Science and Engineering Research Council
Rutherford Appleton Laboratory
CHILTON, DIDCOT, OXON, OX11 0QX

RAL-87-024

A Kernel for a Generator of Syntax Driven Editors for Dimensional Designs

Prof. M Bertran-Salvans

April 1987

TABLE OF CONTENTS

A KERNEL FOR A GENERATOR OF SYNTAX DRIVEN EDITORS FOR DIMENSIONAL DESIGNS

*Miquel Bertran-Salvans**

Rutherford Appleton Laboratory, Chilton, Didcot, UK (Visiting)

1. INTRODUCTION AND OVERVIEW

The present document has as a primary purpose the provision of the necessary information for understanding, usage and continuation of a library of procedures which forms the nucleus of a generator of syntax driven editors for dimensional designs [1],[2]. This represents a fundamental step towards the fulfilment of the requirements for a Dimensional Design Environment [8].

The relation of DD's and formal specifications is explored as well, and examples of readability enhancement in displaying abstract syntax trees as DD's are given. In addition, the abstract type DD, needed for the generator, is specified formally. All this is done in Section 6. Subsections 6.1 and 6.2 give an introduction to DD's and their grammars. Readers not familiar with DD's are advised to read these sections first.

The key idea for the generator is the construction of syntax trees. If this process is done by substitution of non-terminals, then the corresponding DD phrase will be syntactically correct, with respect to the grammar.

In addition to the routines for the construction of syntax trees, input and output functions for grammars and DD's are provided as well. They involve the corresponding external representations, which are also defined. The input functions constitute non-backtracking parsers for grammars and DD's, involving error generation.

Finally, some macrodefinitions for plotting DD's in documents, like the ones in this one, are also available and explained.

2. EDITABLE GRAMMAR FORMAT

The editor is grammar dependent. By changing the grammar, syntax-driven editing of different dimensional languages can take place. In the current version, grammars are edited with any normal text editor. This section gives information and an example about the format of the editable grammars.

As a general comment, this metalanguage corresponds to a simple extension of BNF. Of course, it accounts for the four dimensions: cuboid (c), diagonal(d), vertical(v), and horizontal(h); which are not present in BNF. Therefore, four dimensional terminals have been added. They are formed by two characters, the first is one of the dimensional letters (c,d,v,h). The second one denotes the font for the corresponding cuboid lines or edges.

The following line fonts are available in the current implementation: normal (n), invisible normal line (i), grammar line (g), invisible grammar line (r), and dashed line (d).

The specification of DD stuctures is done in levels. Each DD level starts with a cuboid and ends with the end character `e`. In between both, there may exist diagonal, vertical, and horizontal DD's, preceded by their corresponding dimensional terminals. Their presence is optional, but if present, they should appear in

* On leave of absence from Escola T Superior E Telecomunicacio, Universitat Politecnica de Catalunya, Barcelona, Spain.

the above order.

As an additional restriction, non-terminals within rule bodies (primitive or not) should have empty diagonal DD's. According to the expansion conventions for the formation of syntax trees, this restriction amounts to saying that non-terminal references should be unexpanded.

In addition to terminals and non-terminals, which are represented as usual in many BNF versions, a new kind of non-terminal has been added. It is referred here as *primitive non-terminal*, and corresponds to frequently used non-terminals. The motivation being the avoidance of providing rules for them. Their names are prefixed with a semicolon. The current version has three of them: *:textline*, *:identifier*, and *:nil*. Non-terminals and terminals, primitive or not, are equivalent to complete cuboids starting a DD level. Therefore, for any one of them an end character will have to be provided.

Rules may be either selection or non-selection rules. The bodies of the last ones start with the character *'?'*. Independently, they may also be either recursive on non-recursive. The former have the character *'r'* before the equal sign separating the heading from the body of any rule.

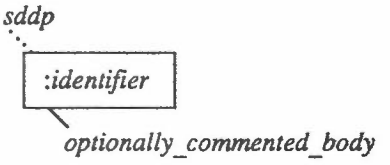
2.1. Example of a grammar in editable format.

A simple Pascal-like language has been chosen. Programs are now dimensional design representations, as defined by the grammar rules.

The first rule defines the global structure of any program in the language. Any program should have a name, inserted within a cuboid, and its diagonal DD will correspond to the body of the program, which may have a global comment.

<sddp> =(cn :identifier e dn <optionally_commented_body>ee)

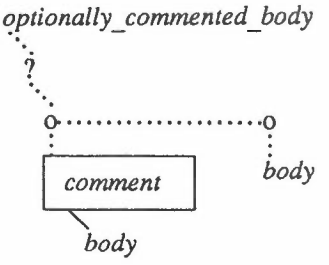
This first rule can be represented as the following equivalent DD:



The second rule is an example of a selection type rule.

<optionally_commented_body> =?(cn <comment>e dn<body>ee | <body>e)

It corresponds to the following DD representation.



The rest of the rules are the following:

<comment> r=?(:textline vn<comment>ee | :nil e)

<body> =('var' dn cn<variable_declarations>ee

```
            vn 'algor' dn cn<statement>e e e e  )
<variable_declarations> r=?( cn<comment>e dn<variable_declarations>ee
                          / <actual_variable_declarations>e  )
<actual_variable_declarations>  r=?( cn<identifier_list>e dn<type>e vn<variable_declarations>ee
                          / :nil e  )
<identifier_list> =( :identifier hn <rest_of_identifier_list>ee )
<rest_of_identifier_list> r=?( ',' hn :ide hn <rest_of_identifier_list>eee
                          / :nil e )
<type> =?( 'boolean' e / 'character' e / 'integer' e / 'real' e )
<statement> r=?( cn <comment>e dn <statement>e e / <statement_sequence>e )
<statement_sequence> r=?( <basic_statement> vn <statement>e e / :nil e )
<basic_statement> r=?( <if_statement>e / <while_statement>e  / <assignment>e / 'skip' e )
<if_statement> r=( 'if' dn 'o' vn <boolean_expression> vn <statement>ee
                          hn 'else' vn <statement> e e e e    )
<while_statement> r=( 'while' dn <boolean_expression> vn <statement>eee )
<assignment> =( :ide hi ':=' hi <expression>eee )
<expression> =?( <boolean_expression>e / <arithmetic_expression>e )
<boolean_expression> =?( 'not' hi :ide e e
                    / :ide hi <logical_binary_operator> hi :ide eee
                    / <arithmetic_expression>
                        hi  <logic_comparison_operator>
                            hi  <arithmetic_expression>e e e    )
<arithmetic_expression> =?( :ide e  /  '-' hi :ide ee
                          / :ide hi <arithmetic_operator> hi :ide eee  )
<logical_binary_operator> =?( 'and' e / 'or' e )
<logic_comparison_operator> =?('='e/'<'e/'>'e/'<>'e/'<='e/'>='e)
<arithmetic_operator> =?('+'e/'-'e/'*'e/'/'e/'exp'e)
```

3. ABSTRACT TYPES AND THEIR OPERATIONS

The procedures and functions of the library are grouped around types. In general, any of them is implemented in terms of procedures or functions of other types; this defines a dependancy hierarchy among types, going from the most basic ones at the bottom to the more global ones at the top.

This section describes the current implementation of the types; i.e. their functions and procedures. It starts with the grammar syntax-tree, the topmost type, and ends with line attribute and character as the bottommost ones. Their presentation ordering corresponds to their hierarchical ordering. Functions and procedures of any given type do not depend on the ones of the precedeeng ones.

3.1. Grammar syntax tree abstract type.

The abstract type grammar syntax tree (ddgstree) is represented internally as a dd, **type** *ddgstree = dd*. It depends on the abstract type *grammar*. Syntax trees are constructed by expanding non-terminals of grammar rules with copies of their rule bodies, taken from the grammar. The process starts with the first rule in the grammar. This gives rise to syntax trees having selection constructs; in other words undecided selections. Strictly speaking, they represent collections of syntax trees. Therefore, it is mandatory to select one alternative for each of the selections.

When all non-terminals have been expanded, and there remains no undecided selection, one has a proper syntax tree. It will be the syntax tree of a syntactically correct program. This is true when no primitive non-terminal is involved in the syntax tree. When they are present, they have to be expanded with the help of appropriate input routines of the text line abstract type. This process is covered at the last section.

Boolean functions.

Is valid unexpanded non-terminal occurrence in st ?

function isunexpo(ddginst:ddgram;stinst:ddgstree;nontinst:textln):boolean;

True if the non-terminal instance within the syntax tree is not expanded yet.

Is valid alternative top of a syntax tree ?

function isaltop(altop: dd): boolean;

True if the given dd is a valid alternative top of a syntax tree. It should correspond to the structural dot of an alternative.

Selection functions

Alternative embedding non-terminal dd

function altnont(altop: dd): dd;

Gives the least non-terminal dd in a syntax tree containing the given alternative. The non-terminal corresponds to the selection rule of the alternative. *Altop* is a character dd whose character corresponds to the structural dot of the alternative.

Constructor functions

Make nil ddg syntaxtree

function mnilst (ddginst: ddgram): ddgstree;

Takes the value of the empty syntaxtree of the given grammar instance *(ddginst)*. Such a nil syntax tree is, by definition, an unrefined non-terminal copy of the goal non-terminal of *ddginst*.

Make expanded non-terminal ddginst syntaxtree

function mexpntst (ddginst:ddgram; stinst:ddgstree; nontinst: textln): ddgstree;

Gives the syntaxtree resulting from expanding the given non-terminal instance within the given syntax tree. It is assumed that *isnildd(diagdd(nontinst))*, i.e. that the non-terminal has an empty diagonal dd in the grammar and that it has not been expanded yet within the syntax tree. Expansion involves the attachment of a diagonal dd, the body of the corresponding ddginst rule.

Make alternative selection

function mselalt(ddginst:ddgram;stinst:ddgstree;altop:dd): ddgstree;

Given a syntax tree, *stinst*, in a given grammar, *ddginst*, and a dd part, *altop*, of the syntax tree

corresponding to an alternative of one of the still undecided selections, the function takes the value of the syntax tree resulting from selecting the given alternative for the selection nonterminal. *Altop* is a character dd whose character corresponds to the structural dot of the alternative.

Derived constructor functions

Recursively expanded ddg syntax tree

function recexpst(ddginst:ddgram;stinst:ddgstree;nontinst:textln):ddgstree;

Gives the syntax tree resulting from expanding the given non-terminal instance, within the given syntax tree; and , if non-recursive, of iteratively expanding the non-terminals of the rule body expansion as well. The same assumptions, as in *mexpntst*, about *nontinst* are made.

Expand rule body within ddgram syntax-tree

This is a function internal to *recexpst*. This is due to the Pascal nesting of functions.

function exprlbdy(ddginst:ddgram;stinst:ddgstree;rlbdy:dd): dd;

Gives the syntax tree with the expansion of the dd part corresponding to one of its unexpanded rule bodies, *rlbdy*. Thus, the cuboid part of *rlbdy* does not have to be a non-terminal, as is the case in *recexpst*. The implementation algorithm searches the rule body for unexpanded non-terminals, as candidates for expansion. It assumes that all non-terminals which are encountered are unexpanded.

3.2. Grammar abstract type.

It is represented internally as a dd, type *ddgram = dd*. One of the external codings of grammars has already been covered in Section 2. A different coding can correspond to a DD, as a copy of its internal representation; this format is covered in Section 4. The internal representation is organised as a vertical chain of rules, each rule being a non-terminal DD with a diagonal DD corresponding to the body of the rule.

Selector functions.

First rule.

function frstrule(ddgrmi: ddgram): rule;

Gives the first rule in the grammar. It corresponds to the first within its list of rules.

Next rule.

function nextrule(ruleinst: rule): rule;

Given a rule, it gives its next rule within the list of rules of the grammar.

Named grammar rule.

function gramrule(ddginst: ddgram; nontinst: textln): rule;

Gives the grammar rule, within *ddginst* grammar, of the given non-terminal *nontinst*.

Input functions.

Input grammar.

function inddgrm: ddgram;

Inputs a DD grammar coded in linear editable format (Section 2.). Maps it into its internal dd representation. The external coding is editable with standard editors, it is distinct from the external coding of dd's. However, since this implementation represents grammars internaly as dd's, they can be written and read as dd's using the input-output functions of the dd abstract type. It invokes the rule input function, and assumes that new rules start at new lines.

Output procedures.

Output a grammar.

procedure outddgrm(ddgrmi: ddgram);

Outpus a dd grammar in readable linear form using standard tabulation conventions.

3.3. Abstract type rule.

The abstract type rule (rule), is represented internally as a dd, **type** *rule = dd*. It corresponds to a non-terminal DD having an empty vertical DD. The body of the rule is the diagonal DD.

Boolean functions.

Is valid rule ?

function isrule(posrule:dd): boolean;

True if the dd argument starts with a non-terminal whose diagonal edge attribute is a grammar line. The vertical DD is not checked for emptyness since the rule may be embedded within another DD, as for instance in grammars.

Is a selection rule ?

function isselrul(posrule:dd):boolean

True if dd argument is a rule and its diagonal dd corresponds to a character dd with the character `?`.

Is recursive rule ?

function isrecrul(posrule:dd): boolean

True if dd argument is a rule and its horizontal dd is an isolated character dd with the character `r` and normal italic character attribute, *nrmitlcr*.

Selector functions.

Rule body.

function rulebody(ruleinst:rule):dd

Takes the value of the body of the given rule, assuming a dd representation for rules. Prints error if invalid rule.

First alternative of a selection rule.

function firstalt(selrule:rule): rulealt

Gives the first alternative of the argument rule. Prints error message if the argument is not a selection rule.

Next alternative of a selection rule.

function nextalt(rulealti:rulealt):rulealt;

Given a parameter alternative, gives its next alternative in its rule. Returns the empty DD if the given alternative was the last one. Assumes the parameter to be a valid alternative, i.e. the vertical DD of the

structural dot DD of a selection rule.

Input functions.

Input grammar rule function.

function inrule: rule;

Inputs a grammar rule, coded in linear form, mapping it into its internal dd representation. Admits two types of rules: selection and non-selection rules. Also, a rule may be either of recursive or of non-recursive type. Prints error messages if the global rule structure, *< ... =(...)* or *< ... =?(... / ... ...)*, is unsuccessfully parsed. It invokes the rule alternative input function.

Output procedures.

Output grammar rule.

procedure outrule(ruleinst:rule);

Outputs a grammar rule, coded in linear form and with standard tabulation; starting at a new line. Prints an error message if the internal rule to be printed is not a valid rule. It invokes the rule alternative output procedure.

3.4. Abstract type rule alternative.

The rule alternative type is represented internally as a dd. type *rulealt = dd*. Any rule has at least one alternative. Thus, the body of a non-selection rule is an alternative. Isolated character DD's are allowed in a rule body only as parts of terminals, non-terminals, or primitive non-terminals. Thus its DD levels should start with any of these metasymbols.

Input functions.

Input rule alternative.

function inrlalt: rulealt;

Inputs a rule alternative, coded in linear form, mapping it into its internal dd representation. This linear coding has cuboid, diagonal, vertical and horizontal dd parts, in that order. Grammar terminals, non-terminals or primitive non-terminals may stand at the cuboid part, instead of an explicit cuboid. An `e` character is expected as an end symbol, in order to close all DD levels started with an implicit or explicit cuboid. Error messages are printed if either no cuboid or no end symbol can be detected.

Output procedures.

Output a rule alternative.

procedure outrlalt(alt:rulealt; col:integer);

Outputs a rule alternative, coded in linear form and with standard tabulation. The value of *col* should equal the current column number. Assumes internal representation as a dd. Prints an error message if any DD level starts with a cuboid which is invalid for a rule alternative.

3.5. Textline abstract type.

This type (textln) is represented as a dd, type *textln = dd*. It contains the following subtypes: identifier string (idstr), grammar terminal, grammar non-terminal, grammar primitive non-terminal. Primitive non-terminals are built in into the library; thus, they need not be further expanded with a grammar rule. The

current subtype corresponds to their names only. There exist three primitive non-terminals in the current version: identifier, textline, and nil.

In addition, the string of blank characters is also considered as a subtype, even when more than one line is involved. This helps to view the input text as a continuous string of characters within the implementation.

Constant functions.

They are constants of the primitive non-terminal subtype.

Primitive non-terminal identifier.

function prmntide: textln;

Internally represented as an horizontal chain of single character dd's corresponding to the string `:ide`, where the symbol `:` stands for the primitive non-terminal delimiter (primntdl), which can be changed (see type character). The string dd is enclosed within a cuboid. All lines and edges are grammar lines (gramline) but the characters have the normal italic character attribute (nrmitlcr), with the exception of the delimiter which has a normal character attribute (normchar).

Primitive non-terminal text line.

function prmnttex: textln;

The same as the first function but with the string `:tex`.

Primitive non-terminal nil.

function prmntnil: textln;

The same as the first function but with the string `:nil`.

Boolean functions.

Is a valid text-line ?

function istextln(var text: textln): boolean;

True if parameter dd corresponds to a textline, perhaps connected to other dd's. In other words, if cuboid dd is a linear horizontal dd of characters. Notice that only the cuboid dd of the parameter enters in the checking, therefore the cuboid enclosing the textline may be connected to diagonal, vertical, and horizontal dd's. This definition is motivated to facilitate the search of textlines embedded in larger dd's.

Is valid identifier string ?

function isidstr (text: textln): boolean;

True if text line is a valid identifier. In other words a texline, as in the last function, but consisting of valid identifier characters, *isidencr(ddchar)* (see character type).

Is grammar non-terminal ?

function isnoterm (text: textln): boolean;

True if text is a valid identifier textline of normal italic characters (nrmitlcr).

Non-terminal equality.

function eqnontrm(nont1,nont2: textln): boolean;

True if both parameters are valid nonterminals, *isnoterm(nonti)*, and if they are equal. If at least one of them is not valid, then an error message is printed.

Is grammar terminal ?

function istermnl(text: textln): boolean;

True if textline is a valid textline of characters, the attribute of none of which corresponds to a normal italic character (nrmitlcr), and also, none of which is a standard terminal delimiter (stermldl).

Is a valid primitive non-terminal ?

function isprmnt (tdd: dd): boolean;

True if the cuboid of *tdd*, including its contents, equals one of the defined primitive non-terminals. The diagonal, vertical, and horizontal dd's are irrelevant for the test.

Input functions.

Input text line.

function intxtln: textln;

Maps the remaining text in the current input line into its internal dd representation as a text line. This function uses normal characters and invisible normal lines for edges and cuboids. Prints error in case of empty line.

Input identifier or non-terminal string

function inidstr (selector:notermsl): textln;

Assumes a valid identifier in the next input, i.e. a string starting with an alphabetic character and consisting of alphanumeric characters plus `_` and `.`. Maps this identifier to an internal textline, newly created. When the selector value is 'noterm', the dd attributes of grammar line and italic character are selected for the internal text line, otherwise the normal text line attributes are selected. Prints error in case of empty identifier. Any character which does not satisfy *isidencr*, is identifier character, for instance a blank, will be taken as the right delimiter, which will be lost.

Input a grammar terminal

function intermnl : textln;

Maps the following input text up to, but not including, the next terminal delimiter (istermdl), into an internal text line, using normal characters and invisible normal lines for edges and cuboids. The terminal delimiter is lost. Also it cannot appear in the terminal string. Prints error in case of an empty terminal string, or if an end of line is encountered before the terminal delimiter. Therefore, it assumes that the left terminal delimiter has just been parsed.

Input primitive non-terminal.

function inprmnt: textln;

Inputs one of the defined primitive non-terminal names. Maps the first three characters of the input, preceded by a primitive non-terminal delimiter, into an internal textline, of four characters. The remaining identifier-type characters are input but lost. The first character after them which is not an identifier character is also input and lost. Prints error if the internal textline matches no primitive non-terminal. Thus, it assumes that the starting primitive non-terminal delimiter has just been parsed in the input.

Input next non-blank character.

function nextnbcr: char;

Maps an input string of blanks terminated with a non-blank character, into the terminating character (as an internal char). Skips blank lines of input if necessary.

Output procedures.

Output an internal textline.

procedure outtxtln (text:textln);

Maps a non-empty internal textline into the next positions of the output, as regular text. Prints an error message in case of an empty textline.

Output identifier or terminal string.

procedure outidstr (selector:notermsl; identif:textln);

Maps an internal identifier textline into the next positions of the output, as regular text in the current implementation. If the selector denotes a non-terminal, then the textline is enclosed within angular brackets, and an error message is printed in case of an invalid internal non-terminal. Prints an error message in case of an empty textline.

Output grammar terminal.

procedure outermnl (text: textln);

Outputs an internal text line as a grammar terminal, enclosed within terminal delimiters. Prints an error message in case of an invalid non-terminal.

Output a primitive non-terminal name.

procedure outprmnt (text: textln);

Maps an internal primitive non-terminal reference into the next position of the output, as regular text. Prints error in case of invalid primitive non-terminal.

Output blanks and blank lines.

procedure outblncs(iniblncs,nlines,endblncs:integer);

Outputs a number of blanks equal to *iniblancs* in the current line of output, a number of blank lines equal to *nlines-1*, and if *nlines>0*, a last line with a number of blanks equal to *endblncs*.

3.6. Abstract type DD.

The internal representation of DD's is detailed in Appendix II, Section 6.3. This appendix specifies the constructors, selectors, and booleans given below. However, the present implementation does not make copies of the parameter DD's when constructing a new DD, or when selecting a DD part.

Constructors.

Make nil DD.

function mnildd: dd;

Gives the empty DD.

Make character DD.

function mchardd (cd:char; chat:charattr): dd;

Make cuboid DD.

function mcubdd (ddc: dd; lat: lineattr): dd;

Given a DD and a line attribute, surrounds it with a cuboid rectangle, of the given attribute, thus making a complete cuboid.

Make diagonal DD.

function mdiagdd (ddg,ddd:dd; lat: lineattr): dd;

Given a DD with empty diagonal DD, a second DD, *ddd*, and a line attribute, *mdiagdd* attaches the second DD as the diagonal DD of the first one, with an edge having the given line attribute.

Make vertical DD.

function mvertdd (ddg,ddv: dd; lat: lineattr): dd;

As *mdiagdd* but for the vertical DD.

Make horizontal DD.

function mhordd (ddg,ddh: dd; lat: lineattr): dd;

As *mdiagdd* but for the horizontal DD.

Selectors.

DD character.

function ddchar (ddch:dd):char;

The parameter should be a character DD. Its character is returned.

Cuboid DD.

function cubdd (ddc: dd): dd;

The parameter should be a cuboid DD which is not a character DD. It returns the DD within the cuboid.

Diagonal DD.

function diagdd (ddd: dd): dd;

The parameter should be a non empty DD. It returns its diagonal DD.

Vertical DD.

function vertdd (ddv: dd): dd;

Similar to *diagdd*.

Horizontal DD.

function hordd (ddh: dd): dd;

Similar to *diagdd*.

DD character attribute.

function ddcrattr(ddn: dd): charattr;

The parameter should be a character DD. It returns the character attribute, which contains the character font and the line attribute of the cuboid rectangle.

Cuboid line attribute.

function cubattr(ddc: dd): lineattr;

The parameter should be a cuboid DD which is not a character DD. It returns the line attribute of its cuboid rectangle.

Diagonal edge attribute.

function degattr(ddd: dd): lineattr;

For a non empty DD, it returns the line attribute of its diagonal edge.

Vertical edge attribute.

function vegattr(ddv: dd): lineattr;

Similar as in *degattr*.

Horizontal edge attribute.

function hegattr(ddh: dd): lineattr;

Similar as in *degattr*.

Least embedding DD.

function lstembdd(ddinst:dd):dd;

Takes the value of the father dd of a given dd , a dd instance of a `larger` dd. Takes the nildd value when the given dd has no father.

Boolean operations.

function isnildd (ddc: dd): boolean;

Is a character DD ?

function ischardd (ddc: dd): boolean;

True if the cuboid of the parameter DD corresponds to a character.

DD equality function

function eqdd (dd1, dd2: dd): boolean;

Derived constructor operations.

Copy dd.

function copydd(inidd: dd): dd;

Obtains a copy of the parameter dd. Needed since the present implementation of the constructors uses their parameter dd's as parts of their resulting dd's.

Detach a ddpart from a given dd.

function mdetchdd (ddg,dds: dd): dd;

Disconnects the dd at a ddpoint (dds) of a given dd. The implementation assumes that a ddpoint is represented as a pointer into a dd. It is not checked whether or not the ddpoint is pointing to a part of the given dd (i.e. ddg).

Input functions.

function indd:dd;

Inputs a DD coded in its external format (see Section 4).

Output procedures.

procedure outdd(odd:dd);

Outputs a DD coded in its external format.

3.7. Character attribute abstract type.

It is implemented as a pointer to record, as the following Pascal declaration shows:

```
type    charattr = ^chatnode;
        charfont = (nc,it,bl);
        chatnode = record  carfont: charfont;
                           cclat  : lineattr
                   end;
```

It depends on the abstract type *lineattr*. The type *charfont* is used as an abstract type which needs no explicit definition; due to its Pascal representation.

Constants.

Normal character.

function normchar: charattr;

Normal character font (nc) and normal invisible cuboid line, *nrminvln*.

Normal italic character.

function nrmitlcr: charattr;

Used for grammar non-terminal names. Italic character font (it), and invisible grammar line for its cuboid rectangle, *invgrmln*.

Normal boldface character.

function nrmbldcr: charattr;

Boldface character (bl) with normal invisible line for its cuboid rectangle, *nrminvln*.

Constructors.

Make character attribute.

function mchat (cf: charfont; lat: lineattr): charattr;

Selectors.

Character cuboid attribute.

function ccubattr (chat: charattr): lineattr;

Character font attribute.

function cfntattr (chat: charattr): charfont;

Boolean functions.

Character attribute equality.

function eqchat (chat1,chat2: charattr): boolean;

Input functions.

Input character attribute.

function inchat: charattr;

Coded within one external character. The implemented codings are: `n` for normal character, `i` for normal italic character, and `b` for normal boldface character. Prints an error if none of the above characters is at the current input location.

Output procedures.

Output character attribute.

procedure outchat (chat: charattr);

Outputs the external coding of a character attribute, as detailed in the input function. Codes with a blank any invalid internal character attribute; it prints an error message as well.

3.8. Line attribute abstract type.

It is represented as an enumerated type:

type *lineattr = (nl,il,gl,ig,ds);*

Constants.

Normal line.

function normline: lineattr;

Normal invisible line.

function nrminvln: lineattr;

Grammar line.

function gramline: lineattr;

Invisible grammar line.

function invgrmln: lineattr;

Normal dashed line.

function nrmdshln: lineattr;

Boolean functions.

Line attribute equality.

function eqlat (lat1, lat2: lineattr): boolean;

Input functions.

Input line attribute.

function inlat: lineattr;

Inputs one character as the external coding of a line attribute. The implemented codings are the following: `n` for normal lines, `i` for invisible lines (used for character cuboids or length zero edges.), `g` for grammar lines (grammar cuboids and edges), `r` for invisible grammar lines (within non-terminals), `d` for dashed lines. Prints an error message for any other character.

Output procedures.

Output line attribute.

procedure outlat (lat: lineattr);

Outputs the external coding of an internal line attribute, as in the input function. Prints an error message for an invalid internal line attribute.

3.9. Character abstract type.

Although the current implementation is based on the Pascal character type, this type collects a set of functions and procedures dealing with single characters which are specific to the generator.

Grammar terminal standard delimiter. Corresponds to the Pascal constant: **const** *stermldl* = '''';

Primitive non-terminal delimiter. Corresponds to the Pascal constant **const** *primntdl* = ':';

Boolean functions.

Is decimal digit ?

function isdecdig (c:char): boolean;

Is alphabetic character ?

function isalphcr (c:char): boolean;

Is arithmetic operator ?

function isaritop (c:char): boolean;

Is a valid identifier character ?

function isidencr (c:char): boolean;

Alphanumeric character plus `.` and `_`.

Is a punctuation character ?

function ispuntcr (c:char): boolean;

True for any of the following characters: { . , : ; ? ! }

Is terminal delimiter (standard or non-standard) ?

function istermdl (c:char): boolean;

True if character is equal to *stermldl*, or `"` .

Is non-terminal delimiter ?

function isntrmdl (c: char): boolean;

True for the characters `<` and `>`.

Is primitive non-terminal delimiter ?

function isprmntd (c: char): boolean;

True if character is equal to *primntdl*.

4. EXTERNAL FORMAT OF A DD

The extenal coding of a DD has a resemblance with the format of the alternatives of grammar rules. As there, there are DD levels here. However, the format is more rigid now, and blanks are not allowed.

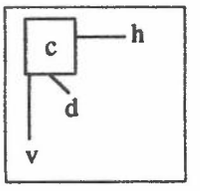
Each DD level starts with either the character '@' or with any other character. In the former case, the level srtarts with a cuboid, otherwise it corresponds to a character DD with the same character. The font of the cuboid line is coded at the character next to '@'. Similarly for the DD character attribute.

After each complete cuboid (cuboid plus cuboid DD, or isolated character DD), exactly three DD's should follow: diagonal, vertical, and horizontal, in this order. Should any of them be empty, then the character '!' has to occur instead. Otherwise the DD's have to be prefixed by their corresponding dimensional terminals. They are the same ones explained in the editable grammar section.

As an example, the following linear coding:

@n @n cn!!! dndn!!!vnvn!!!hnhn!!! !!!

where the blanks have been inserted for readability reasons, corresponds to the DD:



APPENDIX I: PLOTTING DD'S WITH PIC

In order to draw DD's within documents, for illustrative purposes, the primitives given in this section could be used. All the illustrations within the present document have been plotted using them.

1. Primitives available.

The two primitives that should start any DD level are the *text* and *cubst* ones, for a string of characters and for starting a cuboid respectively. For the three edges one has the primitives: *diag*, *hor*, and *vert*. Also, there are three primitives to plot the iterators: *diaiter*, *horiter*, and *veriter* (see Section 7.). The cuboid DD has to be closed with the *cuboid* primitive, which defines the labels of the three continuation points of the DD. and the font and size of the cuboid rectangle.

As a general principle, sizes are not computed automatically. There are some general rules for this computation: the vertical dimension unit is the distance between lines, and the horizontal dimension the approximate width of eight characters. However, for text related horizontal displacements, character width units are used. A trial and error process has to be followed, but in practice and with some expertise a single iteration suffices.

Labels are attached to some points, for instance the cuboid continuation points already mentioned. Two other points to be labeled are the start of cuboids, *cubst*, and the text primitive. The last is optional. The primitive *backto* is provided for moving the pen to any textline starting point or cuboid continuation point.

The following is a brief description of the primitive parameters:

text(_). The text as such goes within the parentheses. When italics is desired, it could be enclosed within the mathematics delimiter.

cubst. Has no parameter, but a label should be attached to it, as in the example below.

cuboid(_,_,_,_,_,_,_). First parameter: the label of its corresponding *cubst*.

Second and third parameters: horizontal and vertical dimensions; the first one in units of four character withs each.

Fourth: the font of the cuboid rectangle: *dotted*, *dashed*, *invis*, or empty for normal.

Fifth, sixth, and seventh parameters: Three distinct labels for the diagonal, horizontal, and vertical continuation points of the cuboid respectively.

diag(_). Font of the edge, as in *cuboid*.

vert(_,_). Length of the edge in normal units, and its font, as in *cuboid*.

hor(_,_,_). Horizontal displacement, with respect to the current point (either a text or a cuboid horizontal continuation label), usualy zero for the cuboid case and the number of characters in the text for the text case. The second parameter defines the horizontal length of the line in *horinc* units (four characters approximately). The third gives the font as in *cuboid*.

horiter(_). Expects the horizontal displacement of the dotted line, as in *hor*.

backto(_). The only parameter gives the label of the text point or cuboid continuation point where the pen should move to.

The remaining primitives have no parameters.

2. Macro definitions

The following listing corresponds to the macro definitions of the above primitives and the actual settings of the dimensional variables. It should precede, together with the PIC starting line '.PS' , and the mathematics delimiter definitions, any PIC program to plot a DD. The ending line '.PE' should close the PIC code. It is not necessary to repeat the macro definitions and variable settings at each DD. It suffices to do it for the first picture and to enclose any DD picture code within the PIC starting and ending lines.

```
define text X line right invis
        "$1" at last line.w ljust X
define diag X line at last line.w + (tdx,-tdyd) down d right d $1
        move to last line.s + (-tdx,-tdyu) X
define hor X line at last line.w + ($1 * chwid, 0.0i) right $2*horinc $3
        move to last line.e + (0.03i,0.0)  X
define vert X line at last line.w + (tdx,-tdyd) down d + $1*lhigth $2
        move to last line.s + (-tdx,-tdyu) X
define backto X move to $1 ; line invis X
define horiter X line at last line.w + ($1 * chwid, 0.0) right d dotted
          "*" at last line.e + (0.06,-0.03i) ljust X
define veriter X line at last line.w + (tdx,-tdyd) down d dotted
          move to last line.s + (-tdx,-tdyu)
          line right invis
          "*" at last line.w + (0.01,-0.03) ljust X
define diaiter X line at last line.w + (tdx,-tdyd) down d right d dotted
          move to last line.s + (-tdx,-tdyu)
          line right invis
          "  *" at last line.w ljust X
define cubst X line right invis
         move to last line.w + (cdx,tdyu - cdy) X
define cuboid X move to $1 + (0.,tdyu - $3 * lhigth)
          line up  $3 * lhigth   $4
          line right $2 * horinc   $4
          line down $3 * lhigth   $4
          line left $2 * horinc   $4
$6: line at 2nd last line.n - (0.,tdyu)  right invis
$7: line at 2nd last line.w + (0.0,tdyd) right invis
$5: line at 3rd last line.w + (cdx,tdyd) right invis X
tdx = 0.02
tdyd = 0.07
tdyu = 0.09
cdx = 0.1
cdy = 0.15
d = 0.1
lhigth = 0.23
chwid = 0.059
horinc = 0.5
```

About the meaning of the dimension variables at the end, and as a general consideration, each text string has an invisible line reference along it. Its left point is a reference with respect to which the three possible edges continuing the DD are positioned.

tdx. Horizontal displacement between vertical edges and the text reference point. It also applies to the top

end of the diagonal edge, and to the vertical and diagonal iterators.

tdyu, tdyd. Up and down vertical sizes of text characters; with respect to text reference line.

cdx, cdy. Horizontal and vertical displacements of cuboid DD's. With respect to a cuboid starting point near the upper left corner of the cuboid rectangle.

d. Horizontal and vertical displacement for the diagonal edge.

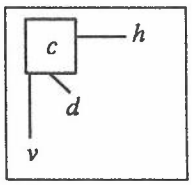
lhigth. Vertical unit of length. Corresponds to the distance between consecutive lines of text within a DD.

horinc. Horizontal length unit. Approximately eight character widths.

chwid. Character width. Some mean value.

3. An example

In order to plot the following DD :



the following PIC program could be used:

```
C1: cubst
C2: cubst
   text(#c#)
   cuboid(C2,0.5,1.2,,C2d,C2h,C2v)
backto(C2d)
diag()
text(#d#)
backto(C2v)
vert(1,)
text(#v#)
backto(C2h)
hor(0,0.5,)
text(#h#)
cuboid(C1,1.8,3.8,,C1d,C1h,C1v)
```

where the delimiters for equations have been set to the symbol '#' in order to obtain the italic font for text-lines. Explicit font changes can be made as well, as in `text("\fId\fp)`.

APPENDIX II: FORMAL SPECIFICATIONS AND DIMENSIONAL DESIGNS

The specification of the internal representation and operations of Dimensional Designs (DD's), a systematic technique for the graphic display of binary trees, is discussed. Different kinds of specification notations are experimented with; thus, grammatical, algebraic, and functional type notations are used. The specification has been one of the initial steps in the development of a syntax-driven editor generator of DD's.

The DD technique is also used for the graphical representation of the specifications themselves, in cases of poor readability due to expression length and parenthesis nestings. This serves as an illustration of the readability enhancement obtained with the application of DD's, and suggests its possible application to the specification area in general.

Grammars for the specification of DD's are introduced; they correspond to an extension of BNF type notation.

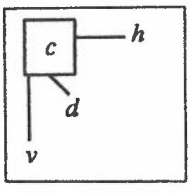
1. Introduction

The readability enhancement of many notations for formal specification is a widely recognised area where much needs still to be done. The problem appears especially in the case of large expressions, where their decomposition into smaller ones provides no satisfactory solution, since the unified or integrated view that the original large expression intended to convey is lost in its fragmentation.

One obvious way to improve readability is to represent expression abstract syntax trees. However, a systematic way for the layout of n-ary trees is needed. Dimensional Design (DD) [1],[2] is a systematic layout technique for arbitrary n-ary trees. It is very convenient to enhance presentation and readability of specifications, programs, expressions, etc... One of the purposes of this work is to provide some examples in this direction; thus, DD representations of data type definitions, terms and equations in formal specifications, etc.. will be given.

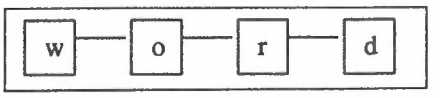
An earlier version of the dimensional technique has been used extensively in practice. An initial set of tools for the use of DD's for program development were developed at Rutherford Appleton Laboratories by R.W.Witty and D.A.Duce. They were extensively used in the development of a major real-time computerised telecontrol system at a power utility in Barcelona [3]. The first version of it was put in operation in 1982. Further tools have been developed by the telecontrol system development team, and the dimensional approach is still being used there with success, in connection with both maintenence and new function development.

Although a more formal definition of a DD will emerge gradually in this work, the following simple illustration may serve to introduce the concept informally:



The characters *c*, *d*, *v*, *h* stand either for themselves or for DD's. The rectangle represents a *cuboid*, whose interior may contain either a DD or a single character; there are two cuboids in the example. The three edges, diagonal (\), vertical (|), and horizontal (-) connect cuboids to their three DD sons. Since the sons of the outer cuboid of the example are empty, the corresponding edges are not represented; they could be if so desired.

The following DD, representing a text word, gives another illustration.



Usually, in text examples, cuboids and edges would be invisible and edges would have zero length. Thus, in general, lines and characters will have attributes.

When a DD is used, its edges should be given specific meanings; for instance, the horizontal edge in the last example could stand for a character connector operator. Much more about possible meanings for edges can be found in [2]. DD's with different sets of edge meanings will be used in this document.

The specification of some of the types needed for the construction of a syntax-driven editor of DD's will be commented on in this work. Various notations will be used, and sample large expressions in all of them will be represented as DD's. In this way, both the illustration of expression readability enhancement and the specification of the DD and other types needed in the editor will be accomplished.

2. Grammar specification of dimensional layout

DD's are diagrams to be represented in the two dimensional plane, therefore BNF type grammar notation would not be the best choice for the specification of their forms, to appear for instance on a screen; only a linearly coded representation of DD's could be specified in this form. One way to approach the specification is to introduce some new notation first, and then use it for the specification of DD's. It will be referred to as DD Grammar (DDG) notation, an introduction to an extended version of it is given in Section 7. However, the conventions needed for this work will be introduced at the appropriate points. Usually, the word *grammar* will denote a DD grammar in the present document.

A grammar (DDG) is represented itself as a DD. Its cuboid and edge lines are of two types: terminals and metasymbols. The former ones denote concrete structures of the DD being defined by the grammar. Metasymbol lines are represented in dotted form. There are metasymbols other than lines. Metasymbols are grammar auxiliary notation corresponding to the symbols `<`,`>`,`::=`, etc... in BNF.

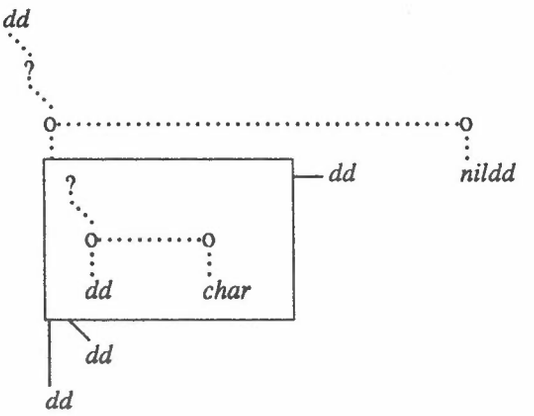
The selection, or choice, among alternate structures at a given point will be denoted by the metasymbol `?`, which will be diagonally connected to a structural dot `o` whose vertical DD son specifies the first alternative. The other alternatives are connected to their corresponding structural dots in the same manner; all the dots are connected horizontally. The following is an example of a selection

where *name* would denote a non-terminal of the grammar and the ai's would stand for dimensional structures (DD's) specifying the alternatives which may substitute the non-terminal. Non-terminals are represented in italics.

All edges in the above DD are metasymbols; the horizontal dotted edge denotes a connection operator among alternatives. The diagonal dotted edge denotes an equality definition relation of its upper DD in terms of its lower DD. For instance, *name* is defined as a selection; the symbol `?`, meaning *this* selection, is defined in terms of its alternatives as well. The vertical dotted connection has no special meaning in this case, it is used as a convenient way to separate the alternatives from the horizontal line.

Grammars specify in general subsets of DD's, for instance one may be interested in specifying with them

only the class of DD's representing the usual text; notice that there are DD's which do not represent text. Therefore, the motivation for DD grammars is the specification of the *form*, or *layout*, of sets (subsets) of DD's, in other words *languages* of DD's. The set of all DD's can also be specified by a grammar; the following is one way to do it



Non-terminals *char* and *nildd* would specify the set of valid characters and the empty DD respectively. A recursive definition is involved. As a convention, when a DD son connected to an edge is empty, the edge is not drawn. This grammar gives a visual picture of the structure of any DD and it will be used often in this document.

3. Internal representation of DD's

This section goes further into the description of DD's. The goal is here to obtain an internal representation of DD's, i.e. to construct an internal type DD. Auxiliary types are needed and introduced, they correspond to the parts of a DD which are displayed graphically in the grammar, but that have to be detailed in order to obtain a working internal type specification. For instance, the types *cuboid*, *diagcont* for diagonal continuation, *vertcont*, *horcont* correspond in the internal representation to the four DD's which are specified in the grammar as parts of a nonempty DD, together with their edge attributes. In addition, *fatherdd*, to denote the inverse of the DD son relation, and attributes such as character and line font are introduced as well.

A linear notation will be used for the definition of internal types and their associated operations. Translation to DD representation will be done from time to time, especially for long expressions, or to provide a more unified presentation of a fragmented set of expressions. Details of such a linear notation are reported in [4], but it will be explained as needed within this document. It will be referred to as functional type notation. The arrow (→) will denote definition equality. Juxtaposition of two types denotes their cartesian product type (record). The disjoint sum type (variant record, enumerated type) is denoted by the selection symbol `?`, the disjoint components are enclosed within parentheses and separated by the symbol `/`. A motivation for this notation is to highlight the similarities with BNF grammar notation. BNF defines sets (types) of strings. Here types of internal objects are defined. A linear model for the store will help to link the two cases.

The following is a possible internal representation of DD's.

type *(dd → ?(nildd / fatherdd cuboid diagcont vertcont horcont);*

cuboid → rectattr ?(cdd / ch cfont);

diagcont → dedgattr diagdd ;

vertcont → vedgattr vertdd ;

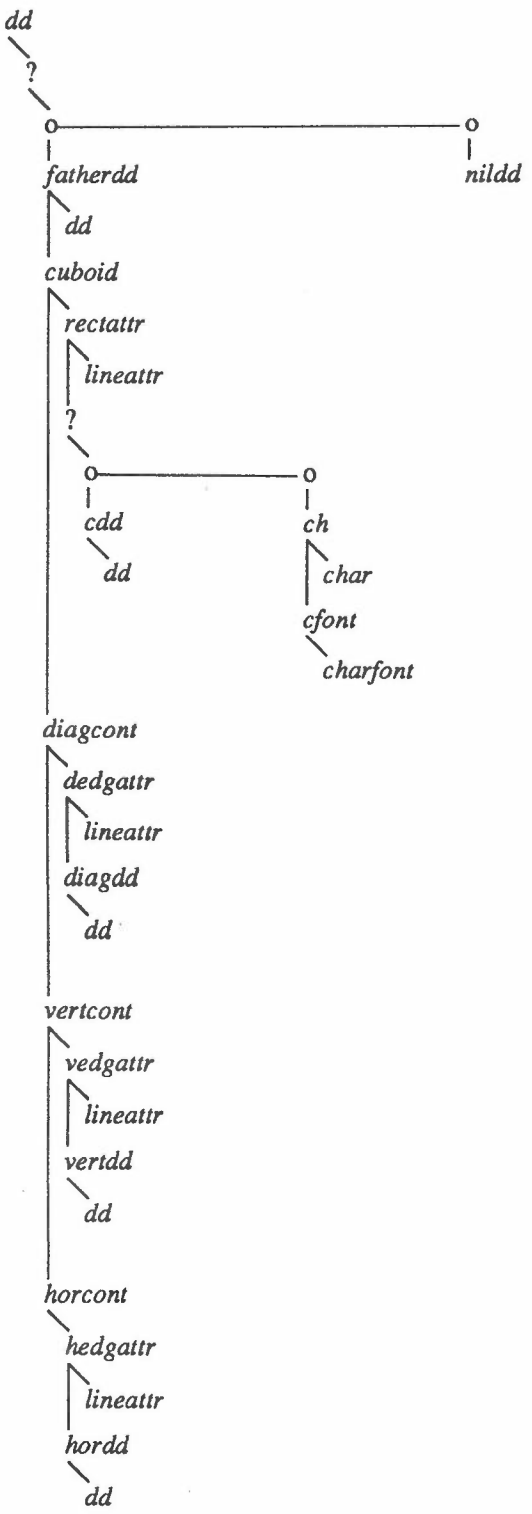
horcont → *hedgattr hordd ;*
fatherdd, cdd, diagdd, vertdd, hordd → *dd;*
rectattr, dedgattr, vedgattr, hedgattr → *lineattr ;*
ch → *char; cfont* → *charfont;)*

This type definition depends on the types *charfont*, and *lineattr*. A possible definition for them is the following

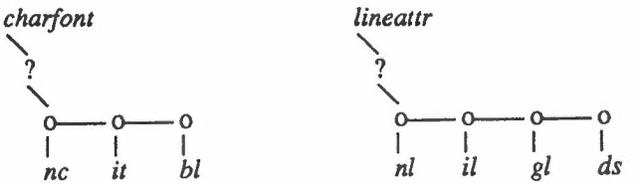
type *(lineattr* → *?(nl/il/gl/ds))*
type *(charfont* → *?(nc/it/bl))*

Notice that four line attributes are defined: normal, invisible, grammar line, and dashed ; also, the defined character fonts are the following three : normal, italics, and boldface. They are enough for the current purposes of representing grammars; more types can be introduced if needed. An invisible line attribute could be associated, for instance, to the cuboids of text characters.

DD notation can be introduced in order to obtain a more integrated form for the above type definition. Let the diagonal edge stand for definition equality, the vertical edge for cartesian product of types, and the horizontal one, in combination with the selection symbol '?', for the disjoint sum of types. Then the following is a DD presentation of the above type definition.



where the following types have been used



Finally, a possible translation into Pascal of the above DD type definition is the following

```
type dd = ^ddnode;
   hordesc = record hedgattr: lineattr; hordd: dd end ;
   vertdesc = record vedgattr: lineattr; vertdd: dd end ;
   diagdesc = record dedgattr: lineattr; diagdd: dd end ;
   cubdesc = record rectattr: lineattr;
             contents: record case ischar: boolean of
                         false:( cdd: dd );
                         true:( ch: char;
                                cfont: charfont
                              )
                       end
         end ;
    ddnode = record fatherdd: dd;
             cuboid: cubdesc;
             diagcont: diagdesc;
             vertcont: vertdesc;
             horcont: hordesc;
    end ;
```

Notice that new auxiliary names have been introduced, such as *cubdesc*, *diagdesc*, *vertdesc*, and *hordesc*.

4. Some simple operations for the internal construction of DD's

A dynamic, rather than structural or static, view is taken in this section. The type DD is considered now to be a set whose elements will be DD's but which is initially empty. The operations which inject elements into this set will be defined. The valid DD's will be the results of such operations only. Both functional and algebraic specifications will be used. The underlying concepts for the algebraic specifications of this and the following sections may be found in [5],[6]. An interesting application of algebraic specifications to graphics is reported in [7].

4.1. Make nil DD

The first operation has no parameter, its result is the empty DD; i.e. it would correspond to the second alternative of the global selection of the grammar given earlier in this document for the definition of a DD. Since the operation has no input, its result is a constant; it is a nullary function, a function without parameters (arguments). The following notation specifies the parameters and results of this operation

mnildd → **res** (*ddr* → *dd*) **par** *()*

This would correspond to the signature of the operation, which contains the name of the operation and the parameter (domain) and result types. The arrow means definition equality in the notation. A general operation would be defined in terms of parameters, results, and either a constructive or an equational definition body. Since there are no parameters the construct ` **par** *()*` could have been omitted. In the present case, the result type *ddr*, is defined to be equal to *dd*. A similar thing will be done for parameter types. The names of these new types introduced in the parameter and result type definition will be used as variables within the definition body, to denote either the corresponding parameters or result. This is not required in the case of *mnildd* since the function is a nullary function, i.e. a constant, needing no definition body. It is hoped that this form of introducing variables causes no confusion. Whenever a type needs to be denoted within a definition body, which is not the case for the present work, an explicit indication will be used. Further details on the notation are given in [4].

4.2. Make character DD

A character within a cuboid is the simplest non-empty DD structure, as specified by the above grammar. The operation which constructs such a DD, denoted as *mchardd*, will have as parameters a character, and a character attribute, defined as follows

type (*charattr* → *charfont lineattr*)

The following is the parameter and result specification for the operation

mchardd → **res** (*ddc* → *dd*) **par** (*cd* → *char* ; *chat* → *charattr*)

Now the operation has a nonempty parameter list. Here is an example of an invocation to it

mchardd (cd0, chat0)

where the variables hold values of the respective types. The whole expression would be of type DD.

4.3. Make character headed DD's

The following three operations, paralleling the construction of lists from heading atoms and lists, could be considered in principle

mchdddd → **res** *(ddc → dd)* **par** (*ddd* → *dd; cd* → *char; chat* → *charattr; lat* → *lineattr*)

mchdvdd → **res** (*ddc* → *dd)* **par** (*ddv* → *dd; cd* → *char; chat* → *charattr; lat* → *lineattr*)

mchdhdd → **res** (*ddc* → *dd)* **par** (*ddh* → *dd; cd* → *char; chat* → *charattr; lat* → *lineattr*)

They construct the DD which consists of a heading character DD with either a diagonal, vertical, or horizontal DD son, the other remaining two DD sons being empty.

Notice that the operations informaly defined up to now allow the construction of DD's whose cuboids enclose characters only, and which are linear in structure; i.e. their character DD's have only one DD son. An arbitrary DD cannot be constructed yet.

5. Some operations to analyse DD's

Before going into the complete set of operations to construct DD's, which is still needed, let us introduce some operations which construct no DD, but whose purpose is to analyze the parts of already constructed DD's. The main intention is not to define them formally, since the set of constructors to be introduced in the next section would be needed for that, but to introduce some of their properties only. In doing so, we can rely, for the moment, on the DD representation given in Section 6.3.

Some *boolean* operations come first. In order to test whether a DD is empty the following function may be invoked:

isnildd → **res** (*b* → *boolean*) **par** (*ddc* → *dd*)

For a nonempty DD, the following function will take the value true if its (heading) cuboid contains a character only:

ischardd → **res** (*b* → *boolean*) **par** (*ddc* → *dd* ; ¬ *isnildd (ddc)*)

Observe the notation to formulate the nonemptyness requirement for the parameter.

Equations will be introduced now in order to formulate properties holding on operations. They will state the equality among two terms, i.e. two expressions involving the operations introduced so far and variables. As stated earlier, the names of types used in the equations will denote variables of the corresponding types, which will be assumed universally quantified. Equality among boolean, character, and other simple types will be used without definition. Later, equality between DD's will be defined as a boolean operation on DD's. The following are equations formalising some properties already noted:

isnildd (mnildd) = true

ischardd (mchardd (cd, chat)) = true

The following also holds:

ischardd (mchdddd (ddd, cd, chat, lat)) = true,

and similarly for *mchdhdd* and for *mchdvdd*. In fact, for all nonempty DD's constructed with the operations given so far, *ischardd* would take the value true.

Some *selector* operations will be given next; they take the value of the components of a DD. In order to access a character within a DD, one has the operation

ddchar → **res** (*c* → *char*) **par** (*ddch* → *dd* ; ¬ *isnildd(ddch) ischardd(ddch)*)

Notice that juxtaposition denotes logical conjunction in the logic expression.

For accessing the diagonal, vertical, and horizontal DD sons of a nonempty DD, the following selector operations are provided

diagdd → **res** (*ddn* → *dd*) **par** (*ddd* → *dd* ; ¬ *isnildd(ddd)*)

vertdd → **res** (*ddn* → *dd*) **par** (*ddv* → *dd* ; ¬ *isnildd(ddv)*)

hordd → **res** (*ddn* → *dd*) **par** (*ddh* → *dd* ; ¬ *isnildd(ddh)*)

The value of these functions is the empty DD when the corresponding DD son is empty.

Since, with the operations given so far, a DD within a cuboid cannot be constructed, the selection operation whose result is the cuboid DD could not be applied yet. However, since a set of constructors will be given in the next section, this fourth DD son selector operation, *cubdd*, can be introduced now in advance

cubdd → **res** (*ddn* → *dd*) **par** (*ddc* → *dd* ; ¬*isnildd(ddc)* ¬*ischardd(ddc)*)

Note the formulation of the above restriction on the parameter. The result may be the empty DD.

Here are some equations formulating some of the above properties

isnildd(diagdd(mchardd(cd, chat))) = true

isnildd(diagdd(mchdddd(ddd, cd, chat, lat))) = false

isnildd(diagdd(mchdvdd(ddv, cd, chat, lat))) = true

and similarly for *vertdd*, and *hordd*.

The following equation

diagdd(mchdddd(ddd, cd, chat, lat)) = ddd

and the parallel ones for *mchdvdd* and *mchdhdd* involve equality between DD's, which has not been defined yet. In order to do so, the following selectors for the character and edge attributes are needed

ddcrattr → **res** (*chat* → *charattr*) **par** (*ddch* → *dd* ; ¬ *isnildd(ddch) ischardd(ddch)*)

cubattr → **res** (*lat* → *lineattr*) **par** (*ddc* → *dd* ; ¬*isnildd(ddc)*)

degattr → **res** (*lat* → *lineattr*) **par** (*ddd* → *dd* ; ¬*isnildd(ddd)*)

and similarly for *vegattr* and *hegattr*. The following is a functional definition of equality among DD's

which assumes that equality predicates among line attributes and among characters and character fonts have been defined.

```
eqdd → res ( b → boolean ) par ( dd1, dd2 → dd )
    ?( isnildd(dd1) isnildd(dd2)
    / (¬isnildd(dd1)) (¬isnildd(dd2))
      ?( ischardd(dd1) ischardd(dd2)
         ( ddchar(dd1) = ddchar(dd2) )
         ( ddcrattr(dd1) = ddcrattr(dd2) )
      / (¬ischardd(dd1) ¬ischardd(dd2))
         eqdd( cubdd(dd1), cubdd(dd2) )
      ) ( cubattr(dd1) = cubattr(dd2) )
      eqdd( diagdd(dd1), diagdd(dd2) ) ( degattr(dd1) = degattr(dd2) )
      eqdd( vertdd(dd1), vertdd(dd2) ) ( vegattr(dd1) = vegattr(dd2) )
      eqdd( hordd(dd1), hordd(dd2) ) ( hegattr(dd1) = hegattr(dd2) )
    )
```

This is the first complete definition using a functional type notation,[4]. A logical expression using the same conventions which have already been used in earlier parameter specifications constitutes the main body of the definition. Its value is the value of the result. It depends on the definitions of the operations introduced earlier rather informally.

6. A set of constructors

The operations to form linear DD's : *mchdddd*, *mchdvdd*, *mchdhdd* are now going to be replaced by a new set, referred to as a set of *constructors*. It will be a complete set in the sense that any DD which could be generated with the grammar given earlier, could also be generated by the application of operations of the set. Therefore, a correspondence between the elements of the grammar and the new constructors has to be established.

The first two members of the set have already been defined: *mnildd* and *mchardd*. They correspond to the second (right) alternatives of the global and cuboid selections of the grammar respectively. Therefore, only the four operations corresponding to the four references to *dd* need to be introduced. The cuboid rectangle and the three solid edges in the grammar will be associated to one operation each.

Given a DD and line attributes, the constructor *mcubdd*

mcubdd → **res** (*ddn* → *dd*) **par** (*ddc* → *dd* ; *lat* → *lineattr*)

gives as result a new DD consisting of the given DD enclosed within a cuboid with the given line attributes. The following equations can now be established :

cubdd(mcubdd(ddc, lat)) = ddc

cubattr(mcubdd(ddc, lat)) = lat

The following also holds

isnildd(diagdd(mcubdd(ddc, lat)) = true

and similarly for *vertdd* and *hordd*. In addition one has that

isnildd(mcubdd(ddc, lat)) = false

ischardd(mcubdd(ddc, lat)) = false

The remaining three operations attach a DD son to a cuboid or character DD having an empty diagonal, vertical or horizontal DD son respectively

mdiagdd → **res** (*ddn* → *dd*) **par** (*ddg* → *dd; isnildd(diagdd(ddg)); ddd* → *dd; lat* → *lineattr*)

mvertdd → **res** (*ddn* → *dd*) **par** (*ddg* → *dd; isnildd(vertdd(ddg)); ddv* → *dd; lat* → *lineattr*)

mhordd → **res** (*ddn* → *dd*) **par** (*ddg* → *dd; isnildd(hordd(ddg)); ddh* → *dd; lat* → *lineattr*)

Here are some properties :

diagdd(mdiagdd(ddg, ddd, lat)) = ddd

degattr(mdiagdd(ddg, ddd, lat)) = lat

and the parallel equations for *mvertdd* and *mhordd*.

The following expression takes the value of a DD and involves the complete set of constructors, its components can be put in a one to one correspondence with the components of the grammar. Its variables *ddc*, *ddd*, *ddv*, and *ddh* denote the four DD sons; *latc*, *latd*, *latv* and *lath* correspond to the cuboid rectangle and the three solid edges; *cd* and *chat* correspond to the character

?(mhordd(mvertdd(mdiagdd(?(mcubdd(ddc,latc) / mchardd(cd,chat)),ddd,latd),ddv,latv),ddh,lath)

/ mnildd)

The cuboid rectangle attributes of a cuboid containing a character are implicit in *chat*. Any DD can be constructed now by proper application of the constructors.

The above expression defines an order for the construction of DD's, which starts from its innermost level. First, a cuboid with either a DD or a character within it is constructed; then a diagonal son is appended to it, continuing with a vertical, and finishing with a horizontal son. This will be referred to as *canonical order*, and the corresponding expression *canonical term*.

Observe that there are other orders for constructing the same DD. This fact may be expressed with equations. Let *ddd* denote a DD such that

¬isnildd(ddd) isnildd(vertdd(ddd)) isnildd(hordd(ddd))

then one has that

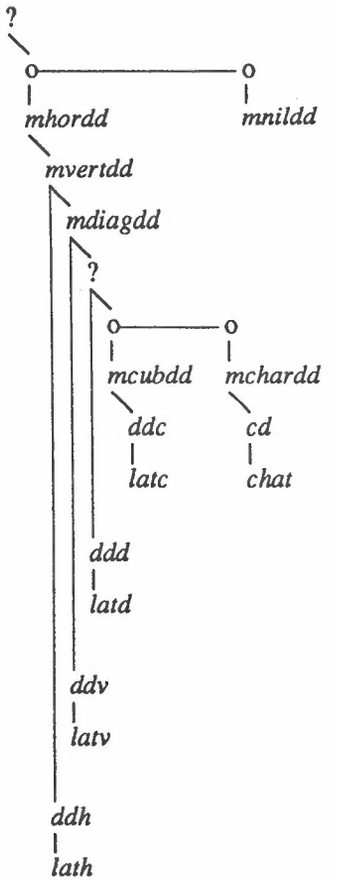
mvertdd(mhordd(ddd, ddh, lath), ddv, latv) = mhordd(mvertdd(ddd, ddv, latv), ddh, lath)

which expresses the equivalence. Two parallel equations involving the operation pairs *mhordd*, *mdiagdd* and *mvertdd*, *mdiagdd* can be established as well. Then, given a term involving the constructors, these three equations can be used as rewrite rules to obtain an equivalent canonical term, since the constructors appear in canonical order in their right hand sides. A term subexpression matching a left hand side would be substituted by the corresponding right hand side.

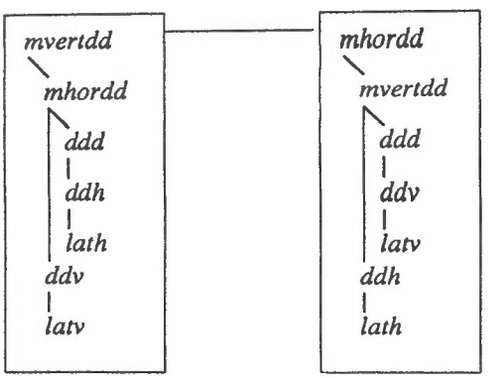
7. DD representation of terms and equations

The readability of terms and equations may be enhanced if they are representd as DD's. Many possibilities for such a representation exist, depending on the meanings assigned to the edges. One of them interprets the vertical edge as denoting a `brother` relation; where actual parameters (operands) of an operation are considered to be brothers. Furthermore, the horizontal edge would still connect alternatives for a parameter occurrence, in combination with the symbol `?` standing at the parameter location. The diagonal edge would connect the operation name (operator) to its first operand (son).

The following is a DD representation of the term which involved, above, the complete set of constructors, in accordance with the given interpretation for edges.



Finally, in order to express equality among terms, they will be enclosed within cuboids, and the cuboids will be connected with an horizontal edge which, outside cuboids, will denote equality among terms. As an illustration, the last equation of the last section could be represented as follows:

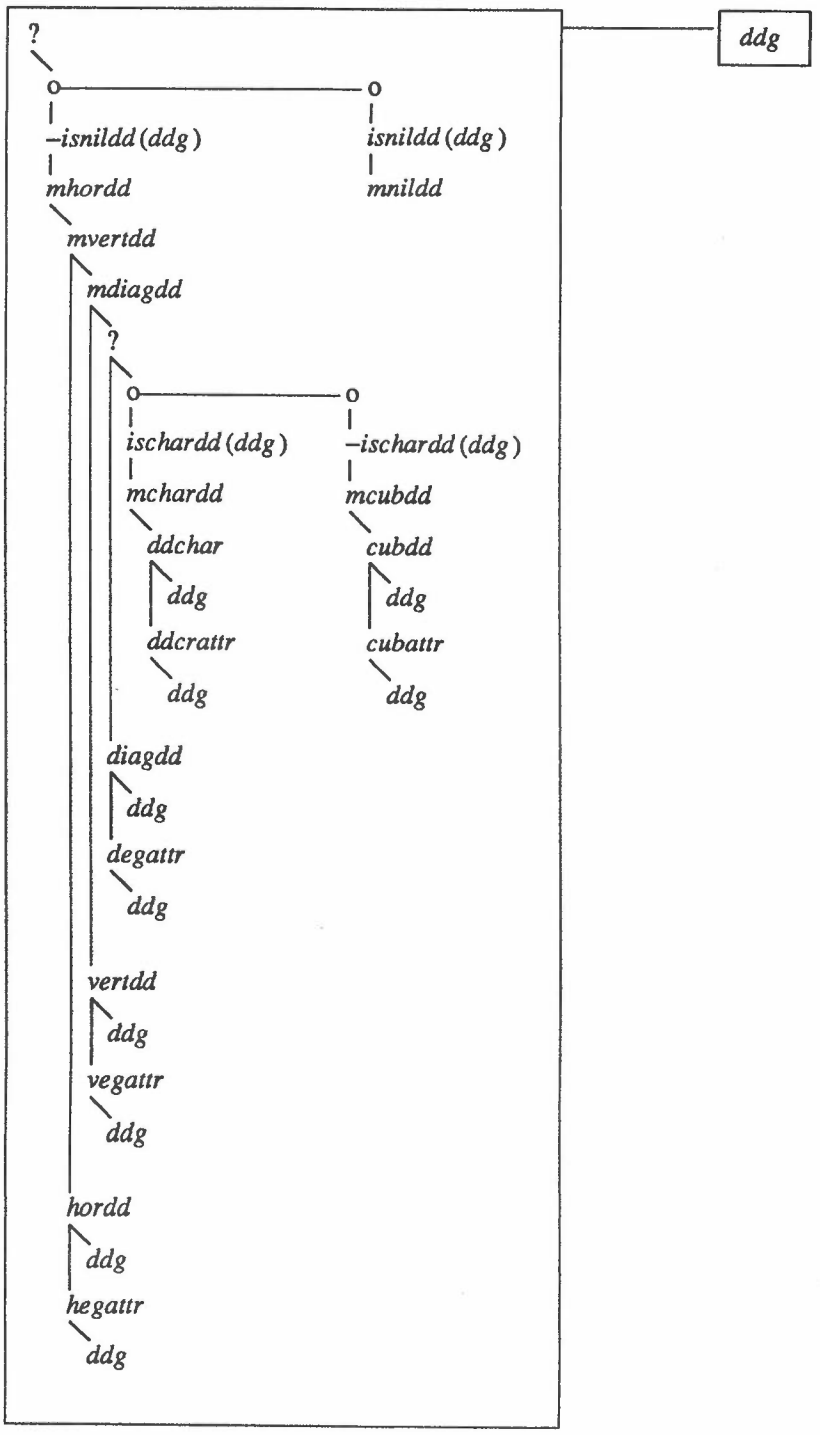


As it can be appreciated, readability is enhanced considerably; at the expense, however, of an increase in paper space.

An interesting equation involving the operations introduced so far, with the exception of *mchdddd*, *mchdvdd*, *mchdhdd* which were already discarded, is the following

```
?( ¬isnildd(ddg)  mhordd( mvertdd( mdiagdd(
        ?( ischardd(ddg) mchardd(ddchar(ddg),ddcrattr(ddg))
         / ¬ischardd(ddg) mcubdd(cubdd(ddg),cubattr(ddg) ) )
      , diagdd(ddg), degattr(ddg) ), vertdd(ddg), vegattr(ddg) ), hordd(ddg), hegattr(ddg) )
/ isnildd(ddg) mnildd )   =  ddg
```

It expresses the inverse annihilation between suitable pairs of operations. The same expression can be presented more legibly as



Notice that the meaning of the vertical edge has been extended to the case where a boolean operator stands at its upper end as the upper item of an alternative. Should its value be true, then the construct is equivalent to the DD at the lower end of the vertical edge; otherwise, the DD is replaced by the empty DD, and the alternative is not selected.

8. DD points

Some DD operations, still to be defined, will have to be of `random access` type. They will have an `address` within a DD as parameter; or, more technicaly, a *DD occurrence point*, more succinctly a *DD point*. This section introduces the specification of this concept as a type, *ddpoint*. It has five constructors

nilpt → **res** *(ddpt → ddpoint)*

cpt → **res** *(ddpt → ddpoint)* **par** *(ddpt0 → ddpoint)*

and the three constructors, *dpt*, *vpt*, and *hpt*, having the same type of signature as *cpt*.

The boolean operation

isnilpt → **res** *(b → boolean)* **par** *(ddpt → ddpoint)*

such that *isnilpt(nilpt) = true*, and taking the value *false* otherwise, decides whether a point is *nilpt*.

Equality among DD points can be defined as follows:

eqddpt → **res** *(b → boolean)* **par** *(ddpt1, ddpt2 → ddpoint)*

with the equations

eqddpt(ddpt1,ddpt2) = eqddpt(ddpt2,ddpt1)

eqddpt(nilpt,ddpt) = isnilpt(ddpt)

eqddpt(cpt(ddpt1),cpt(ddpt2)) = eqddpt(ddpt1,ddpt2)

with three similar equations for *dpt*, *vpt*, and *hpt*. The remaining twelve combinations would give a *false* value.

A DD point may denote a part of a given DD, i.e. a sub DD. The function *pointed DD*

ptddd → **res** *(ddp → dd)* **par** *(ddg → dd ; ddpt → ddpoint)*

gives such a part. It is defined as follows :

ptddd(ddn,nilpt) = ddn

¬isnilpt(ddpt) (ptddd(mnildd,ddpt) = undefined)

¬isnilpt(ddpt) (ptddd(mchardd(ch,chat),ddpt) = undefined)

ptddd(mdiagdd(ddn,ddd,lat),dpt(ddpt)) = ptddd(ddd,ddpt)

and similarly for the pairs *(mvertdd, vpt)*, *(mhordd, hpt)*, and *(mcubdd, cpt)*.

Lists of DD points will be needed as well. A new type, *ddptlist*, will be introduced for them. The operation *cons* will give the list obtained by appending a given list to a given point. The operation *mnilddpl* will take the value of the empty DD point list. Also, *mlist* will put together two lists to form a new one.

9. Searching pruning and attaching DD's

The first operation to involve points searches for a DD, *dds*, within another DD, *ddg*.

searchdd → **res** *(ddplst → ddptlist)* **par** *(ddg, dds → dd)*

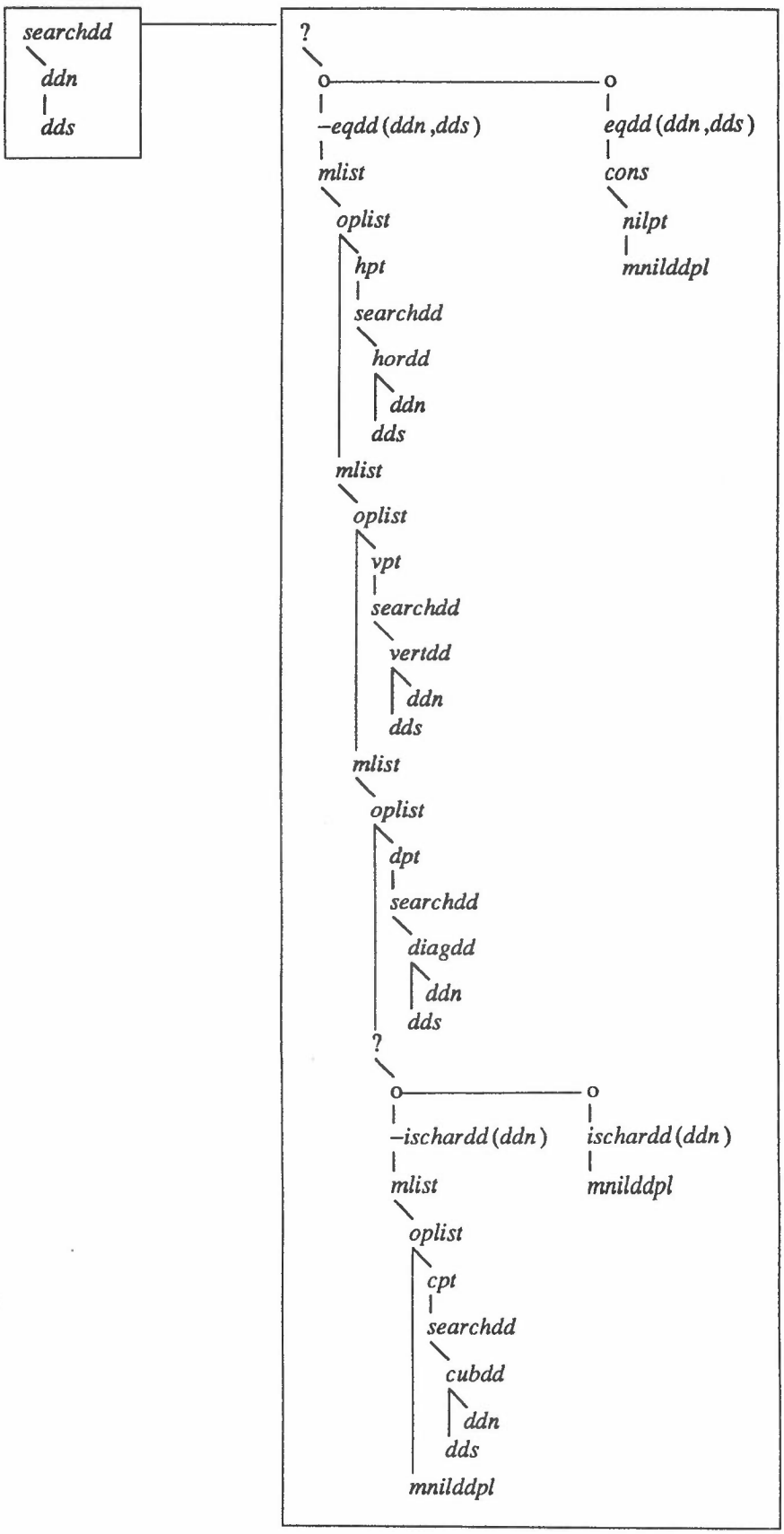
It takes the value of the list of all the points of *ddg*, at all of which *dds* occurs within *ddg*. The following equations define this operation

¬isnildd(dds) (searchdd(mnildd,dds) = mnilddpl)

searchdd(dds,dds) = cons(nilpt, mnilddpl)

¬eqdd(ddn,dds) (searchdd(ddn,dds) = mlist(oplist(hpt,searchdd(hordd(ddn),dds)),
mlist(oplist(vpt,searchdd(vertdd(ddn),dds)),
mlist(oplist(dpt,searchdd(diagdd(ddn),dds)),
?(ischardd(ddn) mnilddpl
/¬ischardd(ddn)
mlist(oplist(cpt,searchdd(cubdd(ddn),dds)),mnilddpl)))))

The last two equations can be expressed in DD notation:


searchdd
ddn
dds
?
o
o
–eqdd (ddn ,dds)
eqdd (ddn ,dds)
mlist
cons
oplist
nilpt
hpt
mnilddpl
searchdd
hordd
ddn
dds
mlist
oplist
vpt
searchdd
vertdd
ddn
dds
mlist
oplist
dpt
searchdd
diagdd
ddn
dds
?
o
o
–ischardd (ddn)
ischardd (ddn)
mlist
mnilddpl
oplist
cpt
searchdd
cubdd
ddn
dds
mnilddpl

where the function *oplist* has been assumed to exist. It gives the list resulting from the application of its first argument to each element in the list at its second argument. Also, the lastly used conventions about edge meanings have been used.

The list of *terminating points* of a DD, say *ddg*, is defined as *searchdd(ddg, mnildd)*. The empty DD has only one terminating point, *nilpt*.

Now, the operation to prune DD's can be introduced. Given a nonempty DD and a point within it, the function

mprundd → **res** *(ddn → dd)* **par** *(ddg → dd ; ¬isnildd(ddg); ddpt → ddpoint)*

takes the value of the DD equating the given one except at the given point, where the empty DD occurs now. The following equations define the new operation

mprundd(ddn,nilpt) = mnildd

mprundd(mnildd,ddpt) = mnildd

¬isnilpt(ddpt) (mprundd(mchardd(ch,chat),ddpt) = mchardd(ch,chat))

mprundd(mdiagdd(ddn,ddd,lat),dpt(ddpt)) = mdiagdd(ddn,mprundd(ddd,ddpt),lat)

and a similar equation for the cuboid, vertical, and horizontal cases. The following will hold as a consequence, wherever *mprundd* is defined.

ptdd(mprundd(ddn,ddpt),ddpt) = mnildd

Finally, an opperation to attach a DD at a terminating point of another DD can be introduced

mattachdd → **res** *(ddn → dd)* **par** *(ddo,ddnw → dd ; ddpt → ddpoint)*

which could be defined recursively in a similar manner as *mprundd*. The following property will hold, wherever *mattachdd* is defined

mprundd(mattachdd(ddo,ddnw,ddpt),ddpt) = ddo

APPENDIX III : SOME FURTHER FUNCTIONS

Some comments about functions, which are not yet available but which are either desirable or needed, are given in this section.

1. Detach and attach parts of syntax-trees.

They are required in order to implement the delete operation. A complete version of this function needs temporary storage of parts of syntax trees for their later attachement, for instance when a part of a DD which is not a subtree has to be deleted.

In order to preserve syntactic correctness, the only valid detachement points are the non-terminals, primitive or not. Their names are left unexpanded in the syntax tree, but are copied at the head of the detached parts. These parts are labelled with the name of the grammar, which happens to be the heading non-terminal in the syntax tree. Attachment is done in the obvious inverse way, checking both grammar and expanded non-teminal names.

One way to proceed in the implementation is to introduce a new type for a detached syntax tree, type *detstree = dd;*. It could be implemented as a DD in order to reuse the already available DD operations, especially the input-output ones. For instance, the grammar name could stand as a non-teminal DD as the cuboid, and its vertical DD, connected with a grammar edge, could contain the non-terminal continued with a diagonal DD, consisting of the dettached DD.

The following could be the two function signatures in Pascal notation:

function mdetchst(ddginst:ddgram;stinst:ddgstree;nontinst:ddgstree): detstree;

function mattchst(ddginst:ddgram;stinst:ddgstree;nontinst:ddgstree;newpart:detstree): ddgstree;

By properly applying these functions, many global functions for the user interface can be constructed, and the syntactic correctness of the implied DD will be preserved. Like the following ones: delete, add, insert, copy, move, permanent storage of parts, etc...

2. Expansion of primitive non-terminals.

Something about this topic has already been mentioned in the introduction. When such an expansion has to take place, the expansion DD should be constructed by a built-in function. For instance, the *:tex* primitive has to be expanded by first invoking the *intxtln* input function of the abstract type *textln*. The invocation should be done by the interactive program once it has detected that the user wants to expand an instance of such a non-terminal. The user will then be expected to input the textline from the keyboard, a carriage return character will be interpreted as the end of the text line.

At this point, the internl DD representation of the line of text is available. It has to be attached as the diagonal DD of the unexpanded non-terminal in the syntax tree, with a grammar line attribute for the diagonal edge.

The primitive non-terminal *:ide* should be expanded with the function invocation *inidstr(normid)*. The value used for its selector type parameter will have the effect of interpreting the character string, input by the user, as an identifier; any non-identifier character will be interpreted as its end character.

The *:nil* case involves no function for the construction of internal DD's. Other primitive non-terminals could be added. For instance, since the functions *intermnl*, *inidstr(noterm)*, and *inprmnt* are available, the following primitive non-terminals *:terminal*, *:nonterminal*, and *:primnonterminal* could be incorporated. They are going to be used in the next subsection.

3. Guided editing of grammars.

Up to this point, the grammars for the generator should be edited with a normal text editor with the format explained in Section 2. However, it may be tempting to try to define the metalanguage with a grammar and, by feeding it to the generator, obtain a syntax driven editor for the grammars themselves.

The main lines for the introduction of this new function are covered now. A problem is the ambiguity between grammar lines of the grammar defining grammars and grammar lines of the grammar which is being edited with the syntax driven editor. A way to overcome the ambiguity is to use a different line font

for the latter, like the dashed line font.

Once the editing process is complete, the grammar is extracted from its syntax tree. Such a grammar cannot be used to drive the syntax driven editor due to the dashed lines. However, a suitable procedure can be coded for their change to grammar lines. A level of indirection in the representation of fonts can avoid this process.

Since procedures for inputing terminal and non-terminal names are available within the type *textln*, the addition of the primitive non-terminals *:ter*, *:non*, and *:pri*, as detailed in the last subsection, will make the coding of the grammar defining grammar shorter. The following is a possible one:

```
<grammar>
   = (
       <rule>
       vd <rule_list>
         e
       e
     )

<rule_list>
  r = ?(
       <rule>
       vd <rule_list>
         e
       e
     /
       :nil
       e
     )

<rule>
   = (
       :non
       dd <rule_body>
         e
       hd <recursive_option>
         e
       e
     )

<recursive_option>
   = ?(
       'r'
       e
     /
       :nil
```

```
        e
      )

<rule_body>
    = ?(
        <rule_alternative>
        e
      /
        <selection_body>
        e
      )

<selection_body>
    = (
        '?'
        dd <alternative>
          hd <more_alts>
            e
          e
        e
      )

<alternative>
    = (
        'o'
        vd <rule_alternative>
          e
        e
      )

<more_alts>
  r = ?(
        <alternative>
        hd <more_alts>
          e
        e
      /
        :nil
        e
      )

<rule_alternative>
  r = ?(
```

```
  :non
  vn <rule_alternative>
    e
  hn <rule_alternative>
    e
  e
/
  :pri
  vn <rule_alternative>
    e
  hn <rule_alternative>
    e
  e
/
  :ter
  dn <rule_alternative>
    e
  vn <rule_alternative>
    e
  hn <rule_alternative>
    e
  e
/
  cn <rule_alternative>
    e
  dn <rule_alternative>
    e
  vn <rule_alternative>
    e
  hn <rule_alternative>
    e
  e
)
```

It has been printed with the standard tabulation implicit in the *outddgrm* procedure.

4. Extraction of sentences from syntax trees.

This is a required function, since the user is ultimately interested in a DD rather than in its syntax tree. A draft of a Pascal function for the extraction of a DD sentence from a syntax tree will be given. The main intention is to convey an initial idea of the algorithm for its further improvement.

The function, *sentence*, will extract the sentence from a syntax tree without destroying it, and assuming that it contains no undecided selections. However, some of its non-terminals may not be expanded.

The main idea is to process the syntax tree recursively. The sentences of the DD sons are obtained first,

when this makes sense. Then the sentence of the syntax tree is obtained depending on the type of the current level: terminal, non-terminal, or cuboid. This is done by properly connecting the sentences and forgetting about the non-terminal.

The following is a sketch for the function:

```
sentence(stinst:ddgstree): ddgsentn;
 var cuboiddd: dd;  diagsent,vertsent,horsent: ddgsentn;  cubedg,diagedg,vertedg,horedg: lineattr;
 begin
  if isnildd(stinst) then  sentence := stinst
   else if ischardd(stinst) then {error}
    else
     begin
      cuboiddd := cubdd(stinst); cubedg := cubattr(stinst);
      diagsent := sentence(diagdd(stinst)); diagedg := degattr(stinst);
      vertsent := snetence(vertdd(stinst)); vertedg := vegattr(stinst);
      horsent := sentence(hordd(stinst)); horedg := hegattr(stinst);
      if { stinst is a grammar terminal DD }
       then  sentence := mhordd(mvertdd(mdiagdd(copydd(mcubdd(cuboiddd,cubedg))
                                                ,diagsent,diagedg)
                                        ,vertsent,vertedg)
                               ,horsent,horedg)
        else if { stinst is non-terminal, primitive or not } then
          begin
           if isnildd(diagsent) then          { unexpanded }
              sentence := mhordd(mvertdd(copydd(mcubdd(cuboiddd,cubedg))
                                                ,vertsent,vertedg)
                                       ,horsent,horedg)
            else { form sentence by attaching vertsent at the
                   first empty vertical DD of diagsent, and
                   attaching  horsent at the first empty
                   horizontal DD of the resulting DD.     }
          end
        else  { stinst is a terminal cuboid DD. }
         sentence := mhordd(mvertdd(mdiagdd(mcubdd(sentence(cuboiddd),cubedg)
                                                ,diagsent,diagedg)
                                        ,vertsent,vertedg)
                               ,horsent,horedg)
     end
 end
```

ACKNOWLEDGEMENTS

Many people have contributed directly or indirectly to the work reported in this document. I would like to thank all of them: Dr R W Witty for inviting me to spend my sabbatical at RAL, and for his comments,

criticisms, and corrections. Tom Povey for his criticisms and encouraging comments on the work. Dr D A Duce for his reading and encouragement on parts of this work. A J J Dick for reading and correcting some sections. D Gibson for his interest in incorporating the library of functions into his man machine interface system.

BIBLIOGRAPHY

[1] R W Witty, "Dimensional Flowcharting", *Software Practice and Experience*, vol 7, pp 553-584, 1977.

[2] R W Witty, *Small Scale Software Engineering*, PhD Dissertation, Brunel University, Dept of Computer Science, September 1981.

[3] M Bertran and J Xampeny, "A Computerized Power Network Telecontrol Center: Environment and Solution Framework", *IEEE Power Engineering Society Summer Meeting*, Vancouver, Canada, July 1979.

[4] M Bertran-Salvans, "System rules: A Linguistic Approach to Distributed System Design", Research Report, Escola T Superior E Telecomunicacio, UPC, Barcelona, Spain, March 1986.

[5] J A Goguen, J W Thatcher, E G Wagner, "An Initial Algebra Approach to the Specification, Correcness, and Implementation of Abstract Data Types". In Raymond T Yeh (Ed), *Current Trends in Programming Methodology*. Vol. IV, Data Structuring, Prentice-Hall, 1978, pp 80-149.

[6] G Huet, D C Open, *Equations and Rewrite Rules, A Survey*. Stanford Verification Group, Computer Science Department, Rep. No. STAN-CS-80-785, 1980.

[7] D A Duce, E V C Fielding, *Formal Specification - A Comparison of Two Techniques*. Rutherford Appleton Laboratory, Chilton, Didcot, UK, RAL-85-051, July 1985.

[8] Dimensional Design, Requirements Specification for Prototype, issued by D R Gibson, SE Group note 111, RAL Informatics Division, revision April 1986.