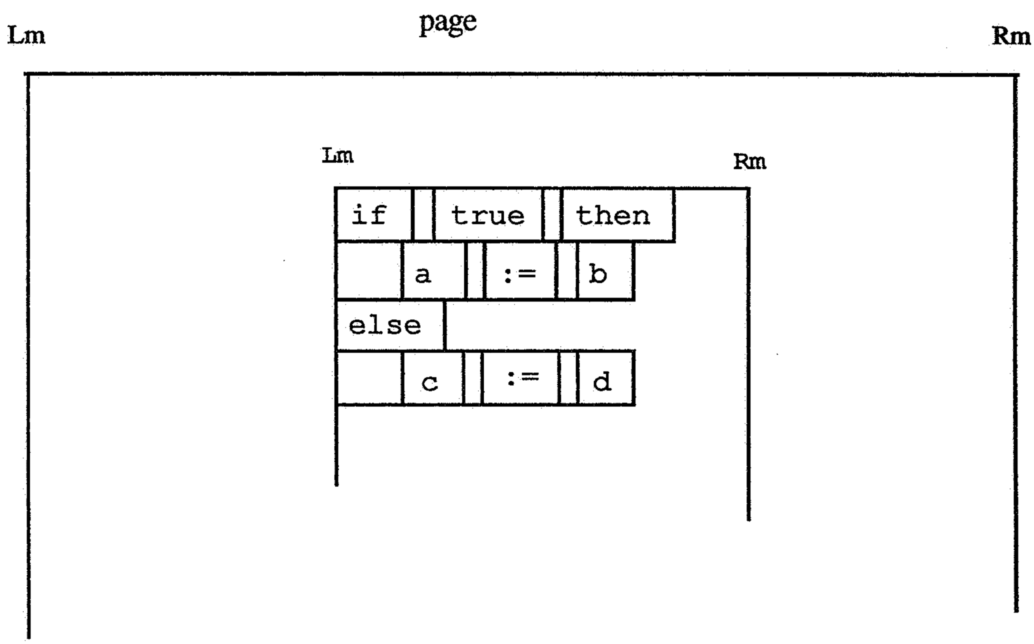
```
place BOX in FRAME      ->  STRING_END
```

A string_end is a tupel of a string and an endpoint. The function `place in` now works as follows. The frame gives the measurements of the space the box must be printed in. The function `place in` places the box in this space according to the operators in the box-term, but also accounting for the limitations of the space. When this is done, it returns a string for the whole box-term, possible with newlines in it, and the endpoint of the string. The endpoint can be the new startpoint for a next box.

The picture below shows a frame. At first the frame is open, because we don't know the endpoint of a frame, until the box-term has been printed.



Every subbox in a box-term is recursively given to the `place in`-function, so this function can translate this subbox into a string and determine the endpoint of the string in its frame. The endpoint will be the startpoint of the frame for the next subbox. Finally, all strings for the subboxes can be combined to form the string for the whole box-term. Here is an example that shows the frames in an if-statement with their begin- and end-points. Also the spaces between the words are frames, because the operators and the arguments on the subboxes are translated in literal strings between the subboxes and these are also placed in frames. When a nothing-operator or a tab is used, blanks will be placed between the subboxes. When the `;`-operator is used, no new frame is needed, the strings that are the outcome of the subboxes will then just be combined by a newline.



3.1.1 The place in-function

Now we will explain what the `place in`-function does with each different kind of box. Because this function does most of the prettyprint-actions on the boxes, this will give a clear general description of how the box-interpreter is defined. We will skip the details. More detailed information can be found in the description of the functions preceding the specification or in the specification itself, both in appendix F.

Remember that the `place in`-function gets a box and a frame as input. It will translate the box into a string and it will then return this string and the endpoint of this string in the frame. Usually it has to make some recursive calls to the same `place in`-function for the subboxes, before it can return the string and the endpoint of the string for the whole box.

If the `place in`-function gets a box surrounded by square brackets and a frame as input, it checks if the depth in the frame is zero. If not, it will lower the depth in the frame by one and go on with the box without the brackets. If the depth was zero, it will not translate the box any more, but instead it will go on with the depthstring. This is how we deal with elision. The `place in`-function also does something else when it gets a box surrounded by square brackets, it checks if the box consists of two boxes combined by an operator. If this is true, it sets in the frame the left margin equal to the x-startpoint. In this way the left margin of all the subboxes in this box is set to the beginning of this composite box. Now a piece of a sentence that does not fit on one line will begin on the correct x-position, when it is placed on a next line. We will talk about this in more detail in the next section where some more information is given about the left margin.

If the `place in`-function gets a string and a frame as input, it will try to place the string from the startpoint of the frame at one line. If it doesn't fit on this line, it will try to place the whole string on the next line, which has probably more open space. Only when this doesn't fit either, it will cut the string. It will place one part starting from the startpoint of the frame and the other part on the next line. In this way strings will only be cut when it is really necessary.

The `place in`-function handles a box preceded by a changelist in the following way. First the changelist will be applied to the frame. This is possible, because every change can be expressed by changing some values of the frame. For example, when `->Int` is in the changelist, we add Int to the x-position of the startpoint of the frame. When all changes are handled in this way, the `place in`-function will go on with the box and the new frame.

If the `place in`-function gets a box which consists of two subboxes combined by a nothing-operator, it will first determine how many blanks must be placed between the subboxes by looking at the `<->Int` -change in the changelist of the second subbox, if it has one, or else in the defaults-equation. It will create a string of the right number of blanks and then it will recursively translate the first subbox, this string of blanks and the second subbox. The resulting strings will be combined together and the endpoint of the string computed for the second subbox will be the endpoint of string for the complete box. Two boxes combined by the `;`-operator are dealt with in the same way.

Placing two boxes combined by the /-operator right is the most complicated task. Because the `->Int` and `|Int` changes will only be applied when the boxes are placed above each other, we will start by removing them from the second box. Then we will try to place the boxes on one line. We recursively place the subboxes on the line, but we continually check if the end of the line is not yet reached. If we succeed to place the subboxes without reaching the end of the line, the boxes are placed right. If not, we only then find out that boxes must be placed vertically. We now place the original boxes, with the `->Int` and `|Int` changes, under each other. Notice that we prettyprint no more than one line extra to find out if the boxes should be placed above each other or not.

Now we have described the global action of the prettyprinter.

3.1.2 The left margin

When a specificator writes down a box-term, he expects the subboxes to be placed relative to the beginning of this box-term. All indentations are relative to this beginpoint. Because the box-interpreter often places the subboxes relative to the left margin for this box, we must make sure that for many boxes the left margin is set to the beginning of the box. The following strategy is used. When the `place` in-function encounters a box, which consists of two boxes combined by an operator and which is surrounded by square brackets, it sets the left margin of its frame to the x-start position of this frame, so the left margin will be set to the beginning of the frame. Then it makes a recursive call on the box without brackets and this new frame. This will guarantee that boxes that are combined with the `;`-operator will be placed exactly above each other and that long lines will be folded in a nice way. The following two examples will show this.

This is a legal box-term

```
[ "const" [["tab] "=" ["4"];
           ["bigtab"] "=" ["8"];
           ["smalltab"] "=" ["2'] ]]
```

The large subbox containing the declarations consists of subboxes combined by the `;`-operator and is surrounded by square brackets, so the left margin for this box will be set to x-start position of this box. Therefore this box will not be printed like this

```
const tab = 4                                    const tab = 4
bigtab = 8          but like this                      bigtab = 8
smalltab = 2                                           smalltab = 2
```

This is just as one would expect, so in this example the left margin is handled correctly.

When you look at the prettyprint-specification for Pascal you 'll see that the following box-term can be created for an if-statement surrounded by begin and end.

```
[ "begin";
   -> [[ "if" [["a"] = [["b"]]] "then",
          -> [["c"] := [["veryverylongidentifier"]] ";"],
         "else",
          -> [["c"] := [["d"]] ";"]]];
   "end" ]
```

If the left margin will stay at the start position of this box, the long identifier will be folded in an unexpected way. The text will be printed like this on a small page.

```
begin
     if a := b then
        c := verylongide
ntifier ;
     else
        c := d ;
 end
```

By setting the left margin to the current x-position each moment the `place in`-function translates two boxes combined by an operator and surrounded by square brackets, in this case also when the if-statement is translated, we will get the right text.

```
begin
     if a := b then
        c := verylongide
        ntifier ;
     else
        c := d ;
 end
```

Maybe one would rather see the two parts of the long identifier printed straight under each other. It is very simple to change the definition so that this would happen. The `place in`-function would then simply change the left margin whenever it translates a box surrounded by square brackets, whether the box consist of two boxes combined by an operator or not. However there are some situations where this is not very wise. Until the implementation is properly tested, we will stick to our strategy. If the specificator really wants the parts of split strings printed straight above each other, he can put the ->0 before strings that might be split. For example he can write the box-term for the identifiers as shown below. This would solve the problem with the if-statement.

```
[26'1]    *(Id)           =  [ ->0 string(Id) ]
```

Another situation where the position of the left margin is very important is in box-terms containing lists of elements. In section 2.1.4, when we discussed the handling of elision in Pretty, we saw that we had to handle lists and list-elements in a special way to make sure that every list-element got the same holophraste level. However, there are two possible formats that assure this. They only differ in the way the left margins will be placed when the boxes are printed. We will show here both formats and explain why the first format is better.

In section 2.1.4 we specified the label-list in Pascal like this.

```
[3'1] *(label Label-list)           = [ "label" [*(Label-list)] ]


[4'1] *(Unsigned-int, Label-list) = *(Unsigned-int)"," *(Label-list)
```

If we only look at the definition of Pretty we gave so far, it would also be possible to solve the elision-problem by writing a list and its elements in the following way

```
[3'1] *(label Label-list)          = [ "label" *(Label-list) ]


[4'1] *(Unsigned-int, Label-list) = [ *(Unsigned-int) ","
                                       d+ *(Label-list) ]
```

Here we compensate for the fact that every element in the list will have a lower holophraste level than the previous element by placing the change d+ before the recursive call for the Label-list. A box-term for a real piece of Pascal-text can now look like this

```
["label" [["12"]"," d+[["16"]"," d+[["1030405"]"," d+[["23958"]]]]]]
```

Here every sublist of the Label-list is a box consisting of boxes combined by the nothing-operator and surrounded by brackets. Therefore for every sublist the left margin will be set on the start position of the sublist. This means that on a small page the box-term will be printed like this.

```
label 12 , 16 , 103040
                5 , 23
                    95
                     8
```

However you would expect this text to be printed

```
label 12 , 16 ,
      1030405 , 23958
```

Here we don't have to cut 100405 in two because it fits completely on the next line. This text would be the result if the label-list was specified in the first way, because then the sublists would not be surrounded by square brackets. Now it is obvious why the first way of specifying the handling of elision in lists is preferred above the second way.

3.2 Test results

We wanted to test if the equations in the specification of the box-interpreter were correct. Do the equations really specify the translation of a box-term into a string that shows the right format? Did we make sensible choices for the handling of exceptional situations? We would also like to investigate, if the specification for the prettyprinting of Pascal would lead to the prettyprinter we had in mind when we wrote the specification. Did we use the right operators and symbols in this specification?

Unfortunately, the generator couldn't handle equations at the time we wanted to test these things and it would take a while before the new version would be ready. If it were finished, we would just have to feed it the syntax definition of Pascal (appendix D), the specification of the prettyprinting of Pascal (appendix E) and the specification of the box-interpreter (appendix F). Then a programming environment for Pascal with a prettyprinter would be generated and we could create and prettyprint all Pascal-texts we liked. This would allow for some serious tests.

Alas, as we said before, the new version was not ready when we wanted to test our prettyprint-specification for Pascal and our specification of the box-interpreter. However there existed a program that supported ASF. Maybe we could translate the specifications written in ASF + SDF into specifications written in ASF only. ASF is a predecessor of ASF + SDF. ASF is very suitable for describing equations, but syntax can only be described in an awkward way. Unfortunately, we soon found out that the syntax of Pascal is too sophisticated and too big to describe in ASF in a reasonable way, so we couldn't describe the prettyprint-specification for Pascal in ASF.

However, because the syntax of Pretty is very small, we were able to change this syntax so that it could be described in ASF. After this was done, it was rather easy to translate the specification of the box-interpreter into ASF. We only had to change the syntax used in the equations and add a few auxiliary functions. The way the algorithms were expressed stayed the same, the same conditions, almost the same equations. The complete specification can be found in appendix G.

We will now say something about the test results in appendix H. Here we first give test input and -output used for some simple functions and for the translation of some small box-terms. When a box-term is translated, the result is a text, which consists of strings combined by $-signs. These signs stand for newlines in the text. All b's in the text stand for blanks, except of course when they occur in a word. The test output shows that simple box-terms are translated correctly.

The next test is done on a larger box-term. The box-term for a case-statement in Pascal is translated. We will now show this case-statement written in SDF-syntax. To keep the box-term simple, we left out some square brackets that don't have a real function here.

```
["case" "buff[1]" "of";
  ->4 ["'o'" ->>11 ":" ["begin" ;
                          ->4 ["filedesc" ":=" "openfile(buff)" ";" ;
                                "replybuff" ":=" "'ok'" ";"] ;
                          "end" ";"] ;
       "'r'" "," "'c'" ->>11 ":" ["replybuff" ":=" "'notopen'" ";"]] ;
 "end"]
```

In appendix H we show the same box-term, but this time written in ASF-syntax. This box-term is very complicated and unreadable. It contains 92 pairs of parentheses! Now it is obvious why we didn't do more tests on large box-terms. When we prettyprinted this large box-term, we first choose a page width of 60. In the next test we set the page width to 30. Some lines had to be folded now. For each test we first show the original test output and then we replace the b's by blanks and we remove the comma's to improve the readability of the output.

We have now described all the testing we have done. When the new version of the generator is finished the specification must be submitted to some more tests. Then we can also test the prettyprint specification of Pascal. Here we only tested one case statement.

3.3 Limitations and future work

What problems and limitations have we found so far in respect to the use of Pretty? Here we will give a list of all things that must still be improved or designed.

- every prettyprint-specification starts with two long lists of declarations. All language structures have to be declared of type STAREXPR and in the variable section for every language structure a variable must be declared. It is very awkward that one has to write down this trivial information. It would be better if this information would be derived automatically from the syntax definition of the language, where all language structures are introduced.

- Pretty still has some limitations. We would like to have an operator that forces the last and first string of two consecutive boxes to be on the same line (see section 2.3). We could also use more operators or symbols to make it easier to describe alignment, although we don't know yet what these operators or symbols should look like. Maybe when we use Pretty for real prettyprinting we will find out that some more operators or symbols are needed.

- It is not possible yet to prettyprint the comments that are placed in a text. The parser simply leaves the comments out of the parse tree, so the prettyprinter does not know where which comments must be placed. This could be solved by placing the comments also in the parse tree or by giving an extra datastructure to the prettyprinter, that holds information about the comments.

- The prettyprinter is not able yet to handle priorities on the language structures. It does not know when extra brackets have to be placed when a priority clash occurs. Only the parser has information about the priorities. It uses these priority-rules to parse the text and leaves all brackets used for grouping out of the parse tree. The prettyprinter can not determine where brackets were left out and where it must put them back into the prettyprinted text. This problem could also be solved by giving more information to the prettyprinter.

What are we going to do with the specification of the box-interpreter? Well, first it must still be integrated in the generator, so that programming environments with a prettyprinter can be generated. We can use the specification as it is. We simply let the equation manager in the generator do all the work. It can translate a box-term into a string using the equations in the specification. If it turns out that in this way prettyprinting takes too much time, we can also write a Lisp-program with the same functionality as the specification. Because all algorithms and datastructures are well specified, it will be easy to write such a Lisp-program. Most parts of the generator are already written in Lisp, the editor, the parser, the equation manager, so this Lisp program will simply be another part of the generator.

Creating a Lisp program for the specification is not the only way we can improve the response time of the prettyprinter. We can also turn the prettyprinter into an incremental one. This technique is also often used in the other parts of the generator. For the prettyprinter this would mean that it must be possible to prettyprint a part of a program without prettyprinting the whole program. So the prettyprinter must be able to prettyprint pieces of a program independently of each other. This would save a lot of work, because then it would also be possible to prettyprint a program while it is created. Continually only the parts that change will be prettyprinted. In this way the prettyprint work will be spread out in time over the creation of the whole program. Because every time only a small part is prettyprinted, the user doesn't really have to wait for it and the text will be continually in prettyprinted form on the screen.

It is not clear yet, if we can create an incremental prettyprinter out of this specification. We expect that the straightforward implementation based on frames offers some possibilities for incrementality.

But first things first, we will first try to let the specification run on the new version of the generator, when it is finished. If we still have time to do so, we will place the new test results in appendix H with the other test results. If not, the new tests will be presented in a next article about Pretty.

Acknowledgements

I would like to thank the whole dutch GIPE-team, Hans van Dijk, Casper Dik, Jan Heering, Paul Hendriks, Paul Klint, Wilco Koorn, Emma van der Meulen, Jan Rekers and Pum Walters for a very pleasant and instructive cooperation. I have enjoyed very much being a part of the GIPE-team. I especially thank Paul Klint, my supervisor, and Pum Walters for their comments and support.

References

[1] "PPML Reference Manual & Compiler Implementation", P. Borras, Fourth Annual Report Esprit Project GIPE, 1989.

[2] "The Box, a layout abstraction for User Interface Toolkits", J. Coutaz, Carnegie-Melon University, Report CMU-CS-84, Pittsburg, Pa 15213, 1984.

[3] "The Synthesizer Generator Reference Manual, second edition", Thomas Reps & Tim Teitelbaum, Department of Computer Science, Cornell University.

[4] "ASF - An algebraic specification formalism", J.A. Bergstra, J. Heering and P. Klint, chapter 1 in Algebraic Specification, ACM Press in cooperation with Addison-Wesley, 1989.

[5] "The syntax definition formalism SDF -reference manual-", J. Heering, P.R.H. Hendriks, P. Klint and J. Rekers, Report CS-R8926, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[6] "PICO Revisited", J.Heering and P. Klint, chapter 9 in Algebraic Specification, ACM Press in co-operation with Addison-Wesley, 1989.

[7] "Principles of lazy and incremental programming generation", Report CS-R8749, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[8] "Berkeley Pascal User's Manual - version 3.1, April 1986", W.N. Joy, S.L. Graham, Ch.B. Haley, M.K. McKusick and P.B. Kessler, 4.3 Berkeley Software Distribution, Computer Systems Research Group, University of California, Berkeley, 1986.

[9] "Algebraic specification of a compiler for a language with pointers", E.A. van der Meulen, Report CS-R8848, Centre for Mathematics and Computer Science, Amsterdam, 1988

Appendix A : PICO in SDF

```
module PICO-syntax
   import Naturals Strings
   export
      sorts ID   TYPE  PROGRAM  DECLS  ID-TYPE  SERIES STATEMENT  EXP
      lexical syntax
       [a-z][a-z0-9]*                                 -> ID
      context-free syntax
       begin DECLS SERIES end                         -> PROGRAM
       declare {ID-TYPE ","}* ";"                     -> DECLS
       ID ":" TYPE                                    -> ID-TYPE
       {STATEMENT ";"}*                               -> SERIES

       natural                                        -> TYPE
       string                                         -> TYPE

       ID ":=" EXP                                    -> STATEMENT
       if EXP then SERIES else SERIES fi              -> STATEMENT
       while EXP do SERIES od                         -> STATEMENT

       EXP "+" EXP                                    -> EXP {left}
       EXP "-" EXP                                    -> EXP {left}
       EXP "||" EXP                                   -> EXP {left}
       ID                                             -> EXP
       NATURAL                                        -> EXP
       STRING                                         -> EXP
       "(" EXP ")"                                    -> EXP {bracket}
```

Appendix B : PICO in PPML

```
prettyprinter of PICO is

constant
    tab = 4;

default
    <h 1>; <v 0,0>; <hv 1,0,0>; <hov 1,0,0>;

rules

*x !0                                -> "...";

pico_program(*decls, *series)        -> [<v> "begin"
                                             <v tab,0> [<v 0,1> *decls
                                                                *series]
                                             "end"];


decls(*decl, **decls)                -> [<v tab,0> "declare"
                                          [<hv> *decl
                                               (<h 0> "," **decls) ";"]];

decl(*id, *type)                     -> [<h> *id <h 0> ":" *type];

integer                              -> "integer";

string                               -> "string";

series()                             -> ;

series(*stat, **stats)               -> [<v> *stat (<h 0> ";" **stats)];

assign(*id, *exp)                    -> [<h> *id <h 0> ":=" *exp !+1];

ifthenelse(*exp, *series1, *series2) -> [<v> [<hv> "if" *exp "then"]
                                                  <v tab,0> *series
                                                  "else"
                                                  <v tab, 0> *series2
                                                  "fi"] ;

while(*exp, *series)                 -> [<v> [<hv> "while" *exp "do"]
                                                  <v tab, 0> *series
                                                  "od"];

var(*id)                             -> *id;

integer_constant(*int)               -> *int;
```

```
string_constant(*string)          -> *string;

plus_exp(*exp1, *exp2)            -> [<hv> *exp1!+ 1 "+" *exp2!+ 1];

conc_exp(*exp1, *exp2)            -> [<hv> *exp1!+ 1 "||" *exp2!+ 1];

id-atom(*x)                       -> uclcidpp(*x);

integer_const-atom(*x)            -> numberpp(*x);

string_const-atom(*x)             -> uclcidpp(*x);

meta-atom(*x)                     -> metapp(*x);

comment-atom(*x)                  -> [<h> "!" uclcidpp(*x)];

comment_s(**x)                    -> [<v> (**x)];

end prettyprinter
```

Appendix C : PICO in Pretty

The functionality of the prettyprinter, that is specified here, will not be the same as that of the prettyprinter specified in PPML in the appendix B, because here we also specify the alignment of the identifierlist. See section 2.2.1 for a specification in Pretty that has the same functionality as the one witten in PPML.

```
module Pico-box
 import Pico-syntax, Box-syntax, Integers
 export
  context-free syntax
    "*" STAREXPR                 -> BRACKBOX
    PROGRAM                      -> STAREXPR
    DECLS                        -> STAREXPR
    D-LIST "^" INT               -> STAREXPR
    ID-TYPE "^" INT              -> STAREXPR
    SERIES                       -> STAREXPR
    STATEMENT                    -> STAREXPR
    EXP                          -> STAREXPR
    {ID-TYPE ,}+                 -> ID-LIST
    [a-zA-Z0-9]                  -> IDCHAR
    IDCHAR*                      -> IDCHARS
    maxlength of ID-LIST         -> INT
    maxl"(" ID-LIST "," INT ")"  -> INT

  variables
    defaults                     -> CHANGELIST
    Stat                         -> STATEMENT
    Spaces, Maxl                 -> INT
    Series, Series1, Series2     -> SERIES
    Exp, Exp1, Exp2              -> EXP
    Decls                        -> DECLS
    Idchar                       -> IDCHAR
    Series                       -> SERIES
    Idchars                      -> IDCHARS
    Id-List                      -> ID-LIST
    Something                    -> STAREXPR
    Id                           -> ID
    Type                         -> TYPE

  equations

  [1]  defaults                  = <->1, |1, -> 4, ->>40, |<- 1, ->| 80,
                                   |d|100, d+1

  [2]  layoutchar                = " "

  [3]  *begin Decls Series end   = [ "begin";
                                     -> *Decls;
                                     -> [*Series];
                                    "end" ]
```

```
[4]   *declare ;                    = [ "declare" ";" ]

[5]   *declare Id-List;             = [ "declare";
                                        -> [*Id-List ^
                                                 maxlength of Id-List] ";" ]

[6]   *Id:Type, Id-List ^ Maxl      =   *Id ->> Maxl ":" *Type ",";
                                        *Id-List ^ Maxl

[7]   *Id:Type ^ Maxl               =   *Id ->> Maxl ":" *Type ";"

[8]   *Stat; Stats                  =   *Stat <->0 ";";
                                        *Stats
[9]   *                             =

[10]  *Id := Exp                    = [ *Id <->0 ":=" d+*Exp ]

[11]  *if Exp then Series1
        else Series2 fi             = [ "if" *Exp "then";
                                        -> [*Series1];
                                       "else";
                                        -> [*Series2];
                                       "fi" ]

[12]  *while Exp do Series od       = [ "while" *Exp "do";
                                        -> [*Series];
                                       "od" ]

[13]  *Exp1 + Exp2                  = [ d+ *Exp1 "+" d+ *Exp2 ]
[14]  *Exp1 - Exp2                  = [ d+ *Exp1 "-" d+ *Exp2 ]
[15]  *Exp1 || Exp2                 = [ d+ *Exp1 "||" d+ *Exp2 ]
[16]  *id(Chars)                    = [ terminal(Chars) ]
[17]  *nat-con(Chars)               = [ terminal(Chars) ]
[18]  *string(Chars)                = [ terminal(Chars) ]

[19]  length(Idchar Idchars)        = length(Idchars) + 1
[20]  length()                      = 0

[21]  maxlength of Id-List          = maxl(Id-List, 0)

[22]                  length(Id) > Max
      -------------------------------------------------------
      maxl(Id:Type;Id-List, Max) = maxl(Id-List, length(Id))

[23]                  length(Id) < Max
      ------------------------------------------------
      maxl(Id:Type;Id-List, Max) = maxl(Id-List, Max)

[24]  maxl( , Max)                  = Max

end Pico-box
```

Appendix D : Pascal in SDF

```
module PascalSyntax
    sorts
        StarComChar Id StringElem
        CharString Base UnsignedInt SignedInt
        UnsignedReal Number OctalConst

        Program ProgHeading Block Decl LabelDecl
        ConstDecl TypeDecl VarDecl ProcDecl FuncDecl LabelList
        ProcHeading FuncHeading Pars Par
        Const Type SimpleType FileType NonfileType
        StructType Field FieldList VariantPart Variant
        Var QualifiedVar Statement Assignment
        CaseListElem Expr SetElem ActualPar

    lexical syntax
        [ \t\n]                                    -> LAYOUT
        "{" ~[}] * "}"                             -> LAYOUT
        ~ [*]                                      -> StarComChar
        "*"+ ~ [*)]                                -> StarComChar
        "(*" StarComChar* "*"+ ")"                 -> LAYOUT
        "#include" ~[\n] * "\n"                    -> LAYOUT

        [a-zA-Z] [a-zA-Z0-9]*                      -> Id

        "oct"                                      -> Base
        "hex"                                      -> Base

        ~ ['\n]                                    -> StringElem
        "''"                                       -> StringElem
        "'" StringElem+ "'"                        -> CharString
        "#" StringElem+ "#"                        -> CharString

        [0-7]+ [bB]                                -> OctalConst
        [0-9]+                                     -> UnsignedInt
        UnsignedInt                                -> SignedInt
        [+\-] UnsignedInt                          -> SignedInt

        UnsignedInt "." UnsignedInt                -> UnsignedReal
        UnsignedInt "." UnsignedInt [eE] SignedInt -> UnsignedReal
        UnsignedInt [eE] SignedInt                 -> UnsignedReal

        UnsignedInt                                -> Number
        UnsignedReal                               -> Number
        OctalConst                                 -> Number
```

```
context-free syntax
    ProgHeading Decl* Block "."                        -> Program
    Decl*                                              -> Program
    program Id "(" {Id ","}+ ")" ";"                   -> ProgHeading
    begin {Statement ";"}+ end                         -> Block

    label {UnsignedInt ","}+                           -> Decl

    const {ConstDecl ";"}+ ";"                         -> Decl
    Id "=" Const                                       -> ConstDecl
    CharString                                         -> Const
    Id                                                 -> Const
    Number                                             -> Const
    "+" Number                                         -> Const
    "-" Number                                         -> Const

    type {TypeDecl ";"}+ ";"                           -> Decl
    Id "=" Type                                        -> TypeDecl
    SimpleType                                         -> Type
    "^" Id                                             -> Type
    FileType                                           -> Type
    StructType                                         -> Type
    packed StructType                                  -> Type

    Id                                                 -> SimpleType
    "(" {Id ","}+ ")"                                  -> SimpleType
    Const ".." Const                                   -> SimpleType
    file of NonfileType                                -> FileType
    SimpleType                                         -> NonfileType
    "^" Id                                             -> NonfileType
    StructType                                         -> NonfileType
    packed StructType                                  -> NonfileType

    array "[" {SimpleType ","}+ "]" of Type            -> StructType
    set of SimpleType                                  -> StructType
    record FieldList end                               -> StructType
    {Field ";"}+ VariantPart                           -> FieldList
    {Id ","}+ ":" Type                                 -> Field
    % empty %                                          -> Field
    case Id of {Variant ";"}+                          -> VariantPart
    case Id ":" Id of {Variant ";"}+                   -> VariantPart
    % empty %                                          -> VariantPart
    {Const ","}+ ":" "(" FieldList ")"                 -> Variant
    % empty %                                          -> Variant

    var {VarDecl ";"}+ ";"                             -> Decl
    {Id ","}+ ":" Type                                 -> VarDecl

    ProcDecl                                           -> Decl
    procedure Id Pars ";"                              -> ProcHeading
    ProcHeading Decl* Block ";"                        -> ProcDecl
    ProcHeading forward ";"                            -> ProcDecl
    ProcHeading external Id ";"                        -> ProcDecl
```

```
ProcHeading external ";"                          -> ProcDecl

FuncDecl                                          -> Decl
function Id Pars ":" Type ";"                     -> FuncHeading
FuncHeading Decl* Block ";"                       -> FuncDecl
FuncHeading forward ";"                           -> FuncDecl
FuncHeading external Id ";"                       -> FuncDecl
FuncHeading external ";"                          -> FuncDecl

"(" {Par ";"}+ ")"                                -> Pars
% empty %                                         -> Pars
{Id ","}+ ":" Id                                  -> Par
var {Id ","}+ ":" Id                              -> Par
procedure Id Pars                                 -> Par
function Id Pars ":" Id                           -> Par

Id                                                -> Var
QualifiedVar                                      -> Var
Id "[" {Expr ","}+ "]"                            -> QualifiedVar
QualifiedVar "[" {Expr ","}+ "]"                  -> QualifiedVar
Id "." Id                                         -> QualifiedVar
QualifiedVar "." Id                               -> QualifiedVar
Id "^"                                            -> QualifiedVar
QualifiedVar "^"                                  -> QualifiedVar

Var ":=" Expr                                     -> Assignment
Assignment                                        -> Statement
begin {Statement ";"}+ end                        -> Statement
if Expr then Statement                            -> Statement
if Expr then Statement else Statement             -> Statement
while Expr do Statement                           -> Statement
repeat {Statement ";"}+ until Expr                -> Statement
for Assignment to Expr do Statement               -> Statement
for Assignment downto Expr do Statement           -> Statement
case Expr of {CaseListElem ";"}+ end              -> Statement
with {Var ","}+ do Statement                      -> Statement
Id                                                -> Statement
Id "(" {ActualPar ","}+ ")"                       -> Statement
UnsignedInt Statement                             -> Statement
goto UnsignedInt                                  -> Statement
% empty %                                         -> Statement

{Const ","}+ ":" Statement                        -> CaseListElem
% empty %                                         -> CaseListElem

Number                                            -> Expr
nil                                               -> Expr
CharString                                        -> Expr
Var                                               -> Expr
Id "(" {ActualPar ","}+ ")"                       -> Expr
not Expr                                          -> Expr
"~" Expr                                          -> Expr
"+" Expr                                          -> Expr
```

```
      "-" Expr                                         -> Expr
      "[" {SetElem ","}* "]"                           -> Expr
      Expr "+" Expr                                    -> Expr {left}
      Expr "-" Expr                                    -> Expr {left}
      Expr "|" Expr                                    -> Expr {left}
      Expr or Expr                                     -> Expr {left}
      Expr "*" Expr                                    -> Expr {left}
      Expr "&" Expr                                    -> Expr {left}
      Expr "/" Expr                                    -> Expr {left}
      Expr div Expr                                    -> Expr {left}
      Expr mod Expr                                    -> Expr {left}
      Expr and Expr                                    -> Expr {left}

      Expr "=" Expr                                    -> Expr {non-assoc}
      Expr ">" Expr                                    -> Expr {non-assoc}
      Expr "<" Expr                                    -> Expr {non-assoc}
      Expr "<>" Expr                                   -> Expr {non-assoc}
      Expr "<=" Expr                                   -> Expr {non-assoc}
      Expr ">=" Expr                                   -> Expr {non-assoc}
      Expr in Expr                                     -> Expr {non-assoc}
      "(" Expr ")"                                     -> Expr {bracket}

      Expr                                             -> SetElem
      Expr ".." Expr                                   -> SetElem

      Expr                                             -> ActualPar
      Expr ":" Expr                                    -> ActualPar
      Expr ":" Expr ":" Expr                           -> ActualPar
      Expr Base                                        -> ActualPar
      Expr ":" Expr Base                               -> ActualPar

priorities
{ not Expr -> Expr, "~" Expr -> Expr,
  "+" Expr -> Expr, "-" Expr -> Expr }
>
{left:
  Expr "*" Expr -> Expr, Expr "&" Expr -> Expr, Expr "/" Expr -> Expr,
  Expr div Expr -> Expr,  Expr mod Expr -> Expr,
  Expr and Expr -> Expr }
     >
     {left:
        Expr "+" Expr -> Expr, Expr "-" Expr -> Expr, Expr "|" Expr -> Expr,
        Expr or Expr -> Expr }
     >
     {non-assoc:
        Expr "=" Expr -> Expr, Expr ">" Expr -> Expr, Expr "<" Expr -> Expr,
        Expr "<=" Expr -> Expr, Expr ">=" Expr -> Expr,
  Expr "<>" Expr -> Expr, Expr in Expr -> Expr }

priorities
      if Expr then Statement else Statement -> Statement
   >
        if Expr then Statement -> Statement
```

Appendix E : Pascal in Pretty

```
module Pascal.pretty
   import  Box-syntax, Pascal-syntax
   export
     sorts
        CONST-DECLS, TYPE-DECLS, VAR-DECLS, FIELDS

     context-free syntax
        {CONST1-DECL ";"}+                        -> CONST-DECLS
        {TYPE1-DECL ";"}+                         -> TYPE-DECLS
        {VAR1-DECL ";"}+                          -> VAR-DECLS
        {FIELD ";"}+                              -> FIELDS
        *(STAREXPR)                               -> BRACKBOX
        PROGRAM                                   -> STAREXPR
        PROG-HEADING                              -> STAREXPR
        BLOCK                                     -> STAREXPR
        DECL                                      -> STAREXPR
        DECL-LIST                                 -> STAREXPR
        LABEL-DECL                                -> STAREXPR
        CONST-DECL "^" INT                        -> STAREXPR
        TYPE-DECL "^" INT                         -> STAREXPR
        VAR-DECL "^" INT "^" INT                  -> STAREXPR
        PROC-DECL                                 -> STAREXPR
        FUNC-DECL                                 -> STAREXPR
        LABEL-LIST                                -> STAREXPR
        CONST1-DECL                               -> STAREXPR
        TYPE1-DECL                                -> STAREXPR
        VAR1-DECL "^" INT "^" INT                 -> STAREXPR
        PROC-HEADING                              -> STAREXPR
        FUNC-HEADING                              -> STAREXPR
        PARS                                      -> STAREXPR
        PAR                                       -> STAREXPR
        PAR-LIST                                  -> STAREXPR
        ID-LIST                                   -> STAREXPR
        CONST                                     -> STAREXPR
        CONST-LIST                                -> STAREXPR
        TYPE                                      -> STAREXPR
        SIMPLE-TYPE                               -> STAREXPR
        SIMPLE-TYPE-LIST                          -> STAREXPR
        FILE-TYPE                                 -> STAREXPR
        NONFILE-TYPE                              -> STAREXPR
        STRUCT-TYPE                               -> STAREXPR
        FIELD                                     -> STAREXPR
        FIELD-LIST                                -> STAREXPR
        VARIANT-PART                              -> STAREXPR
        VARIANT                                   -> STAREXPR
        VARIANT-LIST                              -> STAREXPR
        VAR                                       -> STAREXPR
        VAR-LIST                                  -> STAREXPR
        QUALIFIED-VAR                             -> STAREXPR
        STATEMENT                                 -> STAREXPR
```

```
   STAT-LIST                                    -> STAREXPR
   ASSIGNMENT                                   -> STAREXPR
   CASE-STAT-LIST                               -> STAREXPR
   CASE-LIST-ELEM                               -> STAREXPR
   EXPR                                         -> STAREXPR
   EXPR-LIST                                    -> STAREXPR
   SET-ELEM                                     -> STAREXPR
   SET-ELEM-LIST                                -> STAREXPR
   ACTUAL-PAR                                   -> STAREXPR
   ACTUAL-PAR-LIST                              -> STAREXPR
   _                                            -> SYMBOL

variables
   Prog-heading                                 -> PROG-HEADING
   Decl-list                                    -> DECL-LIST
   Block                                        -> BLOCK
   Label-list                                   -> LABEL-LIST
   Unsigned-int                                 -> UNSIGNED-INT
   Const-Decls                                  -> CONST-DECLS
   Id, Id1, Id2                                 -> ID
   Id-list                                      -> ID-LIST
   Const                                        -> CONST
   Const-list                                   -> CONST-LIST
   MaxC, MaxId, MaxIds, MaxT, Line, Halfline, Halfline3  -> INT
   Type-Decls                                   -> TYPE-DECLS
   Type                                         -> TYPE
   Var                                          -> VAR
   Var-Decls                                    -> VAR-DECLS
   Var-list                                     -> VAR-LIST
   Var1-Decl                                    -> VAR1-DECL
   Ids, Id-list                                 -> ID-LIST
   Proc-heading                                 -> PROC-HEADING
   Func-heading                                 -> FUNC-HEADING
   Pars                                         -> PARS
   Par-list                                     -> PAR-LIST
   Char-string                                  -> CHAR-STRING
   Number                                       -> NUMBER
   Sign                                         -> SIGN
   Struct-type                                  -> STRUCT-TYPE
   Simple-type                                  -> SIMPLE-TYPE
   Simple-type-list                             -> SIMPLE-TYPE-LIST
   Nonfile-type                                 -> NONFILE-TYPE
   Field                                        -> FIELD
   Fields                                       -> FIELDS
   Field-list                                   -> FIELD-LIST
   Variant                                      -> VARIANT
   Variant-part                                 -> VARIANT-PART
   Variant-list                                 -> VARIANT-LIST
   Expr, Expr1, Expr2, Expr3                    -> EXPR
   Expr-list                                    -> EXPR-LIST
   Qual-var                                     -> QUALIFIED-VAR
   Statement, Stat1, Stat2                      -> STATEMENT
   Stat-list                                    -> STAT-LIST
```

```
      Assignment                              -> ASSIGNMENT
      Case-stat-list                          -> CASE-STAT-LIST
      Case-list-elem                          -> CASE-LIST-ELEM
      Actual-par                              -> ACTUAL-PAR
      Actual-par-list                         -> ACTUAL-PAR-LIST
      Set-elem                                -> SET-ELEM
      Set-elem-list                           -> SET-ELEM-LIST
      Base                                    -> BASE

equations

  [0]    _    = <->0

[1'1]  *(Prog-heading Decl-list Block .) = [ *(Prog-heading); |2
                                             [*(Decl-list)]; |2
                                             *(Block) "." ]

[2'1]  *(program Id(Ids));              = [ "program" *(Id)_"("_ [*(Ids)]_ ");" ]

[3'1]  *(label Label-list)              = [ "label" [*(Label-list)] ]

[4'1]  *(Unsigned-int, Label-list)      =   *(Unsigned-int) "," *(Label-list)

[5'1]  *(const Const-Decls;)            = [ "const" [*(Const-Decls
                                                  ^ maxconst of Const-Decls)]  ";"]

[6'1]  *(Id = Const; Const-Decls ^ MaxC) =  *(Id) ->>MaxC "=" *(Const);
                                            *(Const-Decls ^ MaxC)

[7'1]  *(type Type-Decls;)              = [ "type" [*(Type-Decls
                                                  ^ maxid of Type-Decls)] ";" ]

[8'1]  *(Id = Type; Type-Decls ^ MaxId) =   *(Id) ->>MaxId "=" *(Type);
                                            *(Type-Decls ^ MaxId)

[9'1]  *(var Var-Decls;)                = [ "var" [*(Var-Decls
                                                  ^ maxids of Var-Decls
                                                  ^ maxtype of Var-Decls)] ";" ]

[10'1] *(Var1-Decl; Var-Decls ^MaxIds ^MaxT) = *(Var1-Decl ^ MaxIds ^ MaxT) ";"
                                               *(Var-Decls ^ MaxIds ^ MaxT)

[11'1] MaxIds + MaxT < Line
       --------------------
       *(Ids : Type ^ MaxIds ^ MaxT)    = [ [*(Ids)] ->>MaxIds ":" *(Type) ]

[11'2] MaxIds + MaxT > Line, MaxT <= 40,
       Tab = Line - MaxT - 3
       ---------------------------------
       *(Ids : Type ^ MaxIds ^ MaxT)    = [ ->|Tab [*(Ids)] ->>Tab ":" *(Type) ]
```

```
[11'3] MaxIds + MaxT > Line, MaxT > 40,
       Halfline3 = Halfline + 3
       --------------------------------
       *(Ids : Type ^ MaxIds ^ MaxT)    = [ ->|Halfline *(Ids) ->> Halfline "="
                                                                  |<- Halfline3 *(Type) ]

[12'1] *(Proc-heading Decl-list Block;) = [ *(Proc-heading);
                                             [*(Decl-list)];|2
                                            *(Block) ";"  ]

[13'1] *(Proc-heading forward;)         = [ *(Proc-heading) "forward;" ]

[14'1] *(Proc-heading external Id;)     = [ *(Proc-heading), "external" *(Id) ";" ]

[15'1] *(Proc-heading external;)        = [ *(Proc-heading) "external;" ]

[16'1] *(Func-heading Decl-list Block;) = [ *(Func-heading);
                                             [*(Decl-list)]; |2
                                            *(Block) ";"  ]

[17'1] *(Func-heading forward;)         = [ *(Func-heading) "forward;" ]

[18'1] *(Func-heading external Id;)     = [ *(Func-heading) "external" *(Id)_";" ]

[19'1] *(Func-heading external;)        = [ *(Func-heading) "external;" ]

[20'1] *(procedure Id Pars;)            = [ "procedure" *(Id) [*(Pars)] ";" ]

[21'1] *(function Id Pars : Type;)      = [ "function" *(Id)_[*(Pars)] ":" *(Type)_0";" ]

[22'1] *((Par-list))                    = [ "("_[*(Parlist)]_")" ]

[23'1] *(Par; Par-list)                 = [ *(Par)_";" [*(Par-list)] ]

[24'1] *(Id-list : Id)                  = [ [*(Id-list)] ":" *(Id) ]

[25'1] *(Char-string)                   = [ string(Charstring) ]

[26'1] *(Id)                            = [ string(Id) ]

[27'1] *(Number)                        = [ string(Number) ]

[28'1] *(Sign Number)                   =   string(Sign)_ string(Number)

[29'1] *(Const, Const-list)             =   *(Const)_"," *(Const-list)

[30'1] *(^ Id)                          = [ "^"_(Id) ]

[31'1] *(packed Struct-type)            = [ "packed" *(Struct-type) ]

[32'1] *((Id-list))                     = [ "("_[*(Id-list)]_")" ]

[33'1] *(Const..Const)                  = [ *(Const)_".."_*(Const)]
```

```
[34'1] *(Simple-type, Simple-type-list) =  *(Simple-type)_"," *(Simple-type-list)

[35'1] *(file of Nonfile-type)          = [ "file of" *(Nonfile-type) ]

[36'1] *(array[Simple-type-list] of Type) = [ "array["_[*(Simple-type-list)]_"] of"
                                                                   *(Type) ]

[37'1] *(set of Simple-type)            = [ "set of" *(Simple-type) ]

[38'1] *(record Field-list end)         = [ "record",
                                              -> [*(Fieldlist)],
                                            "end" ]

[39'1] *(Fields Variant-part)           = [ [*(Fields)] *(Variant-part) ]

[40'1] *(Field Fields)                  =  *(Field) *(Fields)

[41'1] *(case Id of Variant-list)       = [ "case" *(Id) "of";
                                              -> [*(Variant-list)] ]

[42'1] *(case Id1 : Id2 of Variant-list) = [ "case" *(Id1) ":" *(Id2) "of";
                                               -> [*(Variant-list)] ]

[43'1] *(Variant; Variant-list)         =   *(Variant)_";" ;
                                            *(Variant-list)

[44'1] *(Const-list : (Field-list))     = [ [*(Const-list)] ":" "("_[*(Field-list)]_")" ]

[45'1] *(Var, Var-list)                 =    *(Var)_"," *(Var-list)

[46'1] *(Id[Expr-list])                 = [ *(Id)_"["_ [*(Expr-list)]_"]" ]

[47'1] *(Qual-var[Expr-list])           = [ *(Qual-var)_"[" [*(Expr-list)]_"]" ]

[48'1] *(Id1.Id2)                       = [ *(Id1)_"."_*(Id2) ]

[49'1] *(Qual-var^)                     = [ *(Qual-var)_"^" ]

[50'1] *(Expr, Expr-list)               =  *(Expr)_"," *(Expr-list)

[51'1] *(begin Stat-list end)           = [ "begin";
                                              -> [*(Stat-list)];
                                            "end" ]

[52'1] *(if Expr then Stat)             = [ "if" *(Expr) "then",
                                              -> *(Stat) ]

[53'1] *(if Expr then Stat1 else Stat2) = [ "if" *(Expr) "then",
                                              -> *(Stat1),
                                            "else",
                                              -> *(Stat2) ]
```

```
[54'1] *(while Expr do Statement)          = [ "while" *(Expr) "do";
                                                -> *(Statement) ]

[55'1] *(repeat Stat-list until Expr)      = [ "repeat";
                                                -> [*(Stat-list)];
                                              "until" *(Expr) ]

[56'1] *(for Assignment to Expr do Statement) = [ "for" *(Assignment) "to" *(Expr) "do";
                                                    -> *(Statement) ]

[57'1] *(for Assignment downto Expr do Statement) =
                                              [ "for" *(Assignment) "downto" *(Expr) "do";
                                                  -> *(Statement) ]

[58'1] *(case Expr of Case-stat-list end) = [ "case" *(Expr) "of";
                                                -> [*(Case-stat-list)];
                                               "end" ]

[59'1] *(with Var-list do Statement)       = [ "with" [*(Var-list)] "do";
                                                -> *(Statement) ]

[60'1] *(Id(Actual-par-list))              = [ *(Id)_"("_[*(Actual-par-list)]_")"_]

[61'1] *(Unsigned-int Statement)           = [ *(Unsigned-int) *(Statement) ]

[62'1] *(goto Unsigned-int)                = [ "goto" *(Unsigned-int) ]

[63'1] *(Statement, Stat-list)             =   *(Statement);
                                               *(Stat-list)

[64'1] *(Var := Expr)                      = [ *(Var) ":=" *(Expr) ]

[65'1] *(Case-list-elem; Case-stat-list) =   *(Case-list-elem);
                                               *(Case-stat-list)

[66'1] *(Const-list : Statement)           = [ [*(Const-list)] ":" *(Statement) ]

[67'1] *(nil)                              = [ "nil" ]

[68'1] *(not Expr)                         = [ "not" *(Expr) ]

[69'1] *(~ Expr)                           = [ "~"  *(Expr) ]

[70'1] *([ Set-elem-list ])                = [ "["_[*(Set-elem-list)]_"]" ]

[71'1] *([])                               = [ "[]" ]

[72'1] *(Expr1 + Expr2)                    = [ *(Expr1) "+" *(Expr2) ]

[73'1] *(Expr1 - Expr2)                    = [ *(Expr1) "-" *(Expr2) ]

[74'1] *(Expr1 | Expr2)                    = [ *(Expr1) "|" *(Expr2) ]
```

```
[75'1] *(Expr1 or Expr2)              = [ *(Expr1) "or" *(Expr2) ]

[76'1] *(Expr1 * Expr2)               = [ *(Expr1) "*" *(Expr2) ]

[77'1] *(Expr1 & Expr2)               = [ *(Expr1) "&" *(Expr2) ]

[78'1] *(Expr1 / Expr2)               = [ *(Expr1) "/" *(Expr2) ]

[79'1] *(Expr1 div Expr2)             = [ *(Expr1) "div" *(Expr2) ]

[80'1] *(Expr1 mod Expr2)             = [ *(Expr1) "mod" *(Expr2) ]

[81'1] *(Expr1 and Expr2)             = [ *(Expr1) "and" *(Expr2) ]

[82'1] *(Expr1 = Expr2)               = [ *(Expr1) "=" *(Expr2) ]

[83'1] *(Expr1 > Expr2)               = [ *(Expr1) ">" *(Expr2) ]

[84'1] *(Expr1 < Expr2)               = [ *(Expr1) "<" *(Expr2) ]

[85'1] *(Expr1 <> Expr2)              = [ *(Expr1) "<>" *(Expr2) ]

[86'1] *(Expr1 <= Expr2)              = [ *(Expr1) "<=" *(Expr2) ]

[87'1] *(Expr1 >= Expr2)              = [ *(Expr1) ">=" *(Expr2) ]

[88'1] *(Expr1 in Expr2)              = [ *(Expr1) "in" *(Expr2) ]

[89'1] *((Expr1))                     = [ "("_*(Expr1)_")" ]

[90'1] *(Expr1..Expr2)                = [ *(Expr1)_".."_*(Expr2) ]

[91'1] *(Set-elem, Set-elem-list)     =   *(Set-elem)_"," *(Set-elem-list)

[93'1] *(Expr1 : Expr2)               = [ *(Expr1) ":" *(Expr2) ":" *(Expr) ]

[94'1] *(Expr1 : Expr2 : Expr3)       = [ *(Expr1) ":" *(Expr2) ":" *(Expr3) ]

[95'1] *(Expr1 Base)                  = [ *(Expr1) *(Base) ]

[96'1] *(Expr1 ":" Expr2 Base)        = [ *(Expr1) ":" *(Expr2) *(Base) ]

[97'1] *(Actual-par, Actual-par-list) =  *(Actual-par) "," *(Actual-par-list)

end
```

Appendix F : Box-interpreter in SDF

First we will give some comments on this specification. For every module we will give a brief description of its contents and we will also explain the important or difficult functions in it.

The modules

Characs

Characs stands for a list of characters. This is specified in the lexical syntax

```
~ [\n]* -> CHARACS
```

It says that a list of zero or more characters from the character-class, that consists of all characters except the newline, is of sort CHARACS. Besides this function, two other functions are defined in this module, one that computes the length of a list of characters and one that can split such a list at a particular point. Some variables are of type CHAR or CHAR+. CHAR is a predefined sort used for the connection between the context-free level and the lexical level. For every lexical sort there is always a funcion created for this connection. So also for the lexical sort CHARACS the ASF + SDF implementation creates automatically a function for this purpose. This function looks like this

```
characs "(" CHAR+ ")"          -> CHARACS
```

In the equations for the functions length and splithelp we use this functions to split every element of type CHARACS in a list of separate characters of type CHAR+.

Strings

Here we defined that a string looks like a list of characters surrounded by double quotes. Strings can be combined by the operators - (place the strings next to each other) and $ (place a newline between these strings). By giving the following priority declarations and equations, we make sure that in the end a text consists of a list of strings separated by newlines, just as one normally represents a text.

Boxsyntax

This is exactly the module we showed in section 2.1.

Changelists

Here the functions are specified for manipulating the changelists. Some functions to determine if a symbol is in a changelist and if so, what value this symbol has and for removing a symbol. There are also some special functions. The function

```
sep of SYMBOL before BOX   -> STRING-BOX
```

returns the literal string that must be placed before a box, because of the operator that is applied to this box and the maybe slightly changed box. If the box is combined with another box with the nothing operator, the function-call "sep <-> before box" will be done, because a string of blanks will be placed between the box and the number of blanks depends on whether there is a <-> Int change in the changelist of the box or not. The function will first determine if there is a <-> Int change in the changelist of the box. If so, it will return a string of Int layoutcharacters and also the box without the <-> Int change in the changelist. If <-> Int is not in the changelist, it will refer to the <-> in the defaults-equation and it will return a string of the default number of layoutcharacters and the unchanged box. The function

rmoptionals from BOX -> BOX

removes the -> and | changes from the changelist of this box. This is necessary when boxes that are combined with the / -operator must be printed. First ,we will try to print the boxes on one line without these changes. Only when we have found out that the boxes must be placed below each other the changes must be applied.

Frames

Here we define some functions to retrieve values from a frame and to change the values in a frame. There are also some auxiliary functions. The function

from POINT to POINT -> STRING

is very useful when the endpoint of one box as well as the startpoint of the next box are known and those points are not the same. For example, when a box has an absolute tab in its changelist, the startpoint of this box will be changed and will not longer be the same as the endpoint of the previous box. This function will now deliver the string of blanks and newlines that must be placed between those two points to connect them. The function

apply CHANGELIST to FRAME -> FRAME

will only be used to construct the first frame according to the defaults-equation. The leftmargin, the rightmargin and the depth in this frame will take the values of their symbols in the defaults-equation. Ofcourse we will have to check that the leftmargin is smaller than the rightmargin. The startpoint of the frame will be set to (leftmargin, 1), the beginning of the first line. The function

modify CHANGELIST to FRAME -> FRAME

applies the changes in the changelist to the frame. All changes placed before a box can be expressed by changing the values of the frame for this box. The definition of the function modify consists of a lot of equations, because it has to deal with nine different changes and because it has to make sure some constraints will still be true when the changes are applied: the leftmargin must be smaller than or equal to the rightmargin; the x-position must be between the left- and the rightmargin; the depth may not be negative, etc.. When a change can 't be applied because of the constraints, the frame will not be changed by it or it will be changed in an expected manner.

BoxtoString

See section 3.1 for a description of this module.

The specification

```
specification BoxToString

module Chars
  import Integers
  export
   sorts CHARACS TWOCHARACS
   lexical syntax
     [ \t\n\r]                                  -> LAYOUT
     ~[\n]*                                     -> CHARACS

   context-free syntax
     "(" CHARACS ")"                            -> CHARACS  {bracket}
     length of CHARACS                          -> INT
     CHARACS "&" CHARACS                        -> TWOCHARACS
     split CHARACS at INT                       -> TWOCHARACS
     splithelp "(" CHARACS "," CHARACS ")" at INT  -> TWOCHARACS

   variables
     Char                                       -> CHAR
     Chars, Chars1, Chars2                      -> CHAR+
     Characs Characs1 Characs2                  -> CHARACS

    equations

    [1'1]    length of characs(Char Chars)  =   length of Chars + 1
    [1'2]    length of characs()            =   0

    [2'1]    Int < 0 = true
             ---------------
             split Characs at Int           =   & Characs

    [2'2]    Int >= 0 = true
             ----------------
             split Characs at Int           =   splithelp ( , Characs) at Int

    [3'1]    Int > 0 = true
             ---------------
             splithelp (characs(Chars1), characs(Char Chars2)) at Int =
                                   split (characs(Chars1 Char), characs(Chars2)) at Int - 1

    [3'2]    Int <= 0 = true
             ----------------
             splithelp (Chars1, Chars2) at Int   =   Chars1 & Chars2

    [3'3]    Int > 0 = true
             ---------------
             splithelp (Chars, ) at Int     =   Chars &
```

```
module Strings
  import
    Chars Booleans
  export
   sorts STRING TWOSTRINGS
   lexical syntax
     "\"" CHARACS "\""                        -> STRING

   context-free syntax
     STRING ":" STRING                        -> STRING
     STRING "$" STRING                        -> STRING
     "(" STRING ")"                           -> STRING {bracket}
     length of STRING                         -> INT
     STRING "&" STRING                        -> TWOSTRINGS
     split STRING at INT                      -> TWOSTRINGS
     newline in STRING                        -> BOOL
     newline                                  -> STRING
     string of INT newlines                   -> STRING
     layoutstring                             -> STRING
     string of INT layoutchars                -> STRING

  priorities
     STRING ":" STRING -> STRING  >
     STRING "$" STRING -> STRING

  variables
     Chars Chars [12]                         -> CHARS
     Int                                      -> INTEGER

  equations

  [1'1]   string(\" Chars1 \") : string(\" Chars2 \")  =  string(\" Chars1 Chars2 \")
  [1'2]   String1 : (String2 $ String3)    =   String1 : String2 $ String3
  [1'3]   (String1 $ String2) : String3    =   String1 $ String2 : String3

  [2'1]   length of string(\" Chars \")    =   length of Chars
  [2'2]   length of String1 $ String2      =   length of String1 + length of String2

  [3'1]   split Chars at Int = Chars1 & Chars2
          ------------------------------------
          split string(\" Chars \") at Int  =
                                          string(\" Chars1 \") & string(\" Chars2 \")

  [3'2]   split String1 $ String2 at Int   =   split String1 : String2 at Int

  [4'1]   newline in string(\" Chars \")   =  false
  [4'2]   newline in String1 $ String2     =  true

  [5'1]   newline                          =   string of 1 newlines
```

```
[6'1]   Int > 0 = true
        --------------
        string of Int newlines          =  ""$ string of Int - 1 newlines

[6'2]   Int <= 0 = true
        ------------------
        string of Int newlines          =  ""

[7'1]   Int > 0 = true
        --------------
        string of Int layoutchars       =  layoutstring :
                                           string of Int - 1 layoutchars

[7'2]   Int <= 0 = true
        ---------------
        string of Int layoutchars       =  ""


module Box-syntax
  import Strings
  export
   sorts SYMBOL CHANGE CHANGELIST BOX
   lexical syntax
      [ \t\n\r]                       -> LAYOUT
      "<->"                           -> SYMBOL
      "|"                             -> SYMBOL
      "->"                            -> SYMBOL
      "->>"                           -> SYMBOL
      "|<<-"                          -> SYMBOL
      "->>|"                          -> SYMBOL
      "|<-+"                          -> SYMBOL
      "|d|"                           -> SYMBOL
      "d+"                            -> SYMBOL
   context-free syntax
      STRING                          -> BOX
      SYMBOL INT                      -> CHANGE
      SYMBOL                          -> CHANGE
      {CHANGE ","}+                   -> CHANGELIST
      "(" CHANGELIST ")"              -> CHANGELIST {bracket}
      CHANGELIST BOX                  -> BOX
      BOX BOX                         -> BOX {right}
      BOX "/" BOX                     -> BOX {right}
      BOX ";" BOX                     -> BOX {right}
      "[" BOX "]"                     -> BOX
      "(" BOX ")"                     -> BOX {bracket}
      defaults                        -> CHANGELIST
      depthstring                     -> STRING
   priorities
      CHANGELIST BOX -> BOX       >
      BOX "_" BOX -> BOX          >
      BOX BOX -> BOX              >
      BOX "/" BOX -> BOX          >
      BOX ";" BOX -> BOX
```

```
  equations

%% Here only example-values for defaults, layoutstring and depthstring are
%% specified. These equations should be placed in each prettyprint specification
%% of a language, maybe with different values.

    [1'1]    defaults                      =   <-> 1, | 1, -> 4, ->> 8, |<<- 1,
                                               ->>| 30, |<+ 4, |d| 100, d+ 1
    [2'1]    layoutstring                  =   " "
    [3'1]    depthstring                   =   "..."


module Changelists
  import Box-syntax
  export
   sorts CSTARLIST
   context-free syntax
     eq "(" SYMBOL "," SYMBOL ")"          -> BOOL
     {CHANGE ","}*                         -> CSTARLIST
     "(" CSTARLIST ")"                     -> CSTARLIST {bracket}
     changelist of BOX                     -> CSTARLIST
     SYMBOL in CHANGE                      -> BOOL
     SYMBOL in CSTARLIST                   -> BOOL
     value SYMBOL in CSTARLIST             -> INT
     remove SYMBOL from CSTARLIST          -> CSTARLIST
     remove SYMBOL from c-list of BOX      -> BOX
     changetab in BOX                      -> TAB
     makesep SYMBOL with INT               -> STRING
     sep of SYMBOL before BOX              -> STRING-BOX
     STRING & BOX                          -> STRING-BOX
     rmoptionals from Box                  -> BOX

   variables
     Symbol[12]                            -> SYMBOL
     Change                                -> CHANGE
     Changelist Newc-list C-list2          -> CHANGELIST
     Int Int[12]                           -> INT
     String                                -> STRING
     Box Box[12]                           -> BOX

   equations

   [1'1]    eq(<->, <->)                   = true
   [1'2]    eq(|, |)                       = true
   [1'3]    eq(->, ->)                     = true
   [1'4]    eq(->>, ->>)                   = true
   [1'5]    eq(|<<-, |<<-)                 = true
   [1'6]    eq(->>|, ->>|)                 = true
   [1'7]    eq(|<-+, |<-+)                 = true
   [1'8]    eq(|d|, |d|)                   = true
   [1'9]    eq(d+, d+)                     = true
```

```
[1'10]   Symbol1 != Symbol2
         ------------------
         eq(Symbol1, Symbol2)              = false

[2'1]    Symbol1 in Symbol2 Int            = eq(Symbol1, Symbol2)

[2'2]    Symbol1 in Symbol2                = eq(Symbol1, Symbol2)

[3'1]    Symbol in Change = true
         -----------------------
         Symbol in Change, Changelist      = true

[3'2]    Symbol in Change = false
         ------------------------
         Symbol in Change, Changelist      = Symbol in Changelist

[3'3]    Symbol in  = false

[4'1]    eq(Symbol1, Symbol2) = true
         ---------------------------
         value Symbol1 in Symbol2 Int, Changelist  = Int

[4'2]    eq(Symbol1, Symbol2) = true
         ---------------------------
         value Symbol1 in Symbol2, Changelist  = value Symbol1 in defaults

[4'3]    Symbol in Change = false
         ------------------------
         value Symbol in Change, Changelist  = value Symbol in Changelist

[4'4]    value Symbol in                   = 0

[5'1]    changelist of String              =
[5'2]    changelist of Changelist Box      = Changelist
[5'3]    changelist of Box1 Box2           = changelist of Box1
[5'4]    changelist of Box1/ Box2          = changelist of Box1
[5'5]    changelist of Box1; Box2          = changelist of Box1
[5'6]    changelist of [Box]               =

[6'1]    Symbol in Change = true
         ------------------------------
         remove Symbol from Change, Changelist = Changelist

[6'2]    Symbol in Change = false
         -------------------------------
         remove Symbol from Change, Changelist = Change, remove Symbol from Changelist

[6'3]    remove Symbol from                =

[7'1]    remove Symbol from c-list of String = String
[7'2]    remove Symbol from c-list of Changelist Box  =
                                   (remove Symbol from Changelist) Box
```

```
[7'3]    remove Symbol from c-list of Box1 Box2    =
                                        (remove Symbol from c-list of Box1) Box2

[7'4]    remove Symbol from c-list of Box1, Box2   =
                                        (remove Symbol from c-list of Box1)/ Box2

[7'5]    remove Symbol from c-list of Box1; Box2   =
                                        (remove Symbol from c-list of Box1); Box2

[7'6]    remove Symbol from c-list of [Box]        = [Box]

[8'1]    -> in Changelist = false
         ------------------------
         changetab in Changelist Box      = Changelist Box

[8'2]    -> in Changelist = true,
         <-> in Changelist = true,
         value -> in Changelist = Int2,
         value <-> in Changelist = Int1,
         remove -> from Changelist = Newc-list
         --------------------------------------
         changetab in Changelist Box      =   -> Int2 - Int1/Newc-list Box

[8'3]    -> in Changelist = true,
         <-> in Changelist = false,
         value -> in Changelist = Int2,
         value <-> in defaults = Int1,
         remove -> from Changelist = Newc-list
         --------------------------------------
         changetab in Changelist Box      = -> Int2 - Int1/Newc-list Box

[8'4]    changetab in String              = String
[8'5]    changetab in [Box]               = [Box]
[8'6]    changetab in Box1 Box2           = Box1 Box2
[8'7]    changetab in Box1/ Box2          = Box1/ Box2
[8'8]    changetab in Box1; Box2          = Box1; Box2

[9'1]    makesep <-> with Int             = string of Int layoutchars

[9'2]    makesep | with Int               = string of Int newlines

[9'3]    eq(Symbol, <->) = false, eq(Symbol, |) = false
         -----------------------------------------------
         makesep Symbol with Int          = ""

[10'1]   Symbol in (changelist of Box) = true
         -------------------------------------
         sep of Symbol before Box =
                 makesep Symbol with (value Symbol in changelist of Box) &
                                                      remove Symbol from Box
```

```
[10'2]   Symbol in (changelist of Box) = false
         -------------------------------------
         sep of Symbol before Box =
                        makesep Symbol with (value Symbol in defaults) & Box

[11'1]  rmoptionals from String           = String
[11'2]  rmoptionals from [Box]            = [Box]
[11'3]  or(-> in Change, | in Change) = true
        -------------------------------------
        rmoptionals from Change Box       = Box

[11'4]  or(-> in Change, | in Change) = false
        -------------------------------------
        rmoptionals from Change Box       =  Change Box

[11'5]  remove -> from Change,Changelist = C-list2,
        remove  | from C-list2 = Newc-list
        -------------------------------------------
        rmoptionals from Change,Changelist Box  = Newc-list Box

[11'6]  rmoptionals from Box1 Box2       = rmoptionals from Box1 Box2
[11'7]  rmoptionals from Box1 / Box2     = Box1 / rmoptionals from Box2
[11'8]  rmoptionals from Box1 ; Box2     = Box1 ; Box2


module Frame
  import Changelists
  export
    sorts FRAME
    context-free syntax
        "(" INT "," INT ")"                  -> POINT
        POINT"," INT"," INT"," INT           -> FRAME
        start of FRAME                       -> POINT
        xpoint of FRAME                      -> INT
        ypoint of FRAME                      -> INT
        leftm of FRAME                       -> INT
        rightm of FRAME                      -> INT
        depth of FRAME                       -> INT
        newstart POINT in FRAME              -> FRAME
        newleftm INT in FRAME                -> FRAME
        newrightm INT in FRAME               -> FRAME
        newdepth INT in FRAME                -> FRAME
        from POINT to POINT                  -> STRING
        makeframe outof CHANGELIST           -> FRAME
        apply CHANGELIST to FRAME            -> FRAME
        modify CHANGELIST in FRAME           -> FRAME
        same-line POINT and FRAME            -> BOOL

    variables
        X X2 Y Y2 Lm Lm2 Rm Rm2 D D2 Width
        Tabline Spaces Lines Indent Tab Int  -> INT
        Point Point2                         -> POINT
```

```
    Frame                                    -> FRAME
    Symbol                                   -> SYMBOL
    Change                                   -> CHANGE
    Changelist                               -> CHANGELIST

equations

 [1'1]  start of Point, Lm, Rm, D               = Point
 [2'1]  xpoint of (X,Y), Lm, Rm, D              = X
 [3'1]  ypoint of (X,Y), Lm, Rm, D              = Y
 [4'1]  leftm of Point, Lm, Rm, D               = Lm
 [5'1]  rightm of Point, Lm, Rm, D              = Rm
 [6'1]  depth of Point, Lm, Rm, D               = D
 [7'1]  newstart Point2 in Point, Lm, Rm, D     = Point2, Lm, Rm, D
 [8'1]  newleftm Lm2 in Point, Lm, Rm, D        = Point, Lm2, Rm, D
 [9'1]  newrightm Rm2 in Point, Lm, Rm, D       = Point, Lm, Rm2, D
[10'1]  newdepth D2 in Point, Lm, Rm, D         = Point, Lm, Rm, D2

[11'1]  Y1 = Y2,
        X2 > X1 = true,
        X = X2 - X1
        ---------------
        from (X1,Y1) to (X2,Y2)             = string of X layoutchars

[11'2]  Y = Y2 - Y1, Y > 0 = true
        -------------------------
        from (X1,Y1) to (X2,Y2)             =  string of Y newlines:
                                               from (value |<- in defaults,Y2) to (X2,Y2)

[12'1]  makeframe outof Changelist          = apply Changelist to (1,1), 1, 80, 1

[13'1]  apply Change, Changelist to Frame   = apply Changelist to (apply Change to Frame)

[13'2]  apply <->Spaces to Frame            = Frame
[13'3]  apply |Lines to Frame               = Frame
[13'4]  apply ->Tab to Frame                = Frame
[13'5]  apply ->>Tab to Frame               = Frame
[13'6]  apply |<-+Int to Frame              = Frame

[13'7]  Lm2 <= Rm = true
        ----------------
        apply |<-Lm2 to (X,Y), Lm, Rm, D    = (Lm2,Y), Lm2, Rm, D

[13'8]  Lm2 > Rm = true
        ---------------------
        apply |<-Lm2 to Point2, Lm, Rm, D  = Point, Lm, Rm, D

[13'9]  Rm2 >= Lm = true,
        Rm2 >= X  = true
        -----------------
        apply ->|Rm2 to (X,Y), Lm, Rm, D    = (X,Y), Lm, Rm2, D
```

```
[13'10] Rm2 >= Lm = true,
        Rm2 < X   = true
        -----------------
        apply ->|Rm2 to (X,Y), Lm, Rm, D   = (Lm,Y), Lm, Rm2, D

[13'11] Rm2 < Lm = true
        ---------------
        apply ->|Rm2 to Point, Lm, Rm, D   = Point, Lm, Rm, D

[13'12] D2 >= 0 = true
        --------------
        apply |d|D2 to Point, Lm, Rm, D    = Point, Lm, Rm, D2

[13'13] D2 < 0 = true
        -------------
        apply |d|D2 to Frame               = Frame

[13'14] apply d+D2 to Frame                = Frame

 [14'1] modify Change, Changelist in Frame =
                                    modify Changelist in (modify Change in Frame)

[14'2]  modify Symbol in Frame             =
                                    modify Symbol (value Symbol in defaults) in Frame

[14'3]  X + Int <= Rm = true,
        Int >= 0      = true
        ---------------------
        modify <->Int in (X,Y), Lm, Rm, D  = (X + Int, Y), Lm, Rm, D

[14'4]  Int >= 0 = true
        ---------------
        modify <->Int in (X,Y), Lm, Rm, D  = (X + Int, Y), Lm, Rm, D

[14'5]  X + Int = Width, Width > Rm = true
        ----------------------------------
        modify <->Int in (X,Y), Lm, Rm, D  = (Lm + (Width - Rm) - 1,Y + 1), Lm, Rm, D

 [14'6] Int >= 0  = true,
        X + Int = Tabline,
        Tabline <= Rm = true
        --------------------
        modify ->Int in (X,Y), Lm, Rm, D = (Tabline,Y), Tabline, Rm, D

[14'7]  Int < 0 = true
        --------------
        modify ->Int in Frame              = Frame

[14'8]  X + Int > Rm = true
        -------------------
        modify ->Int in (X,Y), Lm, Rm, D = (X,Y), Lm, Rm, D
```

```
[14'9]   Lm + Int - 1 >= X  = true,
         Lm + Int - 1 <= Rm = true
         --------------------------
         modify ->>Int in (X,Y), Lm, Rm, D = (Lm + Int - 1,Y), Int, Rm, D

[14'10]  Lm + Int - 1 < X = true
         ----------------------------
         modify ->>Int in (X,Y), Lm, Rm, D  = (X,Y), Lm, Rm, D

[14'11]  Lm + Int - 1 >= X = true,
         Lm + Int - 1 > Rm = true
         -------------------------
         modify ->>Int in (X,Y), Lm, Rm, D  = (X,Y), Lm, Rm, D

[14'12]  Int > 0 = true
         ----------------
         modify |Int in (X,Y), Lm, Rm, D    = (Lm, Y + Int), Lm, Rm, D

[14'13]  Int <= 0 = true
         ------------------
         modify |Int in Frame               = Frame

[14'14]  Int >= (value |<- in defaults) = true,
         Int <= Rm = true,
         X >= Int = true
         --------------------------------------
         modify |<-Int in (X,Y), Lm, Rm, D  = (X,Y), Int, Rm, D

[14'15]  Int >= (value |<- in defaults) = true,
         Int <= Rm = true,
         X < Int = true
         --------------------------------------
         modify |<-Int in (X,Y), Lm, Rm, D  = (Int,Y), Int, Rm, D

[14'16]  Int < (value |<- in defaults) = true
         ------------------------------------
         modify |<-Int in Frame             = Frame

[14'17]  Int >= value |<- in defaults = true,
         Int > Rm = true
         ------------------------------------
         modify |<-Int in (X,Y), Lm, Rm, D  = (X,Y), Lm, Rm, D

[14'18]  Int <= value ->| in defaults = true,
         Int >= Lm = true,
         X <= Int = true
         ------------------------------------
         modify ->|Int in (X,Y), Lm, Rm, D  = (X,Y), Lm, Int, D
```

```
[14'19]  Int <= value ->| in defaults = true,
         Int >= Lm = true,
         X > Int = true
         ------------------------------------
         modify ->|Int in (X,Y), Lm, Rm, D  = (X,Y), Lm, Rm, D

[14'20]  Int <= value ->| in defaults = true,
         Int < Lm = true
         ------------------------------------
         modify ->|Int in (X,Y), Lm, Rm, D  = (X,Y), Lm, Rm, D

[14'21]  Int > value ->| in defaults = true
         ----------------------------------
         modify ->|Int in Frame              = Frame

[14'22]  Int >= 0 = true
         ---------------
         modify |d|Int in Point, Lm, Rm, D  = Point, Lm, Rm, Int

[14'23]  Int < 0 = true
         --------------
         modify |d|Int in Frame              = Frame

[14'24]  D + Int >= 0 = true
         -------------------
         modify d+Int in Point, Lm, Rm, D   = Point, Lm, Rm, D + Int

[14'25]  D + Int < 0 = true
         ------------------
         modify d+Int in Point, Lm, Rm, D   = Point, Lm, Rm, 0

[15'1]  same-line (X1,Y1) and (X2,Y2), Lm, Rm, D = or(eq(Y1, Y2),
                                                    and(eq(Y1, Y2 + 1), eq(X1, Lm)))


module BoxtoString
  import Frame
  export
    sorts
        STRING_END RESULT
    context-free syntax
        makestring of BOX              -> STRING
        STRING ending at POINT         -> STRING_END
        place BOX in FRAME             -> STRING_END
        string of STRING_END           -> STRING
        BOOL & STRING & POINT          -> RESULT
        fitshor BOX in FRAME           -> RESULT
        vertical BOX                   -> BOX
        is-duobox BOX                  -> BOOL

  variables
        Int NewD Length Nextline X[1-4] Y[1-4] D          -> INT
        Atthisline Atnextline String String[1-6] Sep Newline   -> STRING
```

```
      Frame Newframe                                       -> FRAME
      Point Point [1-6]                                    -> POINT
      Box Box [123] Nbox [12] Newbox [12]                  -> BOX
      Change                                               -> CHANGE
      Changelist, C-list, C-list2, Newc-list               -> CHANGELIST

 equations

[1'1]  makeframe outof defaults = Frame
       --------------------------------
       makestring of Box                  = from (1,1) to start of Frame:
                                            string of place Box in Frame

[2'1]  string of String ending at Point   = String

[3'1]  depth of Frame = 1
       ------------------
       place [Box] in Frame               = place depthstring in Frame

[3'2]  depth of Frame > 1 = true,
       is-duobox Box = false,
       NewD = depth of Frame - 1,
       newdepth NewD in Frame = Newframe
       ---------------------------------
       place [Box] in Frame               = place Box in Newframe

[3'3]  depth of Frame > 1 = true,
       is-duobox Box = true,
       NewD = depth of Frame - 1,
       Int = xpoint of Frame,
       newdepth NewD in newleftm Int in Frame = Newframe
       -------------------------------------------------
       place [Box] in Frame               = place Box in Newframe

[3'4]  newline in String = false,
       Length = length of String,
       Length =< rightm of Frame - xpoint of Frame = true
       --------------------------------------------------
       place String in Frame              = String ending at (xpoint of Frame + Length,
                                                              ypoint of Frame)

[3'5]  newline in String = false,
       Length = length of String,
       Length = rightm of Frame - xpoint of Frame + 1,
       Newline = from (rightm of Frame, ypoint of Frame) to
                      (leftm of Frame, ypoint of Frame + 1)
       -----------------------------------------------------
       place String in Frame              = (String: Newline) ending at
                                               (leftm of Frame , ypoint of Frame + 1)
```

```
[3'6]  newline in String = false,
       Length = length of String,
       Nextline = ypoint of Frame + 1,
       Length > rightm of Frame - xpoint of Frame + 1 = true,
       Length =< rightm of Frame - leftm of Frame + 1 = true
       -------------------------------------------------------
       place String in Frame                 =
                       (from start of Frame to (leftm of Frame, Nextline): String)
                                   ending at (leftm of Frame + Length, Nextline)

[3'7]  newline in String = false,
       length of String > rightm of Frame - leftm of Frame + 1 = true,
       split String at rightm of Frame - xpoint of Frame + 1 = Atthisline & Atnextline
       Newline = from (rightm of Frame, ypoint of Frame) to
                      (leftm of Frame, ypoint of Frame + 1),
       place Atnextline in newstart (leftm of Frame, ypoint of Frame + 1) in Frame =
                                                String2 ending at Point2
       ---------------------------------------------------------------------------------
       place String in Frame                 =
                       (Atthisline: Newline: String2) ending at Point2

[3'8]  place String1 in Frame = String3 & (X3,Y3),
       X3 = leftm of Frame = false,
       from (rightm of Frame, Y3) to (leftm of Frame, Y3 + 1) = Newline,
       place String2 in newstart (leftm of Frame,Y3 + 1) in Frame = String4 & Point4
       ----------------------------------------------------------------------------
       place String1 $ String2 in Frame   = String3 : Newline : String4 & point4

[3'9]  place String1 in Frame = String3 & (X3,Y3),
       X3 = leftm of Frame = true,
       place String2 in newstart (X3,Y3) in Frame = String4 & Point4
       ------------------------------------------------------------
       place String1 $ String2 in Frame   = String3 : String4 & point4

[3'10] modify C-list in Frame = Newframe,
       from start of Frame to start of Newframe = String1,
       place Box in Newframe = String2 ending at Point2
       ----------------------------------------------------
       place C-list Box in Frame          = (String1: String2) ending at Point2

[3'11] changetab in Box2 = Nbox2,
       sep <-> before Nbox2 = Sep & Newbox2,
       place Box1 in Frame = String1 ending at Point1,
       place Sep in newstart Point1 in Frame = String2 ending at Point2,
       place Newbox2 in newstart Point2 in Frame = String3 ending at Point3
       -------------------------------------------------------------------
       place Box1 Box2 in Frame           = (String1: String2: String3) ending at Point3
```

```
[3'12]  sep | before Box2 = Sep & Newbox2,
        place Box1 in Frame = String1 ending at Point1,
        place Sep in newstart Point1 in Frame = String2 ending at Point2,
        place Newbox2 in newstart Point2 in Frame = String3 ending at Point3
        -------------------------------------------------------------------
        place Box1; Box2 in Frame          = (String1: String2: String3) ending at Point3

[3'13]  rmoptionals from Box2 = Newbox2,
        fitshor Box1 Box2 in Frame = true & String & Point
        ---------------------------------------------------
        place Box1 / Box2 in Frame         = String & Point

[3'14]  rmoptionals from Box2 = Newbox2,
        fitshor Box1 Box2 in Frame = false & String & Point
        ----------------------------------------------------
        place Box1 / Box2 in Frame         = place vertical(Box1 / Box2) in Frame

 [4'1]  depth of Frame <= 1 = true
        --------------------------
        fitshor [Box] in Frame             = fitshor depthstring in Frame

 [4'2]  depth of Frame > 1 = true,
        is-duobox Box = false,
        NewD = depth of Frame - 1,
        newdepth NewD in Frame = Newframe
        ---------------------------------
        fitshor [Box] in Frame             = fitshor Box in Newframe

 [4'3]  depth of Frame > 1 = true,
        is-duobox Box = true,
        NewD = depth of Frame - 1,
        Int = xpoint of Frame,
        newdepth NewD in newleftm Int in Frame = Newframe
        -------------------------------------------------
        fitshor [Box] in Frame             = fitshor Box in Newframe

 [4'4]  newline in String = true
        ------------------------
        fitshor String in Frame            = false & "" & (0,0)

 [4'5]  newline in String = false,
        Length = length of String,
        Length > rightm of Frame - xpoint of Frame + 1 = true
        ------------------------------------------------------
        fitshor String in Frame            = false & "" & (0,0)

 [4'6]  newline in String = false,
        Length = length of String,
        Length <= rightm of Frame - xpoint of Frame = true
        --------------------------------------------------
        fitshor String in Frame            = true & String & (xpoint of Frame + Length,
                                                              ypoint of Frame)
```

```
[4'7]   newline in String = false,
        length of String = rightm of Frame - xpoint of Frame + nat([1]),
        Newline = from (rightm of Frame, ypoint of Frame) to
                       (leftm of Frame, ypoint of Frame + 1)
        ----------------------------------------------------------------
        fitshor String in Frame            = true & String : Newline &
                                               (leftm of Frame, ypoint of Frame + 1)

[4'8]   modify C-list in Frame = Newframe,
        same-line start of Newframe and Frame = false
        ---------------------------------------------
        fitshor C-list Box in Frame        = false & "" & (0,0)

[4'9]   modify C-list in Frame = Newframe,
        same-line start of Newframe and Frame = true,
        fitshor Box in Newframe = false & "" & (0,0)
        ---------------------------------------------
        fitshor C-list Box in Frame        = false & "" & (0,0)

[4'10]  modify C-list in Frame = Newframe,
        same-line start of Newframe and Frame = true,
        fitshor Box in Newframe = true & String2 & Point2,
        from start of Frame to start of Newframe = String1
        ---------------------------------------------------
        fitshor C-list Box in Frame        = true & String1 : String2 & Point2


[4'11]  fitshor Box1 in Frame = false & "" & (0,0)
        --------------------------------------------
        fitshor Box1 Box2 in Frame         = false & "" & (0,0)

[4'12]  fitshor Box1 in Frame = true & String1 & Point1,
        changetab in Box2 = Nbox2,
        sep <-> before Nbox2 = Sep & Newbox2,
        fitshor Sep in newstart Point1 in Frame = false & "" & (0,0)
        -------------------------------------------------------------
        fitshor Box1 Box2 in Frame         = false & "" & (0,0)

[4'13]  fitshor Box1 in Frame = true & String1 & Point1,
        changetab in Box2 = Nbox2,
        sep <-> before Nbox2 = Sep & Newbox2,
        fitshor Sep in newstart Point1 in Frame = true & String2 & Point2,
        fitshor Newbox in newstart Point2 in Frame = false & "" & (0,0)
        -------------------------------------------------------------------
        fitshor Box1 Box2 in Frame         = false & "" & (0,0)

[4'14]  fitshor Box1 in Frame = true & String1 & Point1,
        changetab in Box2 = Nbox2,
        sep <-> before Nbox2 = Sep & Newbox2,
        fitshor Sep in newstart Point1 in Frame = true & String2 & Point2,
        fitshor Newbox in newstart Point2 in Frame = true & String3 & Point3
        ---------------------------------------------------------------------
        fitshor Box1 Box2 in Frame         = true & String1 : String2 : String3 & Point3
```

```
[4'15]  rmoptionals from Box1/Box2 = Box1/Newbox2
        ------------------------------------------
        fitshor Box1/Box2 in Frame        = fitshor Box1 Newbox2 in Frame

[4'16]  fitshor Box1 ; Box2 in Frame      = false & "" & (0,0)

 [5'1]  vertical String                   = String
 [5'2]  vertical [Box]                    = [Box]
 [5'3]  vertical C-list Box               = C-list Box
 [5'4]  vertical Box1/ Box2               = Box1; vertical Box2
 [5'5]  vertical Box1 Box2                = Box1 Box2
 [5'6]  vertical Box1; Box2               = Box1; Box2

 [6'1]  is-duobox String                  = false
 [6'2]  is-duobox [Box]                   = false
 [6'3]  is-duobox C-list Box              = false
 [6'4]  is-duobox Box1/ Box2              = true
 [6'5]  is-duobox Box1 Box2               = true
 [6'6]  is-duobox Box1; Box2              = true
```

Appendix G : Box-interpreter in ASF

```
module Booleans
begin
  exports
  begin
    sorts BOOL
    functions
      true        :                  -> BOOL
      false       :                  -> BOOL
      or          : BOOL # BOOL      -> BOOL   {assoc}
      and         : BOOL # BOOL      -> BOOL   {assoc}
      not         : BOOL             -> BOOL
  end

  variables
      Bool        :-> BOOL

  equations

  [Bool1]  or(true, Bool)            = true
  [Bool2]  or(false,Bool)            = Bool

  [Bool3]  and(true, Bool)           = Bool
  [Bool4]  and(false,Bool)           = false

  [Bool5]  not(true)                 = false
  [Bool6]  not(false)                = true

end Booleans

module Naturals
begin
  exports
  begin
    sorts
         DIGIT, NAT
    functions
         0   :                       -> DIGIT
         1   :                       -> DIGIT
         2   :                       -> DIGIT
         3   :                       -> DIGIT
         4   :                       -> DIGIT
         5   :                       -> DIGIT
         6   :                       -> DIGIT
         7   :                       -> DIGIT
         8   :                       -> DIGIT
         9   :                       -> DIGIT
```

```
      nat  : DIGIT+                           -> NAT
      _+_  : NAT # NAT                        -> NAT    {assoc}
      _-_  : NAT # NAT                        -> NAT    {assoc}
      _*_  : NAT # NAT                        -> NAT    {assoc}
      lt   : NAT # NAT                        -> BOOL
      le   : NAT # NAT                        -> BOOL
      gt   : NAT # NAT                        -> BOOL
      ge   : NAT # NAT                        -> BOOL
      eq   : NAT # NAT                        -> BOOL
  end

imports Booleans
variables
   d1, d2, D, D1, D2, D'                      : -> DIGIT
   y, y1, y2, Y, Y1, Y2, Y'                   : -> DIGIT+
   X1, X2                                     : -> DIGIT*
   n, n1, n2, N, N1, N2                       : -> NAT

equations

  [n1]   nat([0, Y])                        = nat(Y)
  [n3]   nat([0]) + N                       = N
  [n4]   N + nat([0])                       = N
  [n5]   nat([1]) + nat([1])                = nat([2])
  [n6]   nat([1]) + nat([2])                = nat([3])
  [n7]   nat([1]) + nat([3])                = nat([4])
  [n8]   nat([1]) + nat([4])                = nat([5])
  [n9]   nat([1]) + nat([5])                = nat([6])
 [n10]   nat([1]) + nat([6])                = nat([7])
 [n11]   nat([1]) + nat([7])                = nat([8])
 [n12]   nat([1]) + nat([8])                = nat([9])
 [n13]   nat([1]) + nat([9])                = nat([1, 0])
 [n14]   nat([2]) + nat([1])                = nat([3])
 [n15]   nat([2]) + nat([2])                = nat([4])
 [n16]   nat([2]) + nat([3])                = nat([5])
 [n17]   nat([2]) + nat([4])                = nat([6])
 [n18]   nat([2]) + nat([5])                = nat([7])
 [n19]   nat([2]) + nat([6])                = nat([8])
 [n20]   nat([2]) + nat([7])                = nat([9])
 [n21]   nat([2]) + nat([8])                = nat([1, 0])
 [n22]   nat([2]) + nat([9])                = nat([1, 1])
 [n23]   nat([3]) + nat([1])                = nat([4])
 [n24]   nat([3]) + nat([2])                = nat([5])
 [n25]   nat([3]) + nat([3])                = nat([6])
 [n26]   nat([3]) + nat([4])                = nat([7])
 [n27]   nat([3]) + nat([5])                = nat([8])
 [n28]   nat([3]) + nat([6])                = nat([9])
 [n29]   nat([3]) + nat([7])                = nat([1, 0])
 [n30]   nat([3]) + nat([8])                = nat([1, 1])
 [n31]   nat([3]) + nat([9])                = nat([1, 2])
 [n32]   nat([4]) + nat([1])                = nat([5])
 [n33]   nat([4]) + nat([2])                = nat([6])
 [n34]   nat([4]) + nat([3])                = nat([7])
```

```
[n35]  nat([4]) + nat([4])        = nat([8])
[n36]  nat([4]) + nat([5])        = nat([9])
[n37]  nat([4]) + nat([6])        = nat([1, 0])
[n38]  nat([4]) + nat([7])        = nat([1, 1])
[n39]  nat([4]) + nat([8])        = nat([1, 2])
[n40]  nat([4]) + nat([9])        = nat([1, 3])
[n41]  nat([5]) + nat([1])        = nat([6])
[n42]  nat([5]) + nat([2])        = nat([7])
[n43]  nat([5]) + nat([3])        = nat([8])
[n44]  nat([5]) + nat([4])        = nat([9])
[n45]  nat([5]) + nat([5])        = nat([1, 0])
[n46]  nat([5]) + nat([6])        = nat([1, 1])
[n47]  nat([5]) + nat([7])        = nat([1, 2])
[n48]  nat([5]) + nat([8])        = nat([1, 3])
[n49]  nat([5]) + nat([9])        = nat([1, 4])
[n50]  nat([6]) + nat([1])        = nat([7])
[n51]  nat([6]) + nat([2])        = nat([8])
[n52]  nat([6]) + nat([3])        = nat([9])
[n53]  nat([6]) + nat([4])        = nat([1, 0])
[n54]  nat([6]) + nat([5])        = nat([1, 1])
[n55]  nat([6]) + nat([6])        = nat([1, 2])
[n56]  nat([6]) + nat([7])        = nat([1, 3])
[n57]  nat([6]) + nat([8])        = nat([1, 4])
[n58]  nat([6]) + nat([9])        = nat([1, 5])
[n59]  nat([7]) + nat([1])        = nat([8])
[n60]  nat([7]) + nat([2])        = nat([9])
[n61]  nat([7]) + nat([3])        = nat([1, 0])
[n62]  nat([7]) + nat([4])        = nat([1, 1])
[n63]  nat([7]) + nat([5])        = nat([1, 2])
[n64]  nat([7]) + nat([6])        = nat([1, 3])
[n65]  nat([7]) + nat([7])        = nat([1, 4])
[n66]  nat([7]) + nat([8])        = nat([1, 5])
[n67]  nat([7]) + nat([9])        = nat([1, 6])
[n68]  nat([8]) + nat([1])        = nat([9])
[n69]  nat([8]) + nat([2])        = nat([1, 0])
[n70]  nat([8]) + nat([3])        = nat([1, 1])
[n71]  nat([8]) + nat([4])        = nat([1, 2])
[n72]  nat([8]) + nat([5])        = nat([1, 3])
[n73]  nat([8]) + nat([6])        = nat([1, 4])
[n74]  nat([8]) + nat([7])        = nat([1, 5])
[n75]  nat([8]) + nat([8])        = nat([1, 6])
[n76]  nat([8]) + nat([9])        = nat([1, 7])
[n77]  nat([9]) + nat([1])        = nat([1, 0])
[n78]  nat([9]) + nat([2])        = nat([1, 1])
[n79]  nat([9]) + nat([3])        = nat([1, 2])
[n80]  nat([9]) + nat([4])        = nat([1, 3])
[n81]  nat([9]) + nat([5])        = nat([1, 4])
[n82]  nat([9]) + nat([6])        = nat([1, 5])
[n83]  nat([9]) + nat([7])        = nat([1, 6])
[n84]  nat([9]) + nat([8])        = nat([1, 7])
[n85]  nat([9]) + nat([9])        = nat([1, 8])
```

```
[n86]  nat([D1]) + nat([D2]) = nat([D']),
       nat([0, X1]) + nat([0, X2]) = nat(Y')
       =====================================
       nat([X1, D1]) + nat([X2, D2])    = nat([Y', D'])

[n87]  nat([D1]) + nat([D2]) = nat([1, D']),
       nat([0, X1]) + nat([0, X2]) + nat([1]) = nat(Y')
       ================================================
       nat([X1, D1]) + nat([X2, D2])    = nat([Y', D'])


 [k3]  N - N                          = nat([0])
 [k4]  N - nat([0])                   = N
[k14]  nat([2]) - nat([1])            = nat([1])
[k23]  nat([3]) - nat([1])            = nat([2])
[k24]  nat([3]) - nat([2])            = nat([1])
[k32]  nat([4]) - nat([1])            = nat([3])
[k33]  nat([4]) - nat([2])            = nat([2])
[k34]  nat([4]) - nat([3])            = nat([1])
[k41]  nat([5]) - nat([1])            = nat([4])
[k42]  nat([5]) - nat([2])            = nat([3])
[k43]  nat([5]) - nat([3])            = nat([2])
[k44]  nat([5]) - nat([4])            = nat([1])
[k50]  nat([6]) - nat([1])            = nat([5])
[k51]  nat([6]) - nat([2])            = nat([4])
[k52]  nat([6]) - nat([3])            = nat([3])
[k53]  nat([6]) - nat([4])            = nat([2])
[k54]  nat([6]) - nat([5])            = nat([1])
[k59]  nat([7]) - nat([1])            = nat([6])
[k60]  nat([7]) - nat([2])            = nat([5])
[k61]  nat([7]) - nat([3])            = nat([4])
[k62]  nat([7]) - nat([4])            = nat([3])
[k63]  nat([7]) - nat([5])            = nat([2])
[k64]  nat([7]) - nat([6])            = nat([1])
[k68]  nat([8]) - nat([1])            = nat([7])
[k69]  nat([8]) - nat([2])            = nat([6])
[k70]  nat([8]) - nat([3])            = nat([5])
[k71]  nat([8]) - nat([4])            = nat([4])
[k72]  nat([8]) - nat([5])            = nat([3])
[k73]  nat([8]) - nat([6])            = nat([2])
[k74]  nat([8]) - nat([7])            = nat([1])
[k77]  nat([9]) - nat([1])            = nat([8])
[k78]  nat([9]) - nat([2])            = nat([7])
[k79]  nat([9]) - nat([3])            = nat([6])
[k80]  nat([9]) - nat([4])            = nat([5])
[k81]  nat([9]) - nat([5])            = nat([4])
[k82]  nat([9]) - nat([6])            = nat([3])
[k83]  nat([9]) - nat([7])            = nat([2])
[k84]  nat([9]) - nat([8])            = nat([1])
[k88]  nat([1, 0]) - nat([1])         = nat([9])
[k89]  nat([1, 0]) - nat([2])         = nat([8])
[k90]  nat([1, 0]) - nat([3])         = nat([7])
[k91]  nat([1, 0]) - nat([4])         = nat([6])
```

```
[k92]  nat([1, 0]) - nat([5])          = nat([5])
[k93]  nat([1, 0]) - nat([6])          = nat([4])
[k94]  nat([1, 0]) - nat([7])          = nat([3])
[k95]  nat([1, 0]) - nat([8])          = nat([2])
[k96]  nat([1, 0]) - nat([9])          = nat([1])
[k97]  nat([1, D1]) - nat([D2])        = (nat([1, 0]) - nat([D2])) + nat([D1])

[k86]  gt(nat([X1, D1]), nat([X2, D2])) = true,
       ge(nat([D1]), nat([D2])) = true,
       nat([D1]) - nat([D2]) = nat([D']),
       nat([0, X1]) - nat([0, X2]) = nat(Y')
       ====================================
       nat([X1, D1]) - nat([X2, D2])    = nat([Y', D'])

[k87]  gt(nat([X1, D1]), nat([X2, D2])) = true,
       lt(nat([D1]), nat([D2])) = true,
       nat([1, D1]) - nat([D2]) = nat([D']),
       (nat([0, X1]) - nat([1])) - nat([0, X2]) = nat(Y')
       ==================================================
       nat([X1, D1]) - nat([X2, D2])    = nat([Y', D'])

[k88]  le(N1, N2) = true
       ================
       N1 - N2                          = nat([0])

[n88]  nat([0]) * N                     = nat([0])
[n89]  N * nat([0])                     = nat([0])
[n90]  nat([1]) * N                     = N
[n91]  N * nat([1])                     = N
[n92]  nat([2]) * nat([2])              = nat([4])
[n93]  nat([2]) * nat([3])              = nat([6])
[n94]  nat([2]) * nat([4])              = nat([8])
[n95]  nat([2]) * nat([5])              = nat([1, 0])
[n96]  nat([2]) * nat([6])              = nat([1, 2])
[n97]  nat([2]) * nat([7])              = nat([1, 4])
[n98]  nat([2]) * nat([8])              = nat([1, 6])
[n99]  nat([2]) * nat([9])              = nat([1, 8])
[n100] nat([3]) * nat([2])              = nat([6])
[n101] nat([3]) * nat([3])              = nat([9])
[n102] nat([3]) * nat([4])              = nat([1, 2])
[n103] nat([3]) * nat([5])              = nat([1, 5])
[n104] nat([3]) * nat([6])              = nat([1, 8])
[n105] nat([3]) * nat([7])              = nat([2, 1])
[n106] nat([3]) * nat([8])              = nat([2, 4])
[n107] nat([3]) * nat([9])              = nat([2, 7])
[n108] nat([4]) * nat([2])              = nat([8])
[n109] nat([4]) * nat([3])              = nat([1, 2])
[n110] nat([4]) * nat([4])              = nat([1, 6])
[n111] nat([4]) * nat([5])              = nat([2, 0])
[n112] nat([4]) * nat([6])              = nat([2, 4])
[n113] nat([4]) * nat([7])              = nat([2, 8])
[n114] nat([4]) * nat([8])              = nat([3, 2])
[n115] nat([4]) * nat([9])              = nat([3, 6])
```

```
[n116]  nat([5]) * nat([2])            = nat([1, 0])
[n117]  nat([5]) * nat([3])            = nat([1, 5])
[n118]  nat([5]) * nat([4])            = nat([2, 0])
[n119]  nat([5]) * nat([5])            = nat([2, 5])
[n120]  nat([5]) * nat([6])            = nat([3, 0])
[n121]  nat([5]) * nat([7])            = nat([3, 5])
[n122]  nat([5]) * nat([8])            = nat([4, 0])
[n123]  nat([5]) * nat([9])            = nat([4, 5])
[n124]  nat([6]) * nat([2])            = nat([1, 2])
[n125]  nat([6]) * nat([3])            = nat([1, 8])
[n126]  nat([6]) * nat([4])            = nat([2, 4])
[n127]  nat([6]) * nat([5])            = nat([3, 0])
[n128]  nat([6]) * nat([6])            = nat([3, 6])
[n129]  nat([6]) * nat([7])            = nat([4, 2])
[n130]  nat([6]) * nat([8])            = nat([4, 8])
[n131]  nat([6]) * nat([9])            = nat([5, 4])
[n132]  nat([7]) * nat([2])            = nat([1, 4])
[n133]  nat([7]) * nat([3])            = nat([2, 1])
[n134]  nat([7]) * nat([4])            = nat([2, 8])
[n135]  nat([7]) * nat([5])            = nat([3, 5])
[n136]  nat([7]) * nat([6])            = nat([4, 2])
[n137]  nat([7]) * nat([7])            = nat([4, 9])
[n138]  nat([7]) * nat([8])            = nat([5, 6])
[n139]  nat([7]) * nat([9])            = nat([6, 3])
[n140]  nat([8]) * nat([2])            = nat([1, 6])
[n141]  nat([8]) * nat([3])            = nat([2, 4])
[n142]  nat([8]) * nat([4])            = nat([3, 2])
[n143]  nat([8]) * nat([5])            = nat([4, 0])
[n144]  nat([8]) * nat([6])            = nat([4, 8])
[n145]  nat([8]) * nat([7])            = nat([5, 6])
[n146]  nat([8]) * nat([8])            = nat([6, 4])
[n147]  nat([8]) * nat([9])            = nat([7, 2])
[n148]  nat([9]) * nat([2])            = nat([1, 8])
[n149]  nat([9]) * nat([3])            = nat([2, 7])
[n150]  nat([9]) * nat([4])            = nat([3, 6])
[n151]  nat([9]) * nat([5])            = nat([4, 5])
[n152]  nat([9]) * nat([6])            = nat([5, 4])
[n153]  nat([9]) * nat([7])            = nat([6, 3])
[n154]  nat([9]) * nat([8])            = nat([7, 2])
[n155]  nat([9]) * nat([9])            = nat([8, 1])

[n156]  nat(Y) * nat([D2]) = nat(Y')
        ============================
        nat([Y, D1]) * nat([D2])       = nat([Y', 0]) + (nat([D1]) * nat([D2]))

[n157]  nat(Y1) * nat(Y2) = nat(Y')
        ============================
        nat(Y1) * nat([Y2, D])         = nat([Y', 0]) + (nat(Y1) * nat([D]))

   [o1]    lt(nat([0]), nat([1]))      = true
   [o2]    lt(nat([0]), nat([2]))      = true
   [o3]    lt(nat([0]), nat([3]))      = true
   [o4]    lt(nat([0]), nat([4]))      = true
```

```
[o5]    lt(nat([0]), nat([5]))        = true
[o6]    lt(nat([0]), nat([6]))        = true
[o7]    lt(nat([0]), nat([7]))        = true
[o8]    lt(nat([0]), nat([8]))        = true
[o9]    lt(nat([0]), nat([9]))        = true
[o10]   lt(nat([1]), nat([2]))        = true
[o11]   lt(nat([1]), nat([3]))        = true
[o12]   lt(nat([1]), nat([4]))        = true
[o13]   lt(nat([1]), nat([5]))        = true
[o14]   lt(nat([1]), nat([6]))        = true
[o15]   lt(nat([1]), nat([7]))        = true
[o16]   lt(nat([1]), nat([8]))        = true
[o17]   lt(nat([1]), nat([9]))        = true
[o18]   lt(nat([2]), nat([3]))        = true
[o19]   lt(nat([2]), nat([4]))        = true
[o20]   lt(nat([2]), nat([5]))        = true
[o21]   lt(nat([2]), nat([6]))        = true
[o22]   lt(nat([2]), nat([7]))        = true
[o23]   lt(nat([2]), nat([8]))        = true
[o24]   lt(nat([2]), nat([9]))        = true
[o25]   lt(nat([3]), nat([4]))        = true
[o26]   lt(nat([3]), nat([5]))        = true
[o27]   lt(nat([3]), nat([6]))        = true
[o28]   lt(nat([3]), nat([7]))        = true
[o29]   lt(nat([3]), nat([8]))        = true
[o30]   lt(nat([3]), nat([9]))        = true
[o31]   lt(nat([4]), nat([5]))        = true
[o32]   lt(nat([4]), nat([6]))        = true
[o33]   lt(nat([4]), nat([7]))        = true
[o34]   lt(nat([4]), nat([8]))        = true
[o35]   lt(nat([4]), nat([9]))        = true
[o36]   lt(nat([5]), nat([6]))        = true
[o37]   lt(nat([5]), nat([7]))        = true
[o38]   lt(nat([5]), nat([8]))        = true
[o39]   lt(nat([5]), nat([9]))        = true
[o40]   lt(nat([6]), nat([7]))        = true
[o41]   lt(nat([6]), nat([8]))        = true
[o42]   lt(nat([6]), nat([9]))        = true
[o43]   lt(nat([7]), nat([8]))        = true
[o44]   lt(nat([7]), nat([9]))        = true
[o45]   lt(nat([8]), nat([9]))        = true

[o46] nat(y) != nat([0])
      ==================
      lt(nat([d1]), nat([y, d2]))    = true

[o47] nat(y) = nat([0]),
      lt(nat([d1]), nat([d2])) = true
      ==============================
      lt(nat([y, d1]), nat([d2]))    = true
```

```
[o48]  nat(y1) = nat(y2),
       lt(nat([d1]), nat([d2])) = true
       ================================
       lt(nat([y1, d1]), nat([y2, d2])) = true

[o49]  nat(y1) != nat(y2),
       lt(nat(y1), nat(y2)) = true
       ============================
       lt(nat([y1, d1]), nat([y2, d2])) = true

[o50]  eq(n1, n2) = true
       =================
       lt(n1, n2)                          = false

[o50]  lt(n2, n1) = true
       =================
       lt(n1, n2)                          = false

[o50]  le(n, n)                            = true

[o51]  lt(n1, n2) = true
       =================
       le(n1, n2)                          = true

[o51]  lt(n2, n1) = true
       =================
       le(n1, n2)                          = false

[o52]  gt(n1, n2)                          = lt(n2, n1)

[o53]  ge(n1, n2)                          = le(n2, n1)

[o54]  nat([d1]) = nat([d2])
       =====================
       eq(nat([d1]), nat([d2]))            = true

[o55]  nat(y1) = nat([0]),
       nat([d1]) = nat([d2])
       =====================
       eq(nat([y1, d1]), nat([d2]))        = true

[o56]  nat(y2) = nat([0]),
       nat([d1]) = nat([d2])
       =====================
       eq(nat([d1]), nat([y2, d2]))        = true

[o57]  eq(nat(y1), nat(y2)) = true,
       nat([d1]) = nat([d2])
       ============================
       eq(nat([y1, d1]), nat([y2, d2]))        = true
```

```
[o54]  nat([d1]) != nat([d2])
       ======================
       eq(nat([d1]), nat([d2]))          = false

[o55]  nat(y1) = nat([0]),
       nat([d1]) != nat([d2])
       ======================
       eq(nat([y1, d1]), nat([d2]))      = false

[o56]  nat(y2) = nat([0]),
       nat([d1]) != nat([d2])
       ======================
       eq(nat([d1]), nat([y2, d2]))      = false

[o55]  nat(y1) != nat([0])
       ===================
       eq(nat([y1, d1]), nat([d2]))      = false

[o56]  nat(y2) != nat([0])
       ===================
       eq(nat([d1]), nat([y2, d2]))      = false

[o57]  nat([d1]) != nat([d2])
       ======================
       eq(nat([y1, d1]), nat([y2, d2]))        = false

[o57]  nat([d1]) = nat([d2]),
       eq(nat(y1), nat(y2)) = false
       ============================
       eq(nat([y1, d1]), nat([y2, d2]))        = false


end Naturals


module Chars
begin
  exports
  begin
    sorts CHAR
    functions
      a:             -> CHAR
      b:             -> CHAR
      c:             -> CHAR
      d:             -> CHAR
      e:             -> CHAR
      f:             -> CHAR
      g:             -> CHAR
      h:             -> CHAR
      i:             -> CHAR
      j:             -> CHAR
      k:             -> CHAR
      l:             -> CHAR
```

```
      m:              -> CHAR
      n:              -> CHAR
      o:              -> CHAR
      p:              -> CHAR
      q:              -> CHAR
      r:              -> CHAR
      s:              -> CHAR
      t:              -> CHAR
      u:              -> CHAR
      v:              -> CHAR
      w:              -> CHAR
      z:              -> CHAR
      x:              -> CHAR
      y:              -> CHAR
      z:              -> CHAR
   end
end Chars

module Characters
begin
   exports
     begin
        sorts CHARS
        functions
          ch          : CHAR*                 -> CHARS
          null        :                       -> CHARS
          layoutchar  :                       -> CHARS
          depthchar   :                       -> CHARS
          length      : CHARS                 -> NAT
          split       : CHARS # NAT           -> CHARS # CHARS
          append      : CHARS # CHARS         -> CHARS
     end
   imports Naturals, Chars

   functions
        splithelp     : CHARS # CHARS # NAT -> CHARS # CHARS

   variables
          Char        :                       -> CHAR
          X           :                       -> CHAR+
          Y, Z        :                       -> CHAR*
          Chars, Chars1, Chars2 :             -> CHARS
          Int         :                       -> NAT

   equations

   [C1'1]   layoutchar                        = ch([b])
   [C2'1]   null                              = ch([])
   [C3'1]   depthchar                         = ch([p])
   [C4'1]   length(ch([]))                    = nat([0])
   [C4'2]   length(ch([Char, Y]))             = length(ch([Y])) + nat([1])
   [C5'1]   split(Chars, Int)                 = <ch([]), Chars>
                                                    when lt(Int, nat([0])) = true
```

```
  [C5'2]  split(Chars, Int)              = splithelp(ch([]), Chars, Int)
                                                 when ge(Int, nat([0])) = true
  [C6'1]  splithelp(Chars1, Chars2, Int)  = <Chars1, Chars2>
                                                 when le(Int, nat([0])) = true
  [C6'2]  splithelp(ch([Y]), ch([Char,Z]), Int) =
                                       splithelp(ch([Y,Char]), ch([Z]),  Int - nat([1]))
  [C6'3]  splithelp(Chars, ch([]), Int)   = <Chars, ch([])>
                                                 when gt(Int, nat([0])) = true
  [C7'1]  append(ch([Y]), ch([Z]))        = ch([Y,Z])
end Characters


module Strings
begin
  exports
    begin
      sorts STRING
      functions
        ~_              : CHARS                  -> STRING
        _-_             : STRING # STRING        -> STRING
        _$_             : STRING # STRING        -> STRING
        length          : STRING                 -> NAT
        split           : STRING # NAT           -> STRING # STRING
        withnewline     : STRING                 -> BOOL
        layoutstr       :                        -> STRING
        layoutstring    : NAT                    -> STRING
        newline         :                        -> STRING
        newlinestring   : NAT                    -> STRING
    end
  imports Characters
  variables
       Chars, Chars1, Chars2                : -> CHARS
       String1, String2, String3            : -> STRING
       Int                                  : -> NAT
  equations

  [S1'1]  ~Chars1 - ~Chars2               = ~append(Chars1, Chars2)
  [S1'2]  String1 - (String2 $ String3)   = (String1 - String2) $ String3
  [S1'3]  (String1 $ String2) - String3   = String1 $ (String2 - String3)
  [S2'1]  length(~Chars)                  = length(Chars)
  [S2'2]  length(String1 $ String2)       = length(String1) + length(String2)
  [S3'1]  split(~Chars, Int)              = <~Chars1, ~Chars2>
                                                when split(Chars, Int) =
                                                             <Chars1, Chars2>

  [S3'2]  split(String1 $ String2, Int)   = split(String1 - String2, Int)
  [S4'1]  withnewline(~Chars)             = false
  [S4'2]  withnewline(String1 $ String2)  = true
  [S5'1]  newline                         = newlinestring(nat([1]))
  [S6'1]  newlinestring(Int)              = ~null when le(Int, nat([0])) = true
  [S6'2]  newlinestring(Int)              = ~null $ newlinestring(Int - nat([1]))
                                                when gt(Int, nat([0])) = true
  [S7'1]  layoutstr                       = ~layoutchar
  [S8'1]  layoutstring(Int)               = ~null when le(Int, nat([0])) = true
```

```
  [S8'2]  layoutstring(Int)                   = layoutstr - layoutstring(Int - nat([1]))
                                                  when gt(Int, nat([0])) = true
 end Strings

 module Box-syntax
 begin
   exports
     begin
       sorts SYMBOL, OPER, CHANGE, CHANGELIST, BOX
       functions
         width      :                           -> SYMBOL
         height     :                           -> SYMBOL
         reltab     :                           -> SYMBOL
         abstab     :                           -> SYMBOL
         leftm      :                           -> SYMBOL
         rightm     :                           -> SYMBOL
         relleftm   :                           -> SYMBOL
         absdepth   :                           -> SYMBOL
         reldepth   :                           -> SYMBOL
         pl         :                           -> OPER
         as         :                           -> OPER
         st         :                           -> OPER
         b          : STRING                    -> BOX
         c          : SYMBOL # NAT              -> CHANGE
         c          : SYMBOL                    -> CHANGE
         l          : CHANGE                    -> CHANGELIST
         _/_        : CHANGE # CHANGELIST       -> CHANGELIST
         b          : CHANGELIST # BOX          -> BOX
         b          : BOX # OPER # BOX          -> BOX
         brack      : BOX                       -> BOX
         defaults   :                           -> CHANGELIST
     end
   imports Strings
   equations

 [BS1]  defaults     = c(width, nat([1]))/(c(height, nat([1]))/(c(reltab,nat([4]))
                         /(c(abstab, nat([8]))/(c(leftm, nat([1]))/(c(rightm, nat([3, 0]))
                         /(c(absdepth, nat([1, 0, 0]))/l(c(reldepth, nat([1]))))))))))
 end Box-syntax


 module Changelists
 begin
   exports
     begin
       functions
         eq           : SYMBOL # SYMBOL       -> BOOL
         is-in-change : SYMBOL # CHANGE       -> BOOL
         is-in-list   : SYMBOL # CHANGELIST   -> BOOL
         changelist   : BOX                   -> CHANGELIST
         value        : SYMBOL # CHANGELIST   -> NAT
         rmfrombox    : SYMBOL # BOX          -> BOX
         rmfromlist   : SYMBOL # CHANGELIST   -> CHANGELIST
```

```
        changetab     : BOX                    -> BOX
        makesep       : SYMBOL # NAT           -> STRING
        sepbefor      : SYMBOL # BOX           -> STRING # BOX
        rmoptionals   : BOX                    -> BOX
    end
imports Box-syntax
variables
      Symbol, Symbol1, Symbol2                : -> SYMBOL
      String                                  : -> STRING
      Change, Change1, Change2                : -> CHANGE
      Changelist, Newc-list, C-list2          : -> CHANGELIST
      Int, Int1, Int2                         : -> NAT
      Box, Box1, Box2                         : -> BOX

equations

 [C11'1]  eq(width, width)                  = true
 [C11'2]  eq(height, height)                = true
 [C11'3]  eq(reltab, reltab)                = true
 [C11'4]  eq(abstab, abstab)                = true
 [C11'5]  eq(leftm, leftm)                  = true
 [C11'6]  eq(rightm, rightm)                = true
 [C11'7]  eq(absdepth, absdepth)            = true
 [C11'8]  eq(reldepth, reldepth)            = true
 [C11'9]  eq(width, height)                 = false
[C11'10]  eq(width, reltab)                 = false
[C11'11]  eq(width, abstab)                 = false
[C11'12]  eq(width, leftm)                  = false
[C11'13]  eq(width, rightm)                 = false
[C11'14]  eq(width, absdepth)               = false
[C11'15]  eq(width, reldepth)               = false
[C11'16]  eq(height, width)                 = false
[C11'17]  eq(height, reltab)                = false
[C11'18]  eq(height, abstab)                = false
[C11'19]  eq(height, leftm)                 = false
[C11'20]  eq(height, rightm)                = false
[C11'21]  eq(height, absdepth)              = false
[C11'22]  eq(height, reldepth)              = false
[C11'23]  eq(reltab, width)                 = false
[C11'24]  eq(reltab, height)                = false
[C11'25]  eq(reltab, abstab)                = false
[C11'26]  eq(reltab, leftm)                 = false
[C11'27]  eq(reltab, rightm)                = false
[C11'28]  eq(reltab, absdepth)              = false
[C11'29]  eq(reltab, reldepth)              = false
[C11'30]  eq(abstab, width)                 = false
[C11'31]  eq(abstab, height)                = false
[C11'32]  eq(abstab, reltab)                = false
[C11'33]  eq(abstab, leftm)                 = false
[C11'34]  eq(abstab, rightm)                = false
[C11'35]  eq(abstab, absdepth)              = false
[C11'36]  eq(abstab, reldepth)              = false
[C11'37]  eq(leftm, width)                  = false
```

```
[C11'38]  eq(leftm, height)                    = false
[C11'39]  eq(leftm, reltab)                    = false
[C11'40]  eq(leftm, abstab)                    = false
[C11'41]  eq(leftm, rightm)                    = false
[C11'42]  eq(leftm, absdepth)                  = false
[C11'43]  eq(leftm, reldepth)                  = false
[C11'44]  eq(rightm, width)                    = false
[C11'45]  eq(rightm, height)                   = false
[C11'46]  eq(rightm, reltab)                   = false
[C11'47]  eq(rightm, abstab)                   = false
[C11'48]  eq(rightm, leftm)                    = false
[C11'49]  eq(rightm, absdepth)                 = false
[C11'50]  eq(rightm, reldepth)                 = false
[C11'51]  eq(absdepth, width)                  = false
[C11'52]  eq(absdepth, height)                 = false
[C11'53]  eq(absdepth, reltab)                 = false
[C11'54]  eq(absdepth, abstab)                 = false
[C11'55]  eq(absdepth, leftm)                  = false
[C11'56]  eq(absdepth, rightm)                 = false
[C11'57]  eq(absdepth, reldepth)               = false
[C11'58]  eq(reldepth, width)                  = false
[C11'59]  eq(reldepth, height)                 = false
[C11'60]  eq(reldepth, reltab)                 = false
[C11'61]  eq(reldepth, abstab)                 = false
[C11'62]  eq(reldepth, leftm)                  = false
[C11'63]  eq(reldepth, rightm)                 = false
[C11'64]  eq(reldepth, absdepth)               = false
[C12'1]  is-in-change(Symbol1, c(Symbol2, Int))  = eq(Symbol1, Symbol2)
[C12'2]  is-in-change(Symbol1, c(Symbol2))       = eq(Symbol1, Symbol2)
[C13'1]  is-in-list(Symbol, Change/Changelist)   = true
                                 when is-in-change(Symbol, Change) = true
[C13'2]  is-in-list(Symbol, Change/Changelist)   = is-in-list(Symbol, Changelist)
                                 when is-in-change(Symbol, Change) = false
[C13'3]  is-in-list(Symbol, l(Change))           = is-in-change(Symbol, Change)
[C14'1]  value(Symbol1, c(Symbol2, Int)/Changelist) = Int
                                 when eq(Symbol1, Symbol2) = true
[C14'2]  value(Symbol1, c(Symbol2)/Changelist)   = value(Symbol1, defaults)
                                 when eq(Symbol1, Symbol2) = true
[C14'3]  value(Symbol1, Change/Changelist)       = value(Symbol1, Changelist)
                                 when is-in-change(Symbol1,Change) = false
[C14'4]  value(Symbol1, l(c(Symbol2, Int)))      = Int
                                 when eq(Symbol1, Symbol2) = true
[C14'5]  value(Symbol1, l(c(Symbol2, Int)))      = nat([0])
                                 when eq(Symbol1, Symbol2) = false
[C14'6]  value(Symbol1, l(c(Symbol2)))           = value(Symbol1, defaults)
                                 when eq(Symbol1, Symbol2) = true
[C14'7]  value(Symbol1, l(c(Symbol2)))           = nat([0])
                                 when eq(Symbol1, Symbol2) = false
[C15'1]  changelist(b(String))           = l(c(width, nat([1])))
[C15'2]  changelist(b(Changelist, Box))  = Changelist
[C15'3]  changelist(b(Box1,pl,Box2))     = changelist(Box1)
[C15'4]  changelist(b(Box1,as,Box2))     = changelist(Box1)
[C15'5]  changelist(b(Box1,st,Box2))     = changelist(Box1)
```

```
[C15'6]  changelist(brack(Box))          = changelist(Box)
[C16'1]  rmfromlist(Symbol, Change1/(Change2/Changelist))  = Change2/Changelist
                              when is-in-change(Symbol, Change1) = true
[C16'2]  rmfromlist(Symbol, Change1/(Change2/Changelist))   =
                             Change1/rmfromlist(Symbol, Change2/Changelist)
                              when is-in-change(Symbol, Change) = false
[C16'3]  rmfromlist(Symbol, Change1/l(Change2)) = l(Change2)
                              when is-in-change(Symbol, Change1) = true,
                                   is-in-change(Symbol, Change2) = false
[C16'4]  rmfromlist(Symbol, Change1/l(Change2)) = l(Change1)
                              when is-in-change(Symbol, Change2) = true,
                                   is-in-change(Symbol, Change1) = false
[C16'5]  rmfromlist(Symbol, Change1/l(Change2)) = Change1/l(Change2)
                              when is-in-change(Symbol, Change1) = false,
                                   is-in-change(Symbol, Change2) = false
[C16'6]  rmfromlist(Symbol, Change1/l(Change2)) = l(Change2)
                              when is-in-change(Symbol, Change1) = true,
                                   is-in-change(Symbol, Change2) = true
[C16'7]  rmfromlist(Symbol, l(Change)) = l(Change)
                              when is-in-change(Symbol, Change) = false
[C16'8]  rmfromlist(Symbol, l(Change)) = l(c(width, nat([1])))
                              when is-in-change(Symbol, Change) = true
[C17'1]  rmfrombox(Symbol, b(String))           = b(String)
[C17'2]  rmfrombox(Symbol, b(l(Change), Box))     = Box
                              when is-in-change(Symbol, Change) = true
[C17'3]  rmfrombox(Symbol, b(l(Change), Box))     = b(l(Change), Box)
                              when is-in-change(Symbol, Change) = false
[C17'4]  rmfrombox(Symbol, b(Change/Changelist, Box)) =
                                   b(rmfromlist(Symbol, Change/Changelist), Box)
[C17'5]  rmfrombox(Symbol, b(Box1,pl,Box2))       =
                                  b(rmfrombox(Symbol, Box1),pl,Box2)
[C17'6]  rmfrombox(Symbol, b(Box1,as,Box2))       =
                                  b(rmfrombox(Symbol, Box1),as,Box2)
[C17'7]  rmfrombox(Symbol, b(Box1,st,Box2))       =
                                  b(rmfrombox(Symbol, Box1),st,Box2)
[C17'8]  rmfrombox(Symbol, brack(Box))           = brack(Box)
[C18'1]  changetab(b(Changelist, Box))     = b(Changelist, Box)
                              when is-in-list(reltab, Changelist) = false
[C18'2]  changetab(b(Changelist, Box))     =  b(c(reltab, Int2 - Int1)/Newc-list, Box)
                              when is-in-list(reltab, Changelist) = true,
                                   is-in-list(width, Changelist) = true,
                                   value(reltab, Changelist) = Int2,
                                   value(width, Changelist) = Int1,
                                   rmfromlist(reltab, Changelist) = Newc-list
[C18'3]  changetab(b(Changelist, Box))     = b(c(reltab, Int2 - Int1)/Newc-list, Box)
                              when is-in-list(reltab, Changelist) = true,
                                   is-in-list(width, Changelist) = false,
                                   value(reltab, Changelist) = Int2,
                                   value(width, defaults) = Int1,
                                   rmfromlist(reltab, Changelist) = Newc-list
[C18'4]  changetab(b(String))              = b(String)
[C18'5]  changetab(brack(Box))             = brack(Box)
[C18'6]  changetab(b(Box1,pl,Box2))        = b(Box1,pl,Box2)
```

```
 [C18'7]  changetab(b(Box1,as,Box2))        = b(Box1,as,Box2)
 [C18'8]  changetab(b(Box1,st,Box2))        = b(Box1,st,Box2)
 [C19'1]  makesep(width, Int)               = layoutstring(Int)
 [C19'2]  makesep(height, Int)              = newlinestring(Int)
 [C19'3]  makesep(Symbol, Int)              = ~null
                                   when eq(Symbol, width) = false,
                                        eq(Symbol, height) = false
[C110'1]  sepbefore(Symbol, Box)            =
                               <makesep(Symbol, value(Symbol, changelist(Box))),
                                                         rmfrombox(Symbol, Box)>
                                   when is-in-list(Symbol, changelist(Box)) = true
[C110'2]  sepbefore(Symbol, Box)            =
                                      <makesep(Symbol, value(Symbol, defaults)), Box>
                                   when is-in-list(Symbol, changelist(Box)) = false

[C111'1]  rmoptionals(b(String))            = b(String)
[C111'2]  rmoptionals(brack(Box))           = brack(Box)
[C111'3]  rmoptionals(b(l(Change), Box))    = Box
                                   when or(is-in-change(reltab, Change),
                                           is-in-change(height, Change)) = true
[C111'4]  rmoptionals(b(l(Change), Box))    = b(l(Change), Box)
                                   when or(is-in-change(reltab, Change),
                                           is-in-change(height, Change)) = false
[C111'5]  rmoptionals(b(Change/Changelist, Box)) = b(Newc-list, Box)
                                   when rmfromlist(reltab, Change/Changelist) = C-list2,
                                        rmfromlist(height, C-list2) = Newc-list
[C111'6]  rmoptionals(b(Box1,pl,Box2))      = b(rmoptionals(Box1),pl,Box2)
[C111'7]  rmoptionals(b(Box1,as,Box2))      = b(Box1,as,rmoptionals(Box2))
[C111'8]  rmoptionals(b(Box1,st,Box2))      = b(Box1,st,Box2)

end Changelists


module Frame
begin
  exports
    begin
      sorts POINT, FRAME
      functions
        p          : NAT # NAT                    -> POINT
        f          : POINT # NAT # NAT # NAT      -> FRAME
        start      : FRAME                        -> POINT
        xpoint     : FRAME                        -> NAT
        ypoint     : FRAME                        -> NAT
        leftm      : FRAME                        -> NAT
        rightm     : FRAME                        -> NAT
        depth      : FRAME                        -> NAT
        newstart   : POINT # FRAME                -> FRAME
        newleftm   : NAT # FRAME                  -> FRAME
        newrightm  : NAT # FRAME                  -> FRAME
        newdepth   : NAT # FRAME                  -> FRAME
        fromto     : POINT # POINT                -> STRING
        makeframe  : CHANGELIST                   -> FRAME
```

```
      apply       : CHANGELIST # FRAME                     -> FRAME
      apply       : CHANGE # FRAME                         -> FRAME
      modify      : CHANGELIST # FRAME                     -> FRAME
      modify      : CHANGE # FRAME                         -> FRAME
      same-line   : POINT # FRAME                          -> BOOL
   end
imports Changelists
variables
    X, X1, X2, Y, Y1, Y2, Lm, Lm2, Rm, Rm2, D, D2,
    Width, Tabline, Spaces, Lines, Indent, Tab, Int     : -> NAT
    Point, Point2                                       : -> POINT
    Frame                                               : -> FRAME
    Change                                              : -> CHANGE
    Changelist                                          : -> CHANGELIST
    Symbol                                              : -> SYMBOL

equations

 [F1'1]   start(f(Point, Lm, Rm, D))               = Point
 [F2'1]   xpoint(f(p(X,Y), Lm, Rm, D))             = X
 [F3'1]   ypoint(f(p(X,Y), Lm, Rm, D))             = Y
 [F4'1]   leftm(f(Point, Lm, Rm, D))               = Lm
 [F5'1]   rightm(f(Point, Lm, Rm, D))              = Rm
 [F6'1]   depth(f(Point, Lm, Rm, D))               = D
 [F7'1]   newstart(Point2, f(Point, Lm, Rm, D))    = f(Point2, Lm, Rm, D)
 [F8'1]   newleftm(Lm2, f(Point, Lm, Rm, D))       = f(Point, Lm2, Rm, D)
 [F9'1]   newrightm(Rm2, f(Point, Lm, Rm, D))      = f(Point, Lm, Rm2, D)
[F10'1]   newdepth(D2, f(Point, Lm, Rm, D))        = f(Point, Lm, Rm, D2)
[F11'1]   fromto(p(X1,Y1), p(X2,Y2))               = layoutstring(X2 - X1)
                                                          when eq(Y1, Y2) = true,
                                                               ge(X2, X1) = true

[F11'2]   fromto(p(X1,Y1), p(X2,Y2))               = (newlinestring(Y2 - Y1) -
                                                     fromto(p(value(leftm, defaults),Y2),
                                                           p(X2, Y2)))
                                                          when gt(Y2, Y1) = true
[F12'1]   makeframe(Changelist)                    = apply(Changelist,
                                                        f(p(nat([1]),nat([1])), nat([1]),
                                                           nat([8,0]), nat([1])))

[F13'1]   apply(Change/Changelist, Frame)          = apply(Changelist,
                                                             apply(Change, Frame))
[F13'2]   apply(l(Change), Frame)                  = apply(Change, Frame)
[F14'1]   apply(c(width, Int), Frame)              = Frame
[F14'2]   apply(c(height, Int), Frame)             = Frame
[F14'3]   apply(c(reltab, Int), Frame)             = Frame
[F14'4]   apply(c(abstab, Int), Frame)             = Frame
[F14'5]   apply(c(relleftm, Int), Frame)           = Frame
[F14'6]   apply(c(leftm, Lm2), f(p(X,Y), Lm, Rm, D)) = f(p(Lm2,Y), Lm2, Rm, D)
                                 when le(Lm2, Rm) = true
[F14'7]   apply(c(leftm, Lm2), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm, D)
                                                      when gt(Lm2, Rm) = true
[F14'8]   apply(c(rightm, Rm2), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm2, D)
                                                      when ge(Rm2, Lm) = true
```

```
[F14'9]   apply(c(rightm, Rm2), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm, D)
                                                    when lt(Rm2, Lm) = true
[F14'10]  apply(c(absdepth, D2), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm, D2)
                                                    when ge(D2, nat([0])) = true
[F14'11]  apply(c(absdepth, D2), Frame)        = Frame when lt(D2, nat([0])) = true
[F14'12]  apply(c(reldepth, D2), Frame)        = Frame
[F15'1]   modify(Change/Changelist, Frame)     = modify(Changelist,
                                                      modify(Change, Frame))
[F15'2]   modify(l(Change), Frame)             = modify(Change, Frame)
[F16'1]   modify(c(Symbol), Frame)             =
                                  modify(c(Symbol,value(Symbol, defaults)),
                                       Frame)
[F16'2]   modify(c(width, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X + Int,Y), Lm, Rm, D)
                            when ge(Int, nat([0])) = true,
                                 le(X + Int, Rm) = true
[F16'3]   modify(c(width, Int), Frame)      = Frame when lt(Int, nat([0])) = true
[F16'4]   modify(c(width, Int), f(p(X,Y), Lm, Rm, D)) =
                        f(p(Lm + (Width - Rm) - nat([1]), Y + nat([1])), Lm, Rm, D)
                            when X + Int = Width,
                                 gt(Width, Rm) = true
[F16'5]   modify(c(reltab, Int), f(p(X,Y), Lm, Rm, D)) = f(p(Tabline,Y), Tabline, Rm, D)
                            when gt(Int, nat([0])) = true,
                                 X + Int = Tabline,
                                 le(Tabline, Rm) = true
[F16'6]   modify(c(reltab, Int), Frame)     = Frame
                            when le(Int, nat([0])) = true
[F16'7]   modify(c(reltab, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Lm, Rm, D)
                            when ge(X + Int, Rm) = true
[F16'8]   modify(c(abstab, Int), f(p(X,Y), Lm, Rm, D)) =
                        f(p(Lm + Int - nat([1]),Y), Int, Rm, D)
                            when ge(Lm + Int - nat([1]), X) = true,
                                       le(Lm + Int - nat([1]), Rm) = true
[F16'9]   modify(c(abstab, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Lm, Rm, D)
                            when lt(Lm + Int - nat([1]), X) = true
[F16'10]  modify(c(abstab, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Lm, Rm, D)
                            when ge(Lm + Int - nat([1]), X) = true,
                                 gt(Lm + Int - nat([1]), Rm) = true
[F16'11]  modify(c(height, Int), f(p(X,Y), Lm, Rm, D)) = f(p(Lm,Y + Int), Lm, Rm, D)
                            when gt(Int, nat([0])) = true
[F16'12]  modify(c(height, Int), Frame)     = Frame
                            when le(Int, nat([0])) = true
[F16'13]  modify(c(leftm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Int, Rm, D)
                            when ge(Int, value(leftm, defaults)) = true,
                                 le(Int, Rm) = true, ge(X, Int) = true
[F16'14]  modify(c(leftm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(Int,Y), Int, Rm, D)
                            when ge(Int, value(leftm, defaults)) = true,
                                 le(Int, Rm) = true, lt(X, Int) = true
[F16'15]  modify(c(leftm, Int), Frame)      = Frame
                            when lt(Int, value(leftm, defaults)) = true
[F16'16]  modify(c(leftm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Lm, Rm, D)
                            when ge(Int, value(leftm, defaults)) = true,
                                 gt(Int, Rm) = true
```

```
[F16'17]  modify(c(rightm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Lm, Int, D)
                                   when le(Int, value(rightm, defaults)) = true,
                                        ge(Int, Lm) = true, le(X, Int) = true
[F16'18]  modify(c(rightm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(Lm,Y + nat([1])), Lm, Rm, D)
                                   when le(Int, value(rightm, defaults)) = true,
                                        ge(Int, Lm) = true, gt(X, Int) = true
[F16'19]  modify(c(rightm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), Lm, Rm, D)
                                   when le(Int, value(rightm, defaults)) = true,
                                        lt(Int, Lm) = true
[F16'20]  modify(c(rightm, Int), Frame)     = Frame
                                   when gt(Int, value(rightm, defaults)) = true
[F16'21]  modify(c(relleftm, Int), f(p(X,Y), Lm, Rm, D)) = f(p(X,Y), X + Int, Rm, D)
[F16'22]  modify(c(absdepth, Int), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm, Int)
                                   when ge(Int, nat([0])) = true
[F16'23]  modify(c(absdepth, Int), Frame)  = Frame
                                   when lt(Int, nat([0])) = true
[F16'24]  modify(c(reldepth, Int), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm, D + Int)
                                   when ge(D + Int, nat([0])) = true
[F16'25]  modify(c(reldepth, Int), f(Point, Lm, Rm, D)) = f(Point, Lm, Rm, nat([0]))
                                   when lt(D + Int, nat([0])) = true
[F17'1]   same-line(p(X1,Y1), f(p(X2,Y2), Lm, Rm, D)) = or(eq(Y1, Y2),
                                                  and(eq(Y1, Y2 + nat([1])),
                                                      eq(X1, Lm)))

end Frame


module BoxtoString
begin
  exports
    begin
      sorts STRING-END, RESULT
      functions
        makestring : BOX                                    -> STRING
        t          : STRING # POINT                         -> STRING-END
        work       : BOX # FRAME                            -> STRING-END
        string     : STRING-END                             -> STRING
        r          : BOOL # STRING # POINT                  -> RESULT
        fitshor    : BOX # FRAME                            -> RESULT
        vertical   : BOX                                    -> BOX
        is-duobox  : BOX                                    -> BOOLEAN
    end

  imports Frame
  variables
      Int, NewD, Length, Nextline, X1, Y1, X2, Y2, X3, Y3, X4,
      Y4, D                                                   :-> NAT
      Atthisline, Atnextline, String, String1, String2,
      String3, String4, String5, String6, Sep, Newline        :-> STRING
      Frame, Newframe                                         :-> FRAME
      Point, Point1, Point2, Point3, Point4, Point5, Point6   :-> POINT
      Box, Box1, Box2, Box3, Nbox1, Nbox2, Newbox1, Newbox2   :-> BOX
      Change                                                  :-> CHANGE
      Changelist, C-list, C-list2, Newc-list                  :-> CHANGELIST
```

```
equations

[B1'1]  makestring(Box)                         = fromto(p(nat([1]),nat([1])),
                                                    start(Frame)) -
                                                  string(work(Box, Frame))
                     when makeframe(defaults) = Frame
[B2'1]  string(t(String, Point))          = String

[B3'1]  work(brack(Box), Frame)           = work(b(~depthchar), Frame)
                     when eq(depth(Frame), nat([1])) = true
[B3'2]  work(brack(Box), Frame)           = work(Box, Newframe)
                     when gt(depth(Frame),nat([1])) = true,
                          is-duobox(Box) = true,
                          NewD = depth(Frame) - nat([1]) ,
                          newdepth(NewD, newleftm(xpoint(Frame), Frame)) = Newframe
[B3'3]  work(brack(Box), Frame)           = work(Box, Newframe)
                     when gt(depth(Frame),nat([1])) = true,
                          is-duobox(Box) = false,
                          NewD = depth(Frame) - nat([1]),
                          newdepth(NewD, Frame) = Newframe
[B3'4]  work(b(String), Frame)            =
                              t(String,p(xpoint(Frame)+Length, ypoint(Frame)))
                     when withnewline(String) = false,
                          Length = length(String),
                          le(Length, rightm(Frame)-xpoint(Frame)) = true
[B3'5]  work(b(String), Frame)            =
                t(String - fromto(p(rightm(Frame), ypoint(Frame)),
                                   p(leftm(Frame),ypoint(Frame)+nat([1]))),
                   p(leftm(Frame), ypoint(Frame)+nat([1])))
                     when withnewline(String) = false,
                          Length = length(String),
                          eq(Length, rightm(Frame)- xpoint(Frame) + nat([1])) = true
[B3'6]  work(b(String), Frame)            =
                        t(fromto(start(Frame),p(leftm(Frame), Nextline))-String,
                             p(leftm(Frame) + Length, Nextline))
                     when withnewline(String) = false,
                          Length = length(String),
                          Nextline = ypoint(Frame) + nat([1]),
                          gt(Length, rightm(Frame) - xpoint(Frame) + nat([1])) = true,
                          le(Length, rightm(Frame) - leftm(Frame) + nat([1])) = true
[B3'7]  work(b(String), Frame)            = t(Atthisline - Newline - String2, Point2)
                     when withnewline(String) = false,
                          gt(length(String),
                                rightm(Frame) - leftm(Frame) + nat([1])) = true,
                          split(String, rightm(Frame) - xpoint(Frame) + nat([1])) =
                                                           <Atthisline, Atnextline>,
                          fromto(p(rightm(Frame),ypoint(Frame)),
                                p(leftm(Frame), ypoint(Frame)+nat([1]))) = Newline,
                          work(b(Atnextline), newstart(p(leftm(Frame),
                                                     ypoint(Frame)+nat([1])),
                                                  Frame)) = t(String2, Point2)
```

```
[B3'8]   work(b(String1 $ String2), Frame)       = t(String3 - Newline - String4, Point4)
                  when work(b(String1), Frame) = t(String3, p(X3, Y3)),
                       eq(X3, leftm(Frame)) = false,
                       fromto(p(rightm(Frame),Y3),
                              p(leftm(Frame),Y3 + nat([1]))) = Newline,
                       work(b(String2), newstart(p(leftm(Frame),
                                                   Y3 + nat([1])), Frame)) =
                                                              t(String4, Point4)
[B3'9]   work(b(String1 $ String2), Frame) = t(String3 - String4, Point4)
                  when work(b(String1), Frame) = t(String3, p(X3,Y3)),
                       eq(X3, leftm(Frame)) = true,
                       work(b(String2), newstart(p(X3, Y3), Frame)) =
                                                              t(String4, Point4)
[B3'10]  work(b(C-list, Box), Frame)       = t(String1 - String2, Point2)
                  when modify(C-list, Frame) = Newframe,
                       fromto(start(Frame), start(Newframe)) = String1,
                       work(Box, Newframe) = t(String2,Point2)
[B3'11]  work(b(Box1,pl,Box2), Frame)      = t(String1 - String2 - String3, Point3)
                  when work(Box1, Frame) = t(String1, Point1),
                       changetab(Box2) = Nbox2,
                       sepbefore(width, Nbox2) = <Sep, Newbox2>,
                       work(b(Sep), newstart(Point1, Frame)) = t(String2, Point2),
                       work(Newbox2, newstart(Point2, Frame)) = t(String3, Point3)
[B3'12]  work(b(Box1,st,Box2), Frame)      = t(String1 - String2 - String3, Point3)
                  when work(Box1, Frame) = t(String1, Point1),
                       sepbefore(height, Box2) = <Sep, Newbox2>,
                       work(b(Sep), newstart(Point1, Frame)) =t(String2, Point2),
                       work(Newbox2, newstart(Point2, Frame)) = t(String3, Point3)
[B3'13]  work(b(Box1,as,Box2), Frame)      = t(String, Point)
                  when rmoptionals(Box2) = Newbox2,
                       fitshor(b(Box1,pl,Newbox2), Frame) =r(true, String, Point)
[B3'14]  work(b(Box1,as,Box2), Frame)      = work(vertical(b(Box1,as,Box2)), Frame)
                  when rmoptionals(Box2) = Newbox2,
                       fitshor(b(Box1,pl,Newbox2), Frame) = r(false, String, Point)
[B4'1]   fitshor(brack(Box), Frame)        = fitshor(b(~depthchar), Frame)
                  when le(depth(Frame), nat([1])) = true
[B4'2]   fitshor(brack(Box), Frame)        = fitshor(Box, newdepth(D, Frame))
                  when D = depth(Frame) - nat([1]),
                       gt(D, nat([0])) = true
[B4'3]   fitshor(b(String), Frame)         = r(false, ~null, p(nat([0]),nat([0])))
                  when withnewline(String) = true
[B4'4]   fitshor(b(String), Frame)         = r(false, ~null, p(nat([0]),nat([0])))
                  when withnewline(String) = false,
                       Length = length(String),
                       gt(Length, rightm(Frame) - xpoint(Frame) + nat([1])) = true
[B4'5]   fitshor(b(String), Frame)         = r(true, String, p(xpoint(Frame) + Length,
                                                              ypoint(Frame)))
                  when withnewline(String) = false,
                       Length = length(String),
                       le(Length, rightm(Frame) - xpoint(Frame)) = true
```

```
[B4'6]   fitshor(b(String), Frame)          = r(true, String - Newline,
                                                      p(leftm(Frame),
                                                        ypoint(Frame) + nat([1])))
                     when withnewline(String) = false,
                          Length = length(String),
                          eq(Length, rightm(Frame) - xpoint(Frame)+ nat([1])) = true,
                          Newline = fromto(p(rightm(Frame), ypoint(Frame)),
                                           p(leftm(Frame), ypoint(Frame) + nat([1])))
[B4'7]   fitshor(b(C-list, Box), Frame)     = r(false, ~null, p(nat([0]),nat([0])))
                     when modify(C-list, Frame) = Newframe,
                          same-line(start(Newframe), Frame) = false
[B4'8]   fitshor(b(C-list, Box), Frame)     = r(false, ~null, p(nat([0]),nat([0])))
                     when modify(C-list, Frame) = Newframe,
                          same-line(start(Newframe), Frame) = true,
                          fitshor(Box, Newframe) =
                                           r(false, ~null, p(nat([0]),nat([0])))
[B4'9]   fitshor(b(C-list, Box), Frame)     = r(true, String1 - String2, Point2)
                     when modify(C-list, Frame) = Newframe,
                          same-line(start(Newframe), Frame) = true,
                          fitshor(Box, Newframe) = r(true, String2, Point2),
                          fromto(start(Frame), start(Newframe)) = String1
[B4'10]  fitshor(b(Box1,pl,Box2), Frame)          = r(false, ~null, p(nat([0]),nat([0])))
                     when fitshor(Box1, Frame) = r(false, ~null, p(nat([0]),nat([0])))
[B4'11]  fitshor(b(Box1,pl,Box2), Frame)    = r(false, ~null, p(nat([0]),nat([0])))
                     when fitshor(Box1, Frame) = r(true, String1, Point1),
                          changetab(Box2) = Nbox2,
                          sepbefore(width, Nbox2) = <Sep, Newbox2>,
                          fitshor(b(Sep), newstart(Point1, Frame)) =
                                           r(false, ~null, p(nat([0]),nat([0])))
[B4'12]  fitshor(b(Box1,pl,Box2), Frame)          = r(false, ~null, p(nat([0]),nat([0])))
                     when fitshor(Box1, Frame) = r(true, String1, Point1),
                          changetab(Box2) = Nbox2,
                          sepbefore(width, Nbox2) = <Sep, Newbox2>,
                          fitshor(b(Sep), newstart(Point1, Frame)) =
                                                   r(true, String2, Point2),
                          fitshor(Newbox2, newstart(Point2, Frame)) =
                                               r(false, ~null, p(nat([0]),nat([0])))
[B4'13]  fitshor(b(Box1,pl,Box2), Frame)          =
                                        r(true, String1 - String2 - String3, Point3)
                     when fitshor(Box1, Frame) = r(true, String1, Point1),
                          changetab(Box2) = Nbox2,
                          sepbefore(width, Nbox2) = <Sep, Newbox2>,
                          fitshor(b(Sep), newstart(Point1, Frame)) =
                                                      r(true, String2, Point2),
                          fitshor(Newbox2, newstart(Point2, Frame)) =
                                                      r(true, String3, Point3)
[B4'14]  fitshor(b(Box1,as,Box2), Frame)          = fitshor(b(Box1,pl,Newbox2), Frame)
                     when rmoptionals(b(Box1,as,Box2)) = b(Box1,as,Newbox2)
[B4'15]  fitshor(b(Box1,st,Box2), Frame)          = r(false, ~null, p(nat([0]),nat([0])))

[B5'1]   vertical(b(String))                = b(String)
[B5'2]   vertical(brack(Box))               = brack(Box)
[B5'3]   vertical(b(Changelist, Box))       = b(Changelist, Box)
```

```
   [B5'4]  vertical(b(Box1,pl,Box2))          = b(Box1,pl,Box2)
   [B5'5]  vertical(b(Box1,as,Box2))          = b(Box1,st,vertical(Box2))
   [B5'6]  vertical(b(Box1,st,Box2))          = b(Box1,st,Box2)

   [B6'1]  is-duobox(b(String))               = false
   [B6'2]  is-duobox(brack(Box))              = false
   [B6'3]  is-duobox(b(Changelist, Box))      = false
   [B6'4]  is-duobox(b(Box1,pl,Box2))         = true
   [B6'5]  is-duobox(b(Box1,as,Box2))         = true
   [B6'6]  is-duobox(b(Box1,st,Box2))         = true
end BoxtoString
```

Appendix H : Testresults

First we will explain the syntax of Pretty in ASF. This syntax will be used in the tests. For every type used in Pretty an example in SDF-syntax and in ASF-syntax is given. To keep the examples short we will use variables like Int, Box1, Box2. Because in ASF signs like ' and : can not be declared as being of type character, we will use not ofthen used characters like q and x int their place.

	SDF	ASF
character-list	`apple`	`ch([a,p,p,l,e])`
string	`"pear"`	`~ ch([p,e,a,r])`
change	`<-> 5`	`c(width, nat([5]))`
changelist	`| Int1, ->> Int2`	`c(height, Int1) / l(c(abstab, Int2))`
Box with brackets	`[ Box ]`	`brack(Box)`
C-list Box	`-> Int Box`	`b(l(c(reltab, Int)), Box)`
Box1 nothing Box2	`Box1 Box2`	`b(Box1, pl, Box2)` {plus}
Box1 ; Box2	`Box1 ; Box2`	`b(Box1, st, Box2)` {straight}
Box / Box2	`Box1 / Box2`	`b(Box1, as, Box2)` {don't ask me}

First test

Here we test the strings module, so we can see if the strings are really combined in the right way. We also try some small tests on the main module to see if the operators in the box-language are implemented correctly. The page width is set to 30. In test [4] of the module BoxtoString we can also see that the nothing operator will not cut a word in two, when it can place it on the next line. Two characters of the word "possible" would still fit on the current line, but in stead the whole word is placed on the next line.

Input

```
module Strings
begin

terms

[1] ~ch([a,b]) - ~ch([c,d])

[2] ~ch([h]) - (~null $ ~null) - ~ch([i])

[3] length(~ch([a, b, c]))

[4] newlinestring(nat([3]))
```

```
[5] layoutstring(nat([3]))

end Strings


module BoxtoString
begin

terms

[1] makestring(b(~ch([h])))

[2] makestring(b(b(~ch([h])),st,b(l(c(reltab, nat([4]))),b(~ch([i])))))

[3] makestring(b(b(l(c(width, nat([5]))), b(~ch([s,t,e,p]))),st,
                b(c(height, nat([2]))/l(c(reltab, nat([1,8]))),
                          b(~ch([a,s,i,d,e])))))

[3] makestring(b(b(~ch([h,o,w])),pl,b(b(~ch([i,s])),pl,
                b(b(~ch([t,h,e])),pl,b(~ch([w,e,a,t,h,e,r]))))))

[4] makestring(b(b(~ch([n,o])),pl,b(b(~ch([c,l,i,p,p,i,n,g])),pl,
                b(b(~ch([w,i,t,h,i,n])),pl,b(b(~ch([w,o,r,d,s])),pl,
                  b(b(~ch([i,f])),pl,b(~ch([p,o,s,s,i,b,l,e]))))))))

[3] makestring(b(b(~ch([h,o,w])),as,b(b(~ch([i,s])),as,
                b(b(~ch([t,h,e])),as,b(~ch([w,e,a,t,h,e,r]))))))

[6] makestring(b(b(~ch([n,o])),as,b(b(~ch([c,l,i,p,p,i,n,g])),as,
                b(b(~ch([w,i,t,h,i,n])),as,b(b(~ch([w,o,r,d,s])),as,
                  b(b(~ch([i,f])),as,b(~ch([p,o,s,s,i,b,l,e]))))))))

[7] makestring(b(~ch([p,r,e,t,t,y,p,r,i,n,t,i,n,g,i,s,a,v,e,r,y,v,e,r,y,
                     l,o,n,g,w,o,r,d])))

end BoxtoString
```

<u>Output</u>

```
module Strings
begin

   [1]   ~ ch([a, b]) - ~ ch([c, d])

         = ~ ch([a, b, c, d])


   [2]   ~ ch([h]) - (~ null $ ~ null) - ~ ch([i])

         = ~ ch([h]) $ ~ ch([i])
```

```
    [3]    length(~ ch([a, b, c]))

           = nat([3])


    [4]    newlinestring(nat([3]))

           = ~ ch([]) $ (~ ch([]) $ (~ ch([]) $ ~ ch([])))


    [5]    layoutstring(nat([3]))

           = ~ ch([b, b, b])


end Strings



module BoxtoString
begin

    [1]    makestring(b(~ ch([h])))

           = ~ ch([h])


    [2]    makestring(b(b(~ ch([h])),
                        st,
                        b(l(c(reltab, nat([4]))), b(~ ch([i])))))

           = ~ ch([h]) $ ~ ch([b, b, b, b, i])


    [3]    makestring(b(b(l(c(width, nat([5]))), b(~ ch([s, t, e, p]))),
                        st,
                        b(c(height, nat([2])) / l(c(reltab, nat([1, 8]))),
                          b(~ ch([a, s, i, d, e])))))

           = ~ ch([b, b, b, b, b, s, t, e, p]) $ ~ ch([]) $
             ~ ch([b, b, b, b, b, b, b, b, b, b, b, b,
                                          b, b, b, b, b, b, a, s, i, d, e])


    [3]    makestring(b(b(~ ch([h, o, w])),
                        pl,
                        b(b(~ ch([i, s])),
                          pl,
                          b(b(~ ch([t, h, e])),
                            pl,
                            b(~ ch([w, e, a, t, h, e, r]))))))

           = ~ ch([h, o, w, b, i, s, b, t, h, e, b, w, e, a, t, h, e, r])
```

```
[4]    makestring(b(b(~ ch([n, o])),
                   pl,
                   b(b(~ ch([c, l, i, p, p, i, n, g])),
                     pl,
                     b(b(~ ch([w, i, t, h, i, n])),
                       pl,
                       b(b(~ ch([w, o, r, d, s])),
                         pl,
                         b(b(~ ch([i, f])),
                           pl,
                           b(~ ch([p, o, s, s, i, b, l, e]))))))))

      = ~ ch([n, o, b, c, l, i, p, p, i, n, g,
                 b, w, i, t, h, i, n, b, w, o, r, d, s, b, i, f, b]) $
          ~ ch([p, o, s, s, i, b, l, e])

[3]    makestring(b(b(~ ch([h, o, w])),
                   as,
                   b(b(~ ch([i, s])),
                     as,
                     b(b(~ ch([t, h, e])),
                       as,
                       b(~ ch([w, e, a, t, h, e, r]))))))

       = ~ ch([h, o, w, b, i, s, b, t, h, e, b, w, e, a, t, h, e, r])

[6]    makestring(b(b(~ ch([n, o])),
                   as,
                   b(b(~ ch([c, l, i, p, p, i, n, g])),
                     as,
                     b(b(~ ch([w, i, t, h, i, n])),
                       as,
                       b(b(~ ch([w, o, r, d, s])),
                         as,
                         b(b(~ ch([i, f])),
                           as,
                           b(~ ch([p, o, s, s, i, b, l, e]))))))))

       = ~ ch([n, o]) $
         ~ ch([c, l, i, p, p, i, n, g]) $
         ~ ch([w, i, t, h, i, n]) $
         ~ ch([w, o, r, d, s]) $
         ~ ch([i, f]) $
         ~ ch([p, o, s, s, i, b, l, e])

[7]    makestring(b(~ ch([p, r, e, t, t, y, p, r, i, n, t, i, n, g, i,
                           s, a, v, e, r, y, v, e, r, y, l, o, n, g, w,
                           o, r, d])))

       = ~ ch([p, r, e, t, t, y, p, r, i, n, t, i, n, g, i, s, a,
                           v, e, r, y, v, e, r, y, l, o, n, g, w]) $
         ~ ch([o, r, d])
end BoxtoString
```

Second test

Here we try one larger test on a case-statement of Pascal. For the first part of the test the pagewidth is set to 60. This is done by changing the value of the right margin in the defaults-equation in module box-syntax. We will show you the input, the output and a readable form of the output of this test. Then we test the same input, but now with a pagewidth of 30. The output and the readable form of the output of this test are shown.

Input

```
module BoxtoString
begin

terms

[2] makestring(

b(b(b(~ch([c,a,s,e])),pl,b(b(~ch([b,u,f,f,h,e,k])),pl,b(~ch([o,f])))),st,
    b(b(l(c(reltab, nat([4]))), brack(b(b(b(~ch([q,o,q])),pl,
                  b(b(l(c(abstab, nat([1,1]))), b(~ch([x]))),pl,
                        brack(b(b(~ch([b,e,g,i,n])),st,
      b(b(l(c(reltab, nat([4]))), brack(b(b(b(~ch([f,i,l,e,d,e,s,c])),pl,
      b(b(~ch([x,y])),pl,b(b(~ch([o,p,e,n,f,i,l,e,g,b,u,f,f,h])),pl,
                                    b(~ch([z]))))),st,
            b(b(~ch([r,e,p,l,y,b,u,f,f])),pl,b(b(~ch([x,y])),pl,
                        b(b(~ch([q,o,k,q])),pl,b(~ch([z])))))))),st,
                    b(b(~ch([e,n,d])),pl,b(~ch([z])))))))),st,
               b(b(~ch([q,r,q])),pl,b(b(~ch([v])),pl,b(b(~ch([q,c,q])),pl,
                  b(b(l(c(abstab,nat([1,1]))), b(~ch([x]))),pl,
               brack(b(b(~ch([r,e,p,l,y,b,u,f,f])),pl,
                        b(b(~ch([x,y])),pl,
                        b(b(~ch([q,n,o,t,b,o,p,e,n,q])),pl,
                        b(~ch([z]))))))))))))),st,
                         b(~ch([e,n,d])))))
end BoxtoString
```

Output

```
module BoxtoString
begin

   [2]    makestring(b(b(b(~ ch([c, a, s, e])),etc....
 = ~ ch([c, a, s, e, b, b, u, f, f, h, e, k, b, o, f]) $
   ~ ch([b, b, b, b, q, o, q, b, b, b, b, b, b, b, x, b, b, e, g, i, n]) $
   ~ ch([b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, f,
        i, l, e, d, e, s, c, b, x, y, b, o, p, e, n, f, i, l, e, g, b,
        u, f, f, h, b, z]) $
   ~ ch([b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, r,
        e, p, l, y, b, u, f, f, b, x, y, b, q, o, k, q, b, z]) $
   ~ ch([b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, e, n, d, b, z]) $
   ~ ch([b, b, b, b, q, r, q, b, v, b, q, c, q, b, x, b, r, e, p, l, y, b,
        u, f, f, b, x, y, b, q, n, o, t, b, o, p, e, n, q, b, z]) $
   ~ ch([e, n, d])
```

```
end BoxtoString
```

CPU time: 74.8

We now change the outpurt slightly to make it more readable.

```
= ~ ch([c, a, s, e,  , b, u, f, f, h, e, k,  , o, f]) $
  ~ ch([ ,  ,  ,  , q, o, q,  ,  ,  ,  ,  ,  ,  , x,  , b, e, g, i, n])     $
  ~ ch([ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  , f, i, l, e, d, e, s, c,  , x, y,  , o, p, e, n, f, i, l, e, g,
                                                                                        b,u, f, f, h,  , z]) $
  ~ ch([ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  , r, e, p, l, y, b, u, f, f,  , x, y,  , q, o, k, q,  , z]) $
  ~ ch([ ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  ,  , e, n, d,  , z])      $
  ~ ch([ ,  ,  ,  , q, r, q,  , v,  , q, c, q,  , x,  , r, e, p, l, y, b, u, f, f,  , x, y,  , q, n, o, t,  , o, p, e, n, q,  , z]) $
  ~ ch([e, n, d])
```

```
= ~ ch([case buff[1] of]) $
  ~ ch([   'o'      : begin]) $
  ~ ch([                  filedesc := openfile(buff) ;]) $
  ~ ch([                  replybuff := 'ok' ;]) $
  ~ ch([                end ;]) $
  ~ ch([   'r' , 'c' : replybuff := 'not open' ;]) $
  ~ ch([end])
```

Output

```
module BoxtoString
begin

   [2]    makestring(b(b(b(~ ch([c,a,s,e])),....etc.

   = ~ ch([c,a,s,e,b,b,u,f,f,h,e,k,b,o,f]) $
     ~ ch([b,b,b,b,q,o,q,b,b,b,b,b,b,b,x,b,b,e,g,i,n]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,f,i,l,e,d,e,s,c,b]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,x,y,b,o,p,e,n,f,i,l]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,e,g,b,u,f,f,h,b,z]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,r,e,p,l,y,b,u,f,f,b]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,x,y,b,q,o,k,q,b,z]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,e,n,d,b,z]) $
     ~ ch([b,b,b,b,q,r,q,b,v,b,q,c,q,b,x,b,r,e,p,l,y,b,u,f,f,b,x,y,b]) $
     ~ ch([b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,q,n,o,t,b,o,p,e,n,q,b,z]) $
     ~ ch([e,n,d])

end BoxtoString
```

CPU time: 93.3167

Again we change the output slightly to make it more readable.

[2] makestring(b(b(b(~ ch([c,a,s,e])),....etc.

```
= ~ ch([case buff(1) of]) $
  ~ ch([   'o'      : begin]) $
  ~ ch([                 filedescb]) $
  ~ ch([                 := openfil]) $
  ~ ch([                 e(buff) ;]) $
  ~ ch([                 replybuff ]) $
  ~ ch([                  := 'ok' ;]) $
  ~ ch([                end ;]) $
  ~ ch([   'r' , 'c' : replybuff := ]) $
  ~ ch([                 'not open' ;]) $
  ~ ch([end])
```