stichting
mathematisch
centrum

AFDELING INFORMATICA IW 68/77 OKTOBER
(DEPARTMENT OF COMPUTER SCIENCE)

T. HAGEN, P.J.W. TEN HAGEN, P. KLINT &
H. NOOT

# ILP
# INTERMEDIATE LANGUAGE FOR PICTURES

(Preliminary report)

2e boerhaavestraat 49 amsterdam

*Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.*

*The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).*

---

AMS(MOS) subject classification scheme (1970): 68K30, 68T35

---

ACM-Computing Review-categories: 8.2

# ILP

# Intermediate Language for Pictures

by

T. Hagen, P.J.W. ten Hagen, P. Klint & H. Noot

ABSTRACT

The Intermediate Language for Pictures (ILP) determines the structure of an interactive graphics system, in which pictures are represented as ILP programs.
ILP contains elementary drawing actions, state information that can modify actions, structuring functions that combine state and actions in a directed graph structure and a mechanism for external referencing.
The semantics of ILP are described very precisely.
ILP is structured in such a way (mainly by means of the new concept of attributes) that it can be extended for special application area's and for interactive use.

KEYWORDS & PHRASES: Computer graphics,
Graphic language,
Picture representation.

This report has been produced on an HRD-1 Laser Display/Plotter.

CONTENTS

# 1. INTRODUCTION

## 1.1. The kernel of an Interactive Graphics System

The language defined in this report is a special purpose programming language. The restriction implied by the term special purpose is twofold. First of all, the language only serves the description of pictures *). Every construction in the language is justified by the requirement that it should cover a part of this descriptive function. Useful constructs that might have been added because of its function as a programming language have been omitted. The second restriction is derived from the fact that the language is an 'intermediate' language. This means that its second function is to fill in the gap that exists between a picture description in the form of instructions for a physical drawing machine on one side and a picture description as part of a more sophisticated language or data structure for an application area on the other side. The intermediate language may be a low level language in the sense that for each feature required the most simple constructions can be chosen. All these aspects are emphasized in the name Intermediate Language for Pictures or ILP for short.

The definition and implementation of the language constitute the design and implementation of the kernel of an interactive graphics system. The design goal of this system has been published in [4]. Since then two years have passed in which the designers have spent most of their time discussing the syntax and the semantics of ILP. This could only be justified by the fact that ILP is a major step towards the ultimate design of this graphics system. ILP plays a key role in all graphics system facilities:

-- A high level graphical language is obtained by embedding ILP in an existing high level general purpose programming language.

---

*) A picture is defined as a description of some object such that a visible image of that object can be obtained from this description in a uniform way. The description may include both geometrical (shape, size) and non-geometrical (colour, weight) properties of the object.

-- The control of every drawing machine in the graphics system is defined by a conversion between ILP code and device code. This is true for input as well as output. As an important consequence full symmetry between input and output is obtained.

-- A picture file system is defined and organized as a library for ILP programs in which the latter can be stored, retrieved and classified.

With the aid of these facilities, a graphics satellite system can be implemented containing a picture editor that can produce and transform ILP programs.

All these modules (and others that may be added) greatly profit from the conceptual uniformity provided by ILP.

## 1.2. The design of ILP

A further function of ILP, which as such is only implicitly present in the graphics system, is that it provides a means of communicating about the graphics system during the design phase. In the past two years the designers have developed a discipline in which every aspect of the system is discussed in terms of ILP. To support this communication, a symbolic notation for ILP programs has been introduced, which makes ILP look like an ordinary programming language. The success of this symbolic code was so convincing that it was decided to use the same code for the definition of ILP in this report. Moreover, each module of the system will be implemented in such a way that it can work in a state in which it communicates through symbolic ILP code. This constitutes a very useful testing facility and also proves that conceptual uniformity has been preserved.

In the period when the designers decided to work according to the scheme explained above, they assumed overly optimistic that a review of existing graphics systems would either reveal an existing prototype language for ILP or produce a list of features taken from various languages together which make up the prototype. Neither turned out to be the case. Most "prototype" languages suffered from the fact that they had been forced into the frame of a so-called FORTRAN interface. Since the only two means of expression here are subroutine identifier and simple parameters, designers of these packages always argue that the number of identifiers and parameters should be kept small, and above all that the interrelation between function call's must be exceedingly simple, because each structuring function (like opening and closing brackets) requires subroutine calls scattered throughout the application program.

The effect of this type of limitations is that everybody choses a subset of desired features. No two subsets have the same representation in terms of identifiers and parameters and moreover all subsets are mutually different and declared to be the best of all possible choices (in view of the unspecified circumstances).

Given this state of affairs, the designers decided to adhere to the principle that if a feature would be included it would be included completely. One of the consequences is that a FORTRAN subroutine library is most unlikely to be a suitable representation of ILP.

More interesting material was provided by graphic languages that support data structures. In these cases efficiency of problem representation plays a major role. Complicated data structures for graphics are justified by the fact that the applications program can use the same data structure. In this way the problem of representing graphical data structures is generalized towards structuring associative data or towards hierarchies of cyclical data. This type of languages cancels itself out for the bulk of the moderate applications of computer graphics. From this observation the designers drew the conclusion that it made sense to try to characterize the complexity of <u>purely graphical</u> information. The best way to do this seemed to be defining a complete graphical language and find the simplest representation for it.

The language ILP deals with four major facets of graphical information:

-- The elementary drawing actions.

-- Modifications of such drawing actions under control of state information.

-- Structuring (and combining) states and actions.

-- Specification of entry points for external references on which interaction and association of non graphical data can be based.

As such ILP constitutes a so-called general purpose modelling system.

The elementary drawing actions must be understood as a means to visualize elementary geometrical objects. Typical actions are to draw a point, line, contour (closed polygon) and curve. Less typical but useful is text. In fact the exclusive (exceptional) function of text has caused a number of unsolved problems with respect to the orthogonality of the design. Typical state information consists of transformations, coordinate mode (absolute or incremental) and style functions (line style, typographic style for text etc.). All non-geometrical aspects have been isolated from the actions and are controlled by independent state information. For instance, invisible moves that are used for positioning are not considered as drawing actions. This type of information is entirely included in the state. In so far invisible moves can be found among the drawing actions, they represent part of a geometrical object (e.g. invisible line or invisible curve). Here the prefix invisible is state information.

A second important consequence of the distinction between geometrical and non geometrical information is that an exact specification of the effect of actions and the state on the pen position is possible.

The state information follows two important principles. A complete state vector can be split in a number of subvectors which are all manipulated independently, i.e. a change of one subvector never has consequences for the effect of the other subvectors. The state manipulations are chosen in such a way that a new state can be obtained from an existing one by respecifying or adjusting a minimal number of values. The second important principle is that all independent subvectors in the state have the same basic structure. Moreover, the same basic manipulations can be applied to all of them. In other words a uniform scheme has been found that allows a large variety of properties to be associated with geometrical information. The basic manipulations can produce the right values as well as the right structure.

The modifying effect of state information on actions can be specified for each subvector separately, provided that priority rules are obeyed which define (as far as necessary) the order in which the state subvectors must be applied.

The rigorous and simple scheme for such a variety of concepts has imposed surprisingly little restrictions on the expressive power of ILP. In order to make this clear we have, throughout the report, put a strong emphasis on such restrictions. Especially the criterion that only complete features should be included was (almost) never violated.

The structuring of ILP data is obtained by grouping and combining. Grouping means that a number of similar constructions is put together as a unit on a higher level in hierarchy. Combining means that two different constructs are put together in a unit. At the level of elementary actions, the grouping of sequences of similar actions is implicit. At the level of complete pictures, one or more of them can be put together for the purpose of multiple referencing (subpictures) or as a conceptual unit (embracing). Both forms of grouping can be found in the representation. Moreover, grouping itself can be specified without having elementary actions. In this way, the structure skeleton of a picture can be specified. Combining always involves state information on one hand and actions on the other. Combining is used for setting up the right state.

The facilities for structuring are not allowed to produce cyclic structures. This would introduce the need for conditions in ILP, that break the cycle. This is an example of excluding features that are convenient for programming but not fundamental for picture representation.

All references in ILP have the same form and are represented by a symbolic name. All entry points for external references are represented in the same way.

## 1.3. The description of ILP

ILP is described in chapters 2 and 3. Chapter 2 gives an introduction to the basic concepts of ILP, by means of simple examples. The function of chapter 2 is to provide an overview of ILP before plunging into all syntactic and semantic details as presented in chapter 3. Obviously chapter 3 is the most important one. Here, indeed everything is brought together that concerns the basic function of ILP: the representation of pictures.

Two interesting subjects concerning ILP have been left out of chapter 3. First of all the justification of most constructions of ILP is omitted. In most cases the justification can be deduced from the fact that it contributes to the representation of a particular construct. Moreover, it was felt important to concentrate on a precise definition leaving aside all matters that make the definition more complicated (like for instance, defining alternative constructions). Secondly the role of ILP in the various modules of the graphics system is not further explained in chapter 3. Apart from the argument just given, this is also because many of the modules (like the picture editor) at this moment only exist in the designers minds.

The fact that this report contains a preliminary description means that it will be followed by one (!) final version. This version will not be published before sufficient experience has been gained with ILP and the design method based on it.

The reader of chapter 3 will notice that some constructions of ILP have been specified in great detail and an attempt has been made to be very precise about them. Other constructions are presented rather sloppy. The detailed descriptions concern new or for ILP important constructions. The less precise definitions have been used some times to avoid lengthy descriptions of what is intuitively clear (e.g. the conversion of ILP primitives to, say, a plot file), sometimes because the details have yet to be filled in (e.g. parameters for curves). The reason for being less precise (or incomplete) is in most cases given in the form of remarks, which as such are not part of the definition. Any construction of ILP, precise or imprecise, may be subjected to changes in the near future as a result of criticisms from inside or outside. Especially the latter will be very much appreciated.

ILP can be extended in two directions. New primitive actions and state information can be added to cover the representation of other classes of pictures (e.g. grey scales). New constructions for structuring and building hierarchies of states (vectors of vectors) can be introduced to allow all kinds of manipulations (e.g. movies). For both type of extensions ILP must preferably constitute the kernel. Because in that case ILP can close the gap between "classical" and "modern" computer graphics. Chapter 4 gives an idea of developments of ILP planned in the nearest future.

To facilitate reading of the remainder we now give an overview of notational conventions throughout the report. Most basic concepts of ILP are at the same time non-terminals of the syntax. They are denoted in a special font, e.g.: *non_terminal*. Basic concepts that are not a syntactical category are underlined at first (and defining) occurrence. Syntactic terminals are denoted in capital letters, e.g. TERMINAL. There is an index (Appendix 3) that references the occurrences of most concepts. Footnotes and REMARKs are used to add comments to the text in places where it is important to separate the essential from the explanatory.

On the primitive level of ILP the designers have hardly attempted to introduce new concepts in picture description. It is in the field of structuring graphical information that a new, more uniform frame is introduced. This frame, it is hoped, brings the large variety of elementary constructions needed for picture description more happily together.

# 2. AN OVERVIEW OF ILP

## 2.1. Introduction

In this chapter, ILP will be presented in an informal way. The chapter has a tutorial character and will heavily rely on short examples. No attempt has been made to cover the subject exhaustively. Only aspects that are characteristic of ILP and distinguish it from other graphics languages get attention. In particular, standard concepts from computer graphics (linear transformations, styles etc.) will not be discussed in their own right. For these topics, we refer to the introductory texts [1] and [2].

In all cases where the examples leave some doubt about what is precisely possible in ILP and what the exact semantics of ILP constructions are, chapter 3, which contains the formal definitions, should provide the answers.

## 2.2. Picture elements

Picture elements are language primitives, used to describe basic drawing actions. They represent lines, points and the like, drawn in some user-selectable Euclidean space of an arbitrary dimension, called user space. How this space can be selected, will be described in 2.6.. Until then, in all examples a two-dimensional space, with orthogonal coordinate axes, is assumed.

Example 1

```
PICT ( 2 ) ex1
      WITH { VISIBLE ; FIXED }
      DRAW {
              LINE ( [ 0.5 , 1 ] , [ 1 , 0 ] , [ 0 , 0 ] );
              POINT ( [ -0.5 , -1 ] )
              } .
```

A two dimensional picture (a picture that must be drawn in a plane) is defined, having name ex1. The dimension is specified by the number "2" surrounded by parenthesis, immediately following keyword PICT. In general, any positive integer may be used in this place.

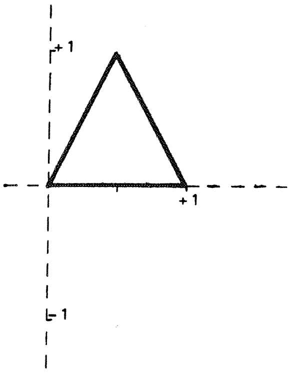
The picture consists of three line segments and a point. All elements of this picture are explicitly declared visible (by VISIBLE). The use of FIXED causes all coordinate pairs [ a , b ] to be interpreted as absolute positions in user space. (The other possibility will be dealt with in 2.3.).

The essential elements in this example are the picture elements LINE and POINT. To describe their effect, we need the concept "untransformed pen position". In general, this is the most recently visited position in user space. At the beginning of the drawing actions specified by an ILP program, it is the origin of user space.

The picture element LINE states, that lines have to be drawn, from the untransformed pen position, in this case [ 0 , 0 ], to [ 0.5 , 1 ], from [ 0.5 , 1 ] to [ 1 , 0 ] and from [ 1 , 0 ] to [ 0 , 0 ].

The picture element POINT says, that a point must be drawn at [ -0.5 , -1 ]. In general, POINT too, can have a number of coordinates as its arguments.

The drawing defined by the program above looks like:



Here, as well as in the following examples, the coordinate axes are only added for illustrative purposes. They are not normally part of ILP output.

Another ILP primitive is the picture element TEXT. Its use is shown in example 2.

Example 2

```
PICT ( 2 ) ex2
       WITH VISIBLE
       DRAW
              TEXT ("A" , " triangle" ) .
```

Here, the string "A " and "triangle" are drawn, starting again at the untransformed pen position.

2.3. Attribute classes

We have already encountered pieces of ILP programs, enclosed between the "brackets" WITH and DRAW. These program parts consisted of sequences of attribute class elements, separated by semicolons. Attributes are instruments to influence the way in which a picture element is drawn, or to associate non graphical information with it.

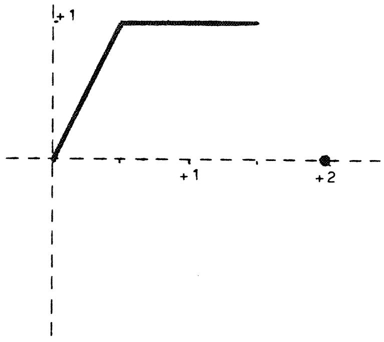
All attributes together, that are relevant for a particular picture element, determine the so-called state of that element. This state determines, what will actually happen, when that element is drawn. Attributes are divided in attribute classes, each corresponding to a particular type of operation on picture elements. In the following, some examples will be given on the attribute classes coordinate mode, transformation and style.

The class coordinate mode can have either the value FIXED or FREE. It operates on the coordinates of picture elements. In the case of FIXED, coordinates denote absolute positions in user space, as illustrated in example 1. When FREE is used, the coordinates denote increments relative to the untransformed pen position.

Example 3

Replace in example 1 FIXED by FREE.

The resulting drawing is:

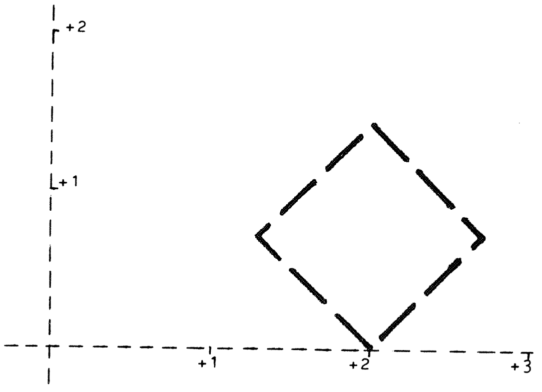


Note in particular, that only two visible lines are shown in this drawing. This is so, because an increment [ 0 , 0 ] represents a line of zero length, which coincides with the end point of the second line.

Example 4

```
PICT ( 2 ) ex4
       WITH {
               FIXED ;
               TRANSLATE [ 1,0 ] ;
               ROTATE 45 AROUND (  [ 1,0 ] ) ;
               PERIOD ( 50 , 25 , 25 ) ;
               MAP ( 0.4 , CONTINUE )
               }
       DRAW
               LINE ( [ 1,0 ] , [ 1,1 ] , [ 0,1 ] , [ 0,0 ] )  .
```

The corresponding drawing is:



A square is drawn, rotated clockwise through 45 degrees, around its lower right-hand corner. This square is translated 1 unit in the x-direction and 0 units in the y-direction. TRANSLATE and ROTATE denote transformations. (Note that the rightmost transformation is applied first!). PERIOD and MAP denote elements of the attribute class style, they determine line style. In this case, the style pattern, defined by PERIOD consists of a dash, a gap and again a dash, with respective length of 50, 25 and 25 units. MAP specifies that the actual length of this pattern in user space is 2, and that the pattern continues from one line (element) to the next.

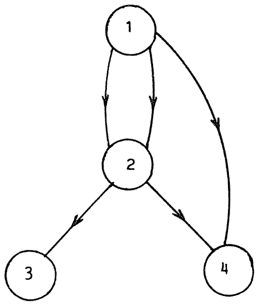
## 2.4. Data structure

ILP can be viewed as a language to describe data structures, which in turn correspond to drawings.

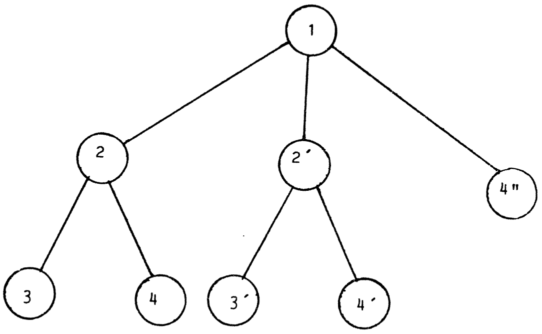
An ILP data structure has the form of a directed acyclic graph. The pictures and attributes correspond to nodes of the graph, the references to pictures and attributes correspond to arcs. The acyclicity results from the semantic rule, that ILP programs may not be recursive. The graph can be converted into a tree, by making copies of all multiply referenced nodes, and creating appropriate references to these new nodes.

Example 5

Suppose, an ILP graph has the form:



The corresponding tree looks like:



In this tree, 2' is a copy of node 2, 3' of 3, and 4'' and 4' of node 4.

The drawing represented by the tree, can be produced by a process called <u>elaboration</u>. During this process, the tree is traversed in <u>preorder</u> [3], which is recursively defined by:

- -- Visit the root of the tree.
- -- Traverse its descendant sub-trees in preorder. Descendants are traversed one by one, starting with the leftmost subtree, then proceeding to its rightmost neighbour and so on, until the rightmost subtree has been traversed.

Example 6

The nodes of the tree in example 5 are visited in the following order: 1 2 3 4 2' 3' 4' 4''.

At every node where, according to the ILP program some action must take place (drawing, evaluating of attributes and updating the state, subspace selection), this action is initiated by the elaboration process when this node is encountered. Only at the picture leaves (representing picture elements), drawing actions are performed.

In sections 2.4.1. till 2.4.3. these data structure aspects of ILP will be elucidated with the help of some examples.

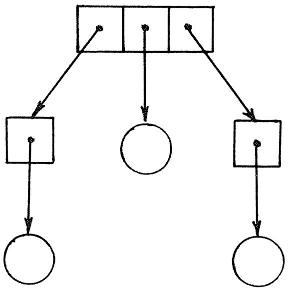
### 2.4.1. Pure pictures

Pure pictures correspond to subtrees (graphs) of the full ILP tree (graph). They are characterized by the property that they do not contain attributes. A pure picture can constitute a correct and complete ILP program in which all attributes have default values. We will ignore attributes for the moment and introduce some ILP concepts using pure pictures as examples.

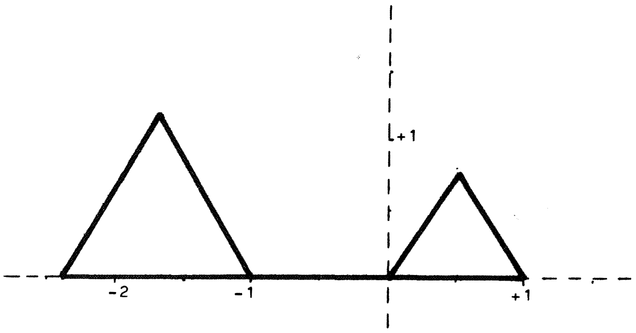
Example 7

```
SUBPICT ( 2 ) pyr1
        LINE ( [ 0.5 , 0.8 ] , [ 1 , 0 ] , [ 0 , 0 ] ) .

SUBPICT ( 2 ) pyr2
        LINE ( [ -1.7 , 1.2 ] , [ -2.4 , 0 ] , [ -1 , 0 ] ) .

PICT ( 2 ) egypt
        { pyr1 ; LINE ( [ -1 , 0 ] ) ; pyr2 } .

PICT ( 2 ) ex7
        WITH FIXED DRAW egypt .
```

The tree, defined by ex7 contains a pure picture tree, corresponding to egypt.



The drawing looks like:



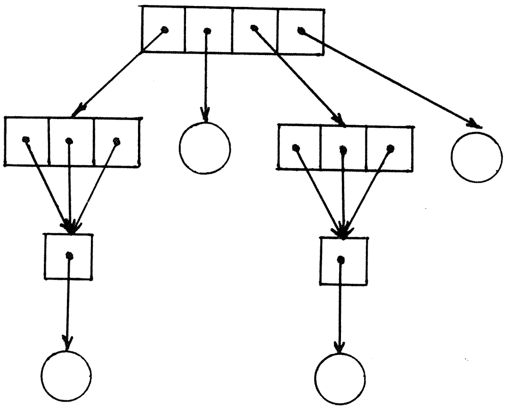
This last example illustrates that there are two kinds of named pictures (pictures having a name); root pictures (designated by PICT) and sub pictures (SUBPICT). Root pictures are the only ones that may be referred to from outside the ILP program in which they are defined. The root of an ILP graph must correspond to a root picture or in other words, elaboration can only start in a root picture.

Another example of a named picture, defining a pure picture graph, is the following:

Example 8

```
SUBPICT ( 2 ) tooth1
       LINE ( [ 0.5 , 2 ] , [ 0.5 , -2 ] ) .

SUBPICT ( 2 ) tooth2
       LINE ( [ -0.5 , -2 ] , [ -0.5 , 2 ] ) .

SUBPICT ( 2 ) teeth1
       { tooth1 ; tooth1 ; tooth1 } .

SUBPICT ( 2 ) teeth2
       { tooth2 ; tooth2 ; tooth2 } .

PICT ( 2 ) jaws
       { teeth1 ; LINE ( [ 0 , 5 ] ) ;
         teeth2 ; LINE ( [ 0 , -5 ] ) } .
```

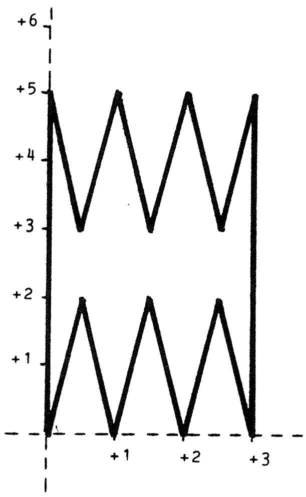
The picture graph defined by jaws is



The ILP statement:

PICT ( 2 ) ex8 WITH FREE DRAW jaws .

defines the drawing:



All elements of a pure picture are elaborated in the same state. The structure of example 8 can therefore be reduced (but not compactified) to a linear list of LINE's. However in that case the logical distinction between tooth and teeth is lost.

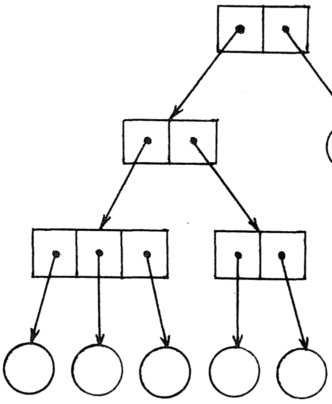
### 2.4.2. Pure attribute graphs and picture nodes

As with named pictures, attributes can be grouped in named units too, called attribute packs.

Example 9

```
ATTR ( 2 ) transformpack {
       ROTATE 90 AROUND ( [ 1 , 1 ] ) ;
       SCALE [ 2 , 3 ] ;
       TRANSLATE [ -1 , 0.5 ]
       } .

ATTR DIMLESS stylepack {
       MAP ( 10 CONTINUE ) ;
       PERIOD ( 10 , 3 , 11 )
       } .

ATTR ( 2 ) ex9 {
       { transformpack ; stylepack } ; THICK ( 10 ) } .
```

Attribute pack ex9 defines a pure attribute graph of the form:



Just as named pictures, attribute packs have a dimension, which is specified in the same way. This dimension is obviously meaningful when the pack contains for instance transformations. In other cases (for instance for a style pack), the pack could be used in combination with pictures of arbitrary dimension. Arbitrary dimension is specified by DIMLESS

This example illustrates another property of ILP programs: by means of brackets, structure can be enforced, without using explicit references to (using names of) objects. In example 9 the attribute node labelled "A" is added because of the construction:

```
        { transformpack ; stylepack }
```

In the same way, picture nodes can be created.

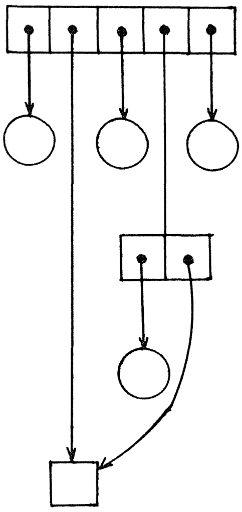
As can be seen from the examples already given, the WITH...DRAW construction links attributes to pictures. In the data structure, a WITH...DRAW node is itself a picture node, i.e. at any place in the data structure where a reference to a pure picture graph is permissible, a reference to a WITH...DRAW node is allowed as well. A picture graph has a structure similar to that of a pure picture graph, but with the extra property, that certain picture nodes (WITH...DRAW nodes) have pure attribute graphs also as descendants. In other words, a WITH...DRAW node of a picture graph, has a number of pure attribute graphs, as well as a number of picture graphs as its descendants.

The data structure defined by an ILP program, is a picture graph.

Example 10

```
PICT ( 2 ) ex10
        WITH FREE DRAW {
                teeth1 ;
                LINE ( [ 0 , 5 ] ) ;
                WITH ROTATE 180 AROUND ( [ 3 , 5 ] )
                DRAW teeth1 ;
                LINE ( [ 0 , -5 ] )
        } .
```

The data structure has the form:



This data structure contains two WITH...DRAW nodes, labelled "WD1" respectively "WD2". When for teeth1 the subpicture defined in example 8 is taken, the drawing "jaws" results again. The lower jaw is only subjected to the attribute from WD1, the upper jaw comes in its anatomicaly correct position, because it is subjected to the rotation attribute from WD2 as well.

### 2.4.3. Combining attributes into a state

As shown in the previous examples, a variety of attributes (possibly specified in different WITH...DRAW constructions), can influence a picture element. Clearly it is necessary, to combine these various entities in units that can be meaningfully applied to picture elements. Only elements from one and the same attribute class will be mutually combined (mixed), in a way that may be specific for the class to which they belong. Next, these combinations are packed into the state. The combinations are applied to picture elements in some fixed order, defined by priority rules.

Example 11

```
PICT ( 2 ) ex11
       WITH { SCALE [ 1 , 2 ] ; SCALE [ 2 , 1 ] }
       DRAW P .
```

A scaling is an attribute of the class transformation. Transformations are simply applied one after the other, starting with the rightmost one (the one, textually closest to the picture element). Hence, this program is semantically equivalent to:

```
PICT ( 2 ) ex11
       WITH SCALE [ 2 , 2 ]
       DRAW P .
```

Example 12

```
PICT ( 2 ) ex12
       WITH SCALE [ 1 , 2 ]
       DRAW
               WITH SCALE [ 2 , 1 ]
               DRAW P .
```

Again this program is semantically equivalent with the previous two.

Example 13

```
PICT ( 3 ) ex13
       WITH MAP ( 3 CONTINUE )
       DRAW {
               P1;
               WITH MAP ( 5 RESETLINE )
               DRAW P2
               } .
```

P1 is drawn under influence of the first map specification, P2 under influence of both the first and the second. Clearly it is meaningless, to apply two map specifications in succession, so they have to be combined into one single map. In general, this mixing is done by concatenation rules, which look like:

A <> B -> C

where A, B and C are elements from the same attribute class. The meaning is, that A concatenated with B, gives C. In case of map, this rule reads:

A <> B -> B

showing simply, that the second map definition replaces the first.

There exist still a third way of combining, used for a type of attributes which are much more complex and powerful then the previous two. In the present version of ILP, it is only used for detection (see 3.4.7.) and will be amply illustrated there.

Example 14

```
PICT ( 1 ) ex14
       WITH { SCALE [ 3 ] ; MAP ( 2 CONTINUE ) }
       DRAW P .
```

Here, the priority rules require, that first the transformation, (SCALE) and then the style element (MAP) is applied. This has consequences, because a transformed picture element drawn with a certain style, can look quite different from a picture element with a certain style applied to it which is thereafter transformed. (The latter is impossible in ILP.)

Example 15

```
PICT ( 2 ) ex15
       WITH A1
       DRAW {
               WITH A2 DRAW P1 ;
               WITH A3 DRAW P2
               } .
```

When the corresponding data structure is traversed, first the collection of attributes contained in $A_1$ is encountered, then those in $A_2$ and finally those in $A_3$. $P_2$ is affected both by attributes $A_1$ and $A_3$, $P_1$ by $A_1$ and $A_2$. Attribute combination is defined in such a way, that the following efficient combination scheme can be employed:

at $A_1$: Combine all attributes from $A_1$ in a (partial) state $SP_1$. $SP_1$ is identical to state $S_1$.

at $A_2$: Combine the attributes from $A_2$ in a (partial) state $SP_2$. Combine $S_1$ and $SP_2$, this gives state $S_2$. $S_2$ is applied to $P_1$.

at $A_3$: Combine the attributes from $A_3$ in a (partial) state $SP_3$. Combine $S_1$ and $SP_3$, this gives state $S_3$. $S_3$ is applied to $P_2$.

Hence, attributes within one WITH...DRAW construction have to be combined only once during elaboration. Attributes from nested WITH...DRAW constructions can be combined and retrieved, using a stack.

## 2.5. Default attribute, matches and prefixes

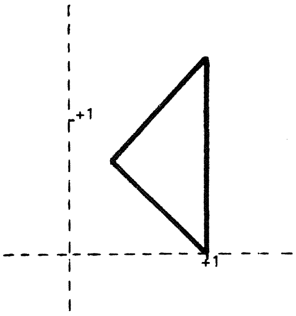
Every attribute class has a default element. If, during elaboration, a picture element is reached and the state does not contain a fully specified element for a certain attribute class, the default element is used. For instance, the default transformation is a unit matrix, the default for visibility is VISIBLE. Defaults release the user of the burden to specify values for all attribute classes.

With most attribute classes, an attribute match is associated. Its function is, to switch at the picture element level, between the default value of the associated class and the value specified in the program.

Example 16

```
PICT ( 2 ) ex16
       WITH { FIXED ; ROTATE 45 AROUND ( [ 1 , 0 ] ) }
       DRAW
       LINE ( [ 1,0 ] , ~TF [ 1,1 ] , [ 0,1 ] , [ 0,0 ] ) .
```

The drawing is:



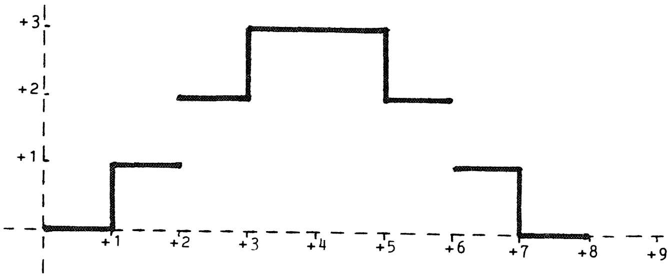
TF is the match for transformations. ~TF here signifies, that the second line element must not be rotated, but subjected to the default transformation (unit matrix) instead.

Matches not only can be applied to "arguments" (for instance coordinates) of a picture element, but also to the element as a whole. This leads to the possibility of two levels of matches. The one, directly preceding an "argument", locally replaces the match of a whole element.

Example 17

```
PICT ( 2 ) ex17
       WITH { INVISIBLE ; FREE }
       DRAW
       LINE ~ VS ( [ 1,0 ] , [ 0,1 ] , [ 1,0 ] ,
       VS [ 0,1 ] , [ 1,0 ] , [ 0,1 ] , [ 2,0 ] ,
       [ 0,-1 ] , [ 1,0 ] , VS [ 0,-1 ] , [ 0,1 ] ,
       [ 0,-1 ] , [ 0,1 ] ) .
```

The drawing is:

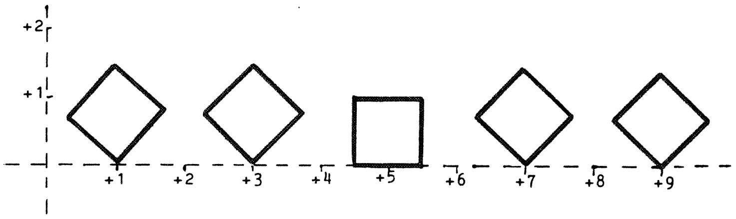


Visibility gets value INVISIBLE. The match ~VS directly following "LINE", replaces this class value by the default value VISIBLE, so the line as a whole is made visible. Locally, the explicit class value INVISIBLE is reinstalled by the match VS, causing some line segments to become invisible.

Every attribute can be prefixed either by ABS or by REL. If no prefix is present (as in all our examples until here), prefix REL is assumed. If an attribute has prefix REL, it will be combined with the appropriate class value, contained in the current state. If it has prefix ABS, the combination starts all over again, beginning with this attribute, thereby disregarding the class value already accumulated.

Example 18

```
SUBPICT ( 2 ) square
        LINE ( [ 1,0] , [ 1,1 ] , [ 0,1 ] , [ 0,0 ] ) .

PICT ( 2 ) ex18
        WITH { FIXED ; ROTATE 45 AROUND ( [ 1 , 0 ] ) }
        DRAW {
                 square ;
                 WITH REL TRANSLATE [ 2 , 0 ] DRAW square ;
                 WITH ABS TRANSLATE [ 4.5 , 0 ] DRAW square ;
                 WITH REL TRANSLATE [ 6 , 0 ] DRAW square ;
                 WITH REL TRANSLATE [ 8 , 0 ] DRAW square
        } .
```



For the first, second, fourth and fifth square, translation is combined with rotation, for the third square, rotation is switched off. When ABS would not have been used, the same drawing could only have been made in a much more clumsy way: either the third square should have been subjected to an inverse rotation, or it should have been defined outside the scope of the outermost WITH...DRAW construction.

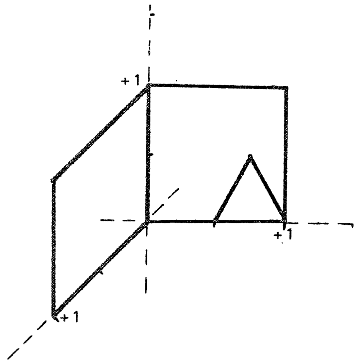
## 2.6. Subspace

The subspace construction is the mechanism to redefine the coordinate system of user space. It can be used to change axes, without changing the dimension of user space and to specify proper subspaces (i.e. with lower dimension) of an envelopping space. Hence dimension can change in an ILP program. The dimension of subpictures, root pictures and attribute packs is explicitly specified and determines the number of components of coordinates , matrices etc. Hence it can be statically checked, whether ILP statements within the scope of a subspace selection, use elements of the proper dimension.

Example 19

```
PICT ( 3 ) ex19
       WITH FIXED
       DRAW {
       LINE ( [ 1,0,0 ] , [ 1,1,0 ] , [ 0,1,0 ] ,
       [ 0,0,0 ] , [ 0,0,1 ] ,
       [ 0,1,1 ] , [ 0,1,0 ] ) ;
       SUBSPACE ( 2 )
               ORIGIN ( [ 0.5,0,0 ] , [ 1,0,0 ] , [ 0,1,0 ] )
               WITH FREE
               DRAW LINE ( [ 0.25,0.5 ] , [ 0.25,-0.5 ] )
       } .
```

The drawing is:



First, squares are drawn in the (x,y) plane, resp (y,z) plane. Then the (x,y) plane is selected as a two dimensional subspace. The subspace origin coincides with the point [ 0.5 , 0 , 0 ]. Its x and y axes are identical to those of the envelopping space.

In general, the first "argument" of ORIGIN specifies the new origin, the further "arguments" specify the new axes as vectors in the old coordinate system. In this subspace a triangle is drawn. The coordinates of this triangle must be specified by two numbers instead of three.

The dimension of the root picture where elaboration starts, is defined by that picture itself. The coordinate axes of the user space at the root (the untransformed user space) form by default a right handed, orthogonal coordinate system. After all transformations have been applied to coordinates in a picture element a position in untransformed user space results. This position must lie in the user unit cube, i.e. all its coordinate components must have absolute values less than or equal to one. As a consequence, there seems to be a choice between using picture elements with only small coordinate values, which is quite impractical, or applying a scale transformation at the root. The second possibility is also unpleasant, because it prohibits the use of "ABS" with lower level transformations, which would switch off the scale. The subspace mechanism provides a practical third alternative however.

Example 20

Suppose, elaboration starts in the following root picture:

```
PICT ( 2 ) ex20
       SUBSPACE ( 2 )
         ORIGIN ( [ 0,0 ] ,  [ 0.0001,0 ] , [ 0,0.001 ] )
               "rest of root picture" .
```

Immediately, a new coordinate system is introduced, with its origin and axes coincident with those of the two dimensional untransformed user space. The length unit in this new space is 0.001 of that of the untransformed space however. As a consequence, the coordinate values produced by "rest of root pict" may have absolute values <= 1000. Furthermore, the subspace transformation, which is not an attribute cannot be switched of by "ABS".

Example 21

```
SUBSPACE ( 2 )
       ORIGIN ( [ 0,0 ] , [ 0.001,0 ] , [ 0.001,0.001 ] )
```

Now, not only coordinate values are expressed in different units, but an additional affine transformation is introduced, because the y-axes of the new space coincide with the line y=x in the envelopping space.

## 2.7. Miscellaneous topics

In the preceding paragraphs, we have focussed attention on the highlights of ILP and have consequently omitted other features. To make the picture given in this overview more complete, we will very briefly discuss them now.

The set of picture elements provided in ILP contains, apart from points, lines and text, also contours (closed polygons) and generators (an elaborate library facility). Only generators will be discussed here.

Whenever the elaboration process (the process that traverses the ILP data structure, see 3.2.3.) encounters a generator, a new data structure is obtained (in some way) and inserted in the place where the generator occurs. Several types of generators exist, which differ in the way they produce a new data structure:

-- symbols: correspond with a previously defined root picture, and can hence completely be specified as ILP program.

-- curves: correspond with a recipe to produce picture elements according to a certain specification (e.g., a sinus curve). The way in which the picture elements are produced can not necessarily be described as ILP program. Curves can only produce data structures from a limited class.

-- templates: correspond with a recipe to produce any legal ILP data structure, which may be produced in any way.

Templates form the most general library facility. However, this generality must be paid for, since the data structures produced by templates have to be checked dynamically for correctness, while the correctness of the data structures produced by curves and symbols can be determined statically.

The set of attribute classes contains, apart from transformations and coordinate mode, also style, pen, detection and control.

Coordinate mode deals with absolute and incremental drawing. Examples were given in section 2.3..

Transformations have, apart from a few exceptions the meaning as normally used in computer graphics systems ([1],[2]). An exhaustive list of transformations is:

-- rotate, scale, matrix transformation, affine transformation, homogeneous matrix transformation all with standard meaning.

-- projection, a central or parallel projection which does not reduce the dimension of a picture.

-- window, viewport which resemble the usual concepts of window and viewport, apart from some additions. It is worth mentioning that windows may be arbitrarily nested and that the nested windows may be rotated relative to each other.

Style determines what kind of picture elements must be produced by a drawing machine. In the preceding paragraphs line style (i.e., a style associated with lines) was already mentioned. A style can also be associated with points (point style: determines the symbol to be used for the representation of points) and text (typographic style: determines boldness, italicity, alphabet and the like for text values).

Pen determines the reproduction method to be used for the visualization of picture elements. Examples are colour and intensity.

Detection determines which parts of the ILP data structure can be pointed at by devices such as lightpen and cursor. The result of such an operation is not simply the picture element pointed at, but may be a part of the data structure in which the picture element is contained. In this manner ambiguities can be resolved: when pointing at a door-in-a-house, is the door or the house intended?

Detection illustrates the power of the attribute mechanism. It is also possible (but not described in this report) to associate non-graphical information with an ILP data structure. The overall scheme is that during the elaboration process the non-graphical attributes can accumulate information for later use. This method imposes the restriction that drawing order is the order in which non-graphical information must be accumulated.

# 3. THE SYNTAX AND SEMANTICS OF ILP

## 3.1. Overall Structure

The complete syntax of ILP is given in Appendices 1 and 2. In this chapter we will use extracts from it as a guide to the discussion. No attempt has been made to exclude all possible syntactical forms that have no semantic meaning. This would make the syntax extremely difficult to read. Instead we tried to keep it as simple as possible.

The syntax rules are grouped in such a way that the basic structure of the language is reflected as much as possible. The syntax is split in two parts: the set of units that will be produced by lexical scanning and the so-called main syntax. Only the main syntax will be described in this chapter, the other part is given in Appendix 2.

The semantic meaning that corresponds with each syntactical construction will be described by means of an interpretation process referred to as elaboration. In the sequel no distinction will be made between the semantic meaning associated with a certain syntactical construction and the result of the elaboration of that construction. When the elaboration of a particular language construction is carried out, the overall interpretation process is in some intermediate stage. This intermediate stage can be considered as the context in which that particular language construction is elaborated and will be referred to as environment. The elaboration process is only used as a description method and is not intended as an implementation proposal.

An ILP program (*picture_program*) consists of three distinct sets: a set of *root_pictures*, a set of *subpictures* and a set of *attribute_packs*:

```
picture_program: pictstruct |
                 picture_program pictstruct $
pictstruct:      named_picture |
                 attribute_pack $
named_picture:   root_picture |
                 subpicture $
```

A *rootpicture* has two properties that distinguish it from a *subpicture*:

- -- The only *pictures* of an ILP program, say $I_1$, that can be referenced from another ILP program, say $I_2$, are the *rootpictures* of $I_1$.
- -- The elaboration of an ILP program starts in a *rootpicture* and not in a *subpicture*. *subpictures* can only be activated via a *namedpicture* in the same ILP program.

REMARK

It might be considered to relate these properties to distinct "pictstruct" types, like:

INPICT: a pictstruct in which the elaboration of an ILP program can start

EXPICT: a pictstruct to which can be referred from another ILP program

INEXPICT: analogous to the present *rootpicture*

PICT: analogous to the present *subpict*

The elements of the three distinct sets are:

*rootpicture*: PICT *dimension pname picture* . $
*subpicture*: SUBPICT *dimension pname picture* . $
*attributepack*: ATTR *dimension aname attribute* . $

The only connection between a *picture* and an *attributepack* is by means of the "WITH ... DRAW" construction, e.g.

WITH A DRAW P .

The resulting construction is again of type *picture*. The rules for *picture* and *attribute* are:

```
picture:          pname |
                  picture_element |
                  { pictures } |
                  subspace picture |
                  WITH attribute
                          DRAW picture $
attribute:        ABS basic_attribute |
                  REL basic_attribute |
                  basic_attribute $
basic_attribute: attribute_class |
                  aname |
                  { attributes } |
                  NIL $
```

Note that a list of *pictures* between brackets is again a *picture* and that a list of *attributes* between brackets is again an *attribute*.

The result of the elaboration of a *picture* depends on the specification of *attributes*. Section 3.2. describes the global organization of ILP programs and the relationship between *pictures* and *attributes*.

The environment contains two groups of values:

- One group the so called state, changes as a result of elaborating *attributes*.
- The remainder changes as a result of two kinds of actions namely, elaboration of *picture_elements* or external actions.

The initial environment contains unique values for the members of both groups.

For every *root_picture*, *subpicture* and *attribute_pack* a *dimension* is specified. It determines the number of components of which coordinates and matrices consist, that occur in these constructions. In an environment with a certain dimension, only constructions of the same *dimension* may be referenced. The dimension can be changed by a *subspace* selection. *Dimension* and *subspace* are described in detail in section 3.3..

*attributes* are divided into classes. It sometimes matters in which order *attributes* from the same class are specified. *attributes* from different classes are mutually unrelated. A complete treatment of *attributes* is given in section 3.4..

The language primitives for which some visual representation exists on drawing machines are called *picture_elements*. Examples are points, lines and characters. They are described in section 3.5..

## 3.2. Graph Structure

An ILP program has no block structure. All *namedpictures* and *attributepacks* are on the same level. However, each ILP program can be considered as the representation of some directed graph structure. The terminology used for graphs is taken from KNUTH [3]. Such a graph is formed by the statical nesting of *pictures* and *attributes*. These objects are nested either as a result of referring to one object from inside another or nested textually by means of brackets. Recursive calls are explicitly forbidden, hence the graph is an oriented graph without cycles. The graph can be expanded into a tree by replacing all multiple referenced subgraphs (*namedpictures*, *attributepacks*) by separate copies. Inside an *attribute* only other *attributes* may be referenced; this gives rise to attribute nests. Attribute nests only contain *attributes*. Through the WITH ... DRAW construction (see section 3.1.), *pictures* may contain references to both *attributes* (attribute nests) and other *pictures*, resulting in picture nests.

In correspondence with the syntax, the graph has two types of nodes namely picture nodes and attribute nodes. There are corresponding types of arcs namely arcs pointing to a picture node (picture arcs) and arcs pointing to an attribute node (attribute arcs). Every *rootpicture* constitutes a connected (directed) subgraph. All picture nodes not connected to this subgraph have no meaning with respect to an elaboration of this particular *rootpicture*. In the following we will restrict ourselves to such connected subgraphs, which will be called picture graphs. If we remove the picture nodes and picture arcs from the complete graph, then for every "WITH...DRAW" node we obtain an isolated attribute graph, which contains only attribute arcs and attribute nodes.

### 3.2.1. Picture nodes

The alternatives in the following syntax rule are the constructions that can represent a picture node:

```
     picture:      pname |
                   picture_element |
                   { pictures } |
                   subspace picture |
                   WITH attribute
                             DRAW picture $
```

A *picture_element* (c.f. 3.5.) is an end node (leave). The other alternatives of the rule are nodes (but not leaves).

Note that,

{ *picture_element* }

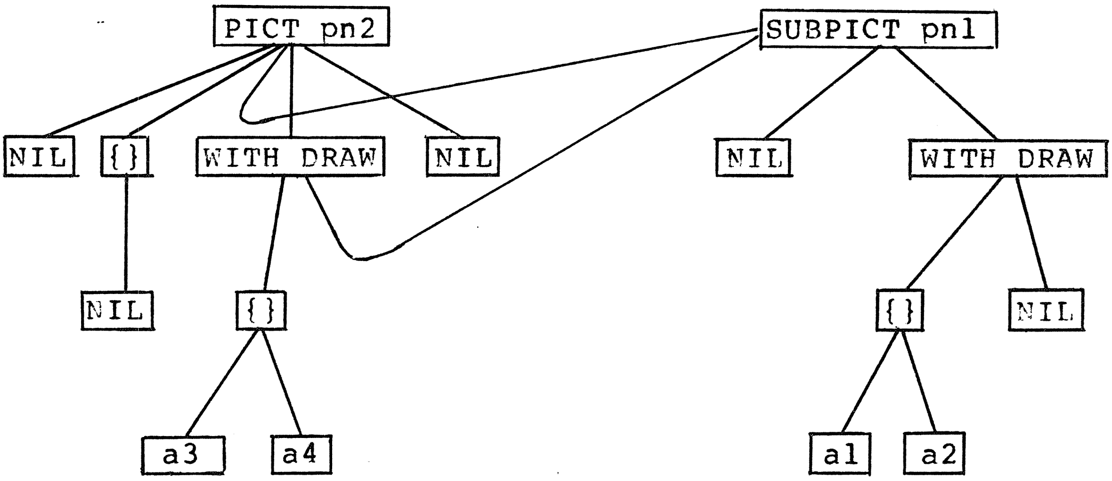
is a special case of

{ *pictures* }

which is not a leave. Because a *picture_element* may have value NIL, arbitrary graph-structures can be specified, even without writing down any other action than NIL e.g.:

```
SUBPICT DIM(3) pn1  {
        NIL ;
        WITH { a1 ; a2 }
        DRAW NIL
}.

PICT DIM(3) pn2 {
        NIL ; { NIL } ; pn1 ;
        WITH { a3 ; a4 } DRAW pn1 ; NIL
}.
```

The graph for this ILP program is:



## 3.2.2. Attribute nodes

Attribute nodes are represented by *basic_attributes* as can be seen in the syntax rules:

```
attribute:       ABS basic_attribute |
                 REL basic_attribute |
                 basic_attribute $
basic_attribute: attribute_class |
                 aname |
                 { attributes } |
                 NIL $
```

The terminal nodes are *attribute_class* and NIL. The other two, *aname* and { *attributes* }, are the non-terminal nodes. An *aname* represents a reference to an *attribute_pack*. The prefixes ABS and REL have no influence on the graph structure, but specify how the *attribute* has to be mixed with members of the same *attribute_class* (see 3.4.).

### 3.2.3. Traversing process

#### 3.2.3.1. Basic rules

The structure explained above plays a vital role in the semantics of ILP programs. The description of ILP semantics proceeds in stages. In each stage an algorithm is used that simplifies the graph towards a canonical form. The basic semantic rules associated with the graph are the following:

- Each (maximal) subgraph containing only *attributes* is converted into one list of *attributes* (algorithm ETA, see 3.4.1.). In this list all references to *attributes* (*anames*) are replaced by the *attributes* themselves (algorithm RAP, see 3.4.2.1.). Hence references to *attributes* are semantically equivalent with textual insertion of the *attributes* referred to. After further simplification (algorithm LIN, see 3.4.2.1.), the resulting list of *attributes* (called <u>state component</u>, see 3.4.3.) is applied to the picture node from which the attribute graph is a direct descendant.

- A <u>state</u> (a combination of state components, see 3.4.3.) can only be applied to picture nodes, in the way described by the following application rule:

  Application of a state to a picture node means one of two things:

  - If the picture node is a *picture_element* then all *attributes* in the state are applied as described in 3.2.3.2..

- If the picture node is not a *picture_element*, the state is applied to all its direct descendants as follows: Whenever a descendant is not a *picture_element*, the state is combined with the state component (if present) of that descendant into a new state, otherwise no action takes place. Next this application rule is used recursively.

As a result of this we have to define three semantic operations on *attributes*:

-- To combine the *attributes* in an attribute-graph into one state component.

-- To combine states and state components.

-- To apply an *attribute* to a *picture_element*.

The combination rules for *attributes* will be given in section 3.4.. The third operation is a special case of applying a state to a *picture*. This will be discussed in general in the next section and for each type of *picture_element* in particular in sections 3.4.4. till 3.4.10..

### 3.2.3.2. Pictures and picture elements

When the elaboration begins, an initial state is set up as part of the initial environment. Then the traversing process starts in the initial *root_picture*.

When during the traversing process a *picture*, which is not a *picture_element*, is encountered, the following rules apply:

-- Set up a state for that node, by combining the state component (if present) of that node and the previous (either parent or initial) state.

-- Visit all descendants of the node in left-to-right order (which corresponds to textual order in the ILP program).

-- Return to the parent node and restore the original state of that node.

In terms of the semantically equivalent tree (the expanded graph), nodes are visited in preorder.

Nodes that are *picture_elements*, represent drawing operations. If these operations are executed by a drawing machine the following happens:

- The mode of the drawing machine is updated according to the state.
- Whenever necessary, the *picture_element* is changed into zero or more new *picture_elements* by applying the state to it. *)
- Each resulting *picture_element* obtained is used to drive the drawing machine.

Thus in addition to the combination rules for state and state component, the semantic operations needed in order to elaborate a *picture* are:

- Restore, save and combine state( component)s.
- Return from and call a *picture*.
- Elaborate *picture_elements*.

So the general scheme is that while traversing the subgraph containing all *pictures* the current state is either updated or applied to a *picture_element*.

## 3.3. Dimension and subspace

Pictures considered as geometrical objects are defined in an Euclidean space with coordinate axes and a certain dimension. The description of a picture can be simplified by choosing a space of minimal dimension. In many cases, for the user, the position of the picture with respect to the axes is another means to simplify the description. The ILP *subspace* mechanism makes it possible to temporarily change the dimension of the space in which a picture is being constructed. It can reduce the dimension in order to reflect the inherent dimension of that picture. It can also redefine the position and orientation of the axes. If a picture lies, for example, in a given plane then two coordinates are sufficient to specify a point of that picture. In this case the given plane can be selected by *subspace* and as a consequence all redundant coordinates in the picture specification must be omitted.

The *subspace* mechanism is an instrument for the posi-

---

*) The result of the application of the state can partly be described by means of ILP primitives. When this method is used in the sequel, this does not imply, that in an actual implementation the modified *picture_elements* must be available as ILP objects.

tioning of individual *pictures* in the common space, because it defines a mapping between its own coordinate system and that of the surrounding space.

### 3.3.1. Dimension

Before we go into the details of *subspace* selection, some attention must be paid to coordinate systems. The *coordinates* in an ILP program are expressed in user coordinates.

By means of a *subspace* selection, the coordinate system is changed. *Coordinates* that occur in a *picture* which starts with a *subspace*, are expressed in so-called transformed user coordinates, otherwise in so-called unit user coordinates. Hence, between the start of the elaboration of the initial *root picture* and the occurrence of the first *subspace*, all coordinates are unit user coordinates.

The unit user coordinates are a right-handed Cartesian coordinate system.

During elaboration, *coordinates* are transformed by *subspace* transformations, matrix *transformations*, and *port transformations* (see 3.4.4.). In general, coordinates can have arbitrary real values, but there is one important restriction: *coordinates*, subjected to all relevant transformations, can be divided into two groups: those that pass through all *windows* involved (see 3.4.4.8.) and those that lie outside at least one *window*. The unit user coordinates of the first group all must lie in the user unit cube, i.e. have values in the real interval [-1.0,+1.0].

Finally, there exists for each drawing machine a fixed, device-dependent mapping from the user unit cube onto points in the addressing area of that device. This mapping is established at the moment of device-selection and is parameterized outside ILP. Because the position and orientation of this addressing area relative to the user unit cube can be chosen freely, devices with non-square (or non-cubic) addressing areas can be handled. In this way the mapping on the physical addressing area of an actual drawing device has to be specified for the unit cube only.

A *dimensional_value* is the ILP equivalent of what is elsewhere known as a "coordinate pair" or "coordinates". As can be seen in the syntax rules:

```
dimensional_value: [ values ] $
values:            value |
                   values , value $
```

In ILP, *coordinates* contain *dimensional_values* as a special case. For instance, a *coordinate* also specifies whether the *values* of the *dimensional_values* are absolute or incremental. When in the sequel the term *dimensional_values* is used some meaning must be assigned to the special properties that come with *dimensional_values* only. In all other cases the term *coordinates* is maintained.

The dimension of a *dimensional_value* (i.e. the number of *values* of which the *dimensional_value* consists) is not dictated by the syntax. On the other hand, *subspace* (see 3.3.2.) fixes, among other things, the dimension of the environment. Therefore the following semantic rule ( general dimension rule ) is required to enforce the right dimension of *dimensional_values* in various contexts:

> In an environment of a certain dimension, the following constructions may only occur with the same dimension as that of the *subspace*:
>
> -- *dimensional_value*;
>
> -- reference to a *subpicture* and a *root_picture*;
>
> -- reference to an *attribute_pack*;
>
> -- *subspace* selection.

To enforce this rule, a *dimension* specification is required for each *root_picture*, *subpicture*, *attribute_pack* or *subspace*. This implies, for example, that in a *subpicture* with *dimension* two, only *dimensional_values* consisting of two *values* may occur. *Dimension* is syntactically described by:

```
dimension:     DIMLESS |
               dim $
dim:           ( value ) |
               empty $
```

Because some *attributes* (like colour and intensity) and *picture_elements* (e.g. NIL) are dimension independent the *dimension* specification DIMLESS exists. A DIMLESS *attribute_pack*, *root_picture* or *subpicture* may be referenced in any environment, regardless of its dimension.

The mechanism just described is extended further to cater for matrices of *dimensional_values*:

*matrix_value*: [ *dimensional_values* ] $
*dimensional_values*: *dimensional_value* |
*dimensional_values* ,
*dimensional_value* $

A *matrix_value* consists of a number of *dimensional_values* equal to the dimension of the current environment.

The other constructions which must fit dimension, are *subspace*, *rotate* and *homogeneous_matrix*. The restrictions on their values are discussed in 3.3.2., 3.4.4.1. and 3.4.4.7..

### 3.3.2. Subspaces

With the aid of this conceptual framework, the *subspace* selection mechanism can now be explained. Syntactically a *subspace* is specified as follows:

*subspace*: SUBSPACE *dim new_axes* $
*new_axes*: *position* ( *shift axes* ) $
*shift*: *dimensional_value* $
*position*: CURRENT |
ORIGIN $
*axes*: *empty* |
, *dimensional_values* $

The elaboration of a *subspace* results in a new transformed user space. The *subspace* construction defines new coordinate axes with respect to the ones, still valid during its elaboration. The new coordinate system is valid during elaboration of the *picture* that begins with the *subspace*.

The origin of the *subspace* follows from *position* and *shift*. In the CURRENT case, it is the untransformed picture position (PP, see 3.5.) shifted by the vector corresponding to *shift*, otherwise it is the origin defined by the previous *subspace* selection, shifted by the same amount.

In a *subspace* selection, two dimensions are involved, the dimension of the environment in which the selection occurs, and the dimension of the subspace being selected, specified by *dim*. This latter dimension becomes the new dimension of the environment, during the elaboration of the *picture* which starts with the *subspace*. *axes* must contain a number of *dimensional_values*, equal to the value of *dim*. These *dimensional_values* specify the direction of the coordinate axes and the units in which coordinates are measured, in the subspace. The directions are those of the vectors defined by *dimensional_values*, the metric follows from the rule, that those vectors have unit length in the *subspace*. It should be noted that we do not require that these axes are orthogonal, only that they are defined by independent vectors.

The general dimension rule excludes the selection of a *subspace* with higher dimension than the environment in which the selection occurs. The *dimensional_values* required to specify such a selection would have been of a higher dimension than the dimension of the environment and are thus illegal.

Let the *dimensional_values* (considered as column vectors) defining the *subspace* be extended with a zero at the bottom, and the result be denoted by the columns $D_1, \ldots, D_n$. Let the column vector from the previous origin to the new origin be extended with a one at the bottom, and be denoted by D. Then the transformation from subspace to environment is given by the matrix:

$$(D_1, \ldots, D_n, D)$$

## 3.4. Attributes

The syntax rules describing the various *attributes* are:

```
attribute:      ABS basic_attribute |
                REL basic_attribute |
                basic_attribute $
basic_attribute: attribute_class |
                aname |
                { attributes } |
                NIL $
attributes:     attribute |
                attributes ; attribute $
attribute_class: transformation |
                detection |
                style |
                control |
                pen |
                coordinate_mode |
                visibility $
```

With every *attribute_class* (except *control*), corresponds an *attribute_match*, defined by the syntax rules:

```
attribute_matches: empty |
                attribute_matches
                               deny attribute_match $
attribute_match: TF
                DT |
                ST |
                PN |
                CM |
                VS $
deny:   empty |
                ~ |
                NOT $
```

*Attribute_matches* are part of *picture_elements*.

An *attribute_class* is a terminal *attribute* node. The *attribute_class* values can range from simple constructs to complex structures. For each class, however, the format of the value is fixed. Here we must differentiate between a complete class value (which as such is not a syntactical category) and a contribution to such values by an individual *attribute_class* element (which is a terminal production of *attribute_class*).

For some *attribute_classes* (e.g. *style* and *pen*) the class value is described as an ordered n tuple of so called atoms. An atom has the following properties: It can have a unit value with respect to combining:

$$a * unit = unit * a = a$$

Each element of such an *attribute_class* specifies precisely one atom. Hence, for a complete class value at least n *attribute_class* elements are required. A unit class value consists of the n unit atom values. A set of $k < n$ different atom values can be expanded to a class value by adding a unit value to each missing atom. In this sense each individual atom can also be considered as a class value ($k = 1$).

Unit values cannot (and need not) be specified. They only serve to simplify the semantic description.

Apart from a unit value for *attribute_classes* there exists a default value for each *attribute_class* and also for each atom. This value is taken when an *attribute_class* must be applied to a *picture_element* and the unit value (for a class or atom) is specified. The default value can also be selected explicitly as *attribute_class* element or through the *attribute_matches*.

In the following, we will elucidate, how *attributes* act upon *picture_elements*. From a semantic point of view, two major steps are needed in the process of applying *attributes* to a *picture_element*.

In the first step, the attribute structure is simplified by applying combination rules for *attributes*, to the effect that *attribute* nests and nested "WITH ... DRAW" constructions are removed. By this process an ILP program can be converted into a so called basic ILP program, that consists of a linear list of "WITH A DRAW P" constructions, where A denotes a linear list of *attribute_class* values and P a *picture_element*. The linear list A contains all *attributes* that have been specified for P. The order of the *picture_elements* in the basic ILP program must be the same as in the picture tree when traversed in preorder. The important reason for this is that each *picture_element* partly sets the environment for its successors. *subspaces* and *picture_elements* can only be elaborated when the environment is known. For a given *picture_element* the major steps must be fully completed before the same steps can be taken for its successor. The first algorithm of the first step takes care of all environment specifications for the subspaces. From there all steps can be carried out independent of any environment. When finally the *picture_element* itself is elaborated, the environment is first used to complete the *picture_element*, next the *attributes* are applied, then the element is drawn and finally the environment is updated. As already stated, there is a correspondence between ILP programs and directed acyclic graph structures. For convenience, we will split the description of the first step in two parts. The first part is described as a conversion of graphs (section 3.4.1.), the second as a conversion of programs (3.4.2.).

In the second major step, the *attribute_class* elements from each "WITH ... DRAW" construction of the basic program, are concatenated or combined, and then applied one after the other. The general features of this step are described in sections 3.4.2.2. and 3.4.2.3., while the aspects that are characteristic for individual *attribute_classes*, are described class wise in sections 3.4.4. till 3.4.10..

### 3.4.1. Decomposition of the picture tree

There exists a unique path in the picture tree (see 3.2.) from the root to each *picture_element*, called element path. For every element path, we will construct a new tree, called element tree, as follows:

Algorithm ET: construct an element tree

ET1 Start with a node of the form "WITH U DRAW NIL", where "U" contains the *attribute_class* unit value for each class. Traverse the element path.

ET2 Every time a *subspace* node is encountered, generate the corresponding *subspace* transformation S (see 3.3.), using the *subspace* specification and the value of the pen position (see 3.3.2.), which is given in the environment. Replace in the original program the *subspace* by "WITH S DRAW (so the *subspace* is evaluated only once and in the right environment). Continue with the same node.

ET3 Every time a "WITH...DRAW" node is encountered, replace the last "NIL" of the element tree by "WITH A DRAW NIL. Here "A" is the *attribute* of the node at hand.

ET4 When the *picture_element* is reached it replaces the last NIL of the element tree.

A picture tree with *picture_elements* is converted by ET into a semantically equivalent picture forest $T_{.}$, ..., $T_n$ of element trees. Tree $T_i$ contains *picture_element* $P_i$, which is the i-th *picture_element* encountered, when the picture tree is traversed in preorder.

With every element tree $T_1$, corresponds an *attribute* $A_i$. A description in the form of a string of every $A_i$ is produced by algorithm ETA , and modified by algorithms RAP and LIN. In this and the following sections manipulations on descriptions of picture and attribute graphs are used. The algorithms as presented, ignore the layout characters in such descriptions.

Algorithm ETA: compute element tree attributes

ETA1 Initialize $A_i$ with "REL{". Traverse $T_i$ from root to leave.

ETA2 Every time a "WITH X DRAW Y" node occurs, expand all *attribute_class* elements contained in X (see 3.4.) which results in X'. Next append "X';" to the right of $A_i$.

ETA3 Finally, replace the last (rightmost) ";" of $A_i$ by "}".

The application of algorithm ETA results in an ILP program with body:

```
WITH A1 DRAW P1;
WITH A2 DRAW P2;
        .
        .
        .
WITH An DRAW Pn;
```

## 3.4.2. Attribute mixing

The process of combining and simplifying *attributes* that will be described in 3.4.2.1. and 3.4.2.2. is called attribute mixing. It can be applied to any sequence of *attributes*, whether this sequence is derived from an element tree or not. The result of mixing is again a construction of type *attribute*.

### 3.4.2.1. Simplification of attributes

Every $A_i$ in the program produced by algorithm ETA, is simplified in the following steps:

Algorithm RAP: remove *anames*, add prefix

RAP1 Replace all references to *attributes* in $A_i$ by their body, e.g. for every *aname* substitute the *attribute* from the *attribute_pack* with that aname. Repeatedly perform this step, as long as references to *attributes* are present (note that recursion is not allowed).

RAP2 Prefix every not prefixed "{" or *attribute_class* with "REL".

Finally, $A_i$ is converted into one list without sublists of *attribute_classes* .

Algorithm LIN: linearize *attribute*

LIN1 Find a construction B of form ABS { *attributes* } or REL { *attributes* } which contains only prefixed *attribute_class* elements. When no such construction can be found, the algorithm terminates here.

LIN2 Sort the elements of B class wise, without disturbing the sub-order in each class (result: construction B').

LIN3 Apply the following substitutions to adjacent elements x and y of B', belonging to the same *attribute_class* until no further substitutions are possible:

"REL x; ABS y" -> "ABS y"
"ABS x; ABS y" -> "ABS y"

whenever x is not a *subspace* transformation, and:

"REL x; [REL S]$_n$; ABS y" -> "[REL S]$_n$; ABS y"
"ABS x; [REL S]$_n$; ABS y" -> "[REL S]$_n$; ABS y"

where S is a *subspace* transformation.

The last two substitution rules guarantee, that *subspace* transformations are preserved, when other transformations are invalidated by a prefix "ABS". The result of this step is construction B''. As a consequence, in B'' only the leftmost element belonging to a certain *attribute_class* (called leftmost class element), can have prefix "ABS".

LIN4 Replace the prefix of every leftmost class element by the prefix preceding the left bracket. Next remove (the only and outermost) "ABS {" or "REL {" and "}". The result is labelled B'''.

LIN5 Finally, replace the original construction B in $A_i$ by B''' and continue at LIN1.

As a result of algorithms RAP and LIN, the *attributes* $A_i$ in the program produced by ETA are transformed into a simple list of prefixed *attribute_class* elements. It should be noted that, occurrencies of prefix "ABS" have been removed.

### 3.4.2.2. Attribute types and concatenation

According to semantic properties, *attribute_classes* are divided in type 1, type 2 and type 3 classes, and correspondingly, *attribute_class* elements in type 1, type 2 and type 3 elements. Different methods are used for the description of the semantics of these three types of *attribute_classes*. In fact, *attribute_classs of type 3 are the most general and allow the most powerful attributes* to be formulated. As a consequence all type 1 and type 2 *attribute_classes* can also be described with the description method used for type 3. However, type 1 and type 2 *attributes* have some special properties, that make possible a considerably easier semantic description.

The formal classification, based on the semantic description method, coincides with a less strict division of *attributes*, according to their usage: Type 1 *attributes* generally describe purely geometric information. Type 2 *attributes* associate information with *picture_elements*, which logically speaking, can be meaningful for an isolated *picture_element*, irrespective of the structure of the picture graph it belongs to. Type 3 *attribute* elements associate external properties with *picture_elements* that are not necessarily meaningful for an isolated *picture_element*, but preserve information about the structure of the element path leading to it. These *attributes* will probably be used mainly for the association of non-graphic information with pictures.

The description of the semantics of *attributes* always consists of at least two steps:

-- describe the semantics of individual elements from the class.

- -- describe the semantics of combining a sequence of class elements into an entity with already defined semantics.

The differences in description method used in these steps, determine the classification of the *attribute*.

The following discussion of *attribute_class* types will be based on sequences of *attributes*, occurring in a list as produced by algorithm LIN, called a LIN list.

- -- Type 1: The semantics of the *attribute_class* element are described by defining for every *picture_element*, a set of *picture_elements*, called the result set. The drawing (in user space), corresponding to the result set is by definition identical to the drawing corresponding to the *picture_element* with the type 1 *attribute* applied to it. The *picture_element* "NIL" is unaffected by type 1 *attribute_class* elements. The semantic description is extended to all *attributes* from a certain type 1 class in the LIN list, by the following rule: The *attributes* are applied one at the time, in the order in which they are encountered when the LIN list is scanned from right to left. In other words each *attribute_class* element is applied to all *picture_elements* of the result sets produced by the previous class element.

- -- Type 2: The semantics of these *attribute_class* elements cannot be described in terms of other ILP primitives, but must be defined in an ad hoc manner. For type 2 classes there exists a <u>concatenation</u> rule of the form:

  REL A <> REL B -> REL C,

  where A, B and C are elements from the same class, and "<>" denotes concatenation. On a sequence of *attributes* from the class, this rule can be applied repeatedly, to combine them into one *attribute_class* value.

  Hence, the semantics for all the type 2 *attributes* from a certain class, contained in a LIN list, follows from the semantics of a single class value and the concatenation rules.

The type 2 *attribute* and its concatenation rules are formulated in such a way, that associativity from the left is obtained. As will become clear in 3.4.3., this will lead to the possibility of an implementation that uses a stack.

-- Type 3: The definition method for class elements depends on the class involved, and will be often quite ad hoc. However, combining a sequence of elements proceeds in a methodical manner, which is the same for all type 3 *attributes*.

  With every type 3 *attribute_class*, a so-called application universe is associated. With (any sequence) of type 3 class elements, corresponds one element from the application universe. An application universe element is applied to *picture_elements* during elaboration, instead of the sequence of *attributes* in a LIN list, to which it corresponds. So the semantics of collections of type 3 *attributes* in a LIN list is defined by:

  - The correspondence between sequences of type 3 elements and an application universe element.
  - The semantics of every application universe element.

The definition method for the semantics of application universe elements depends on the *attribute_class* involved. The correspondence is defined with the help of combination rules, that work according to the following general scheme:

- With every element from a given *attribute_class*, corresponds an element from the application universe.
- These application universe elements are combined into one, using combination rules of the form:

  $$C_1 <> C_2 -> C_3,$$

  where <> denotes combination, and $C_1$, $C_2$ and $C_3$ are application universe elements.
- The combining operator is left associative.

At present, *detection* is the only type 3 *attribute_class*.

### 3.4.2.3. Semantics of attribute class lists

To continue elaboration all concatenations and combinations in the list of prefixed *attribute_class* elements $A_i$ in the program produced by algorithm LIN are performed. Thereafter, for every type 2 class, there is at most one single element left in $A_i$, and for every type 3 class, one application universe element is found, that is included in $A_i$.

The semantics of $A_i$ now follow from the priority rules:

-- first all type 1 elements are applied, next the type 2 elements and type 3 (application universe) elements.

-- the priority, when not fixed by the above rule, is explicitly defined whenever necessary. (At present, all type 2 and type 3 classes may have the same priority).

REMARK

When, in extensions of ILP, new type 3 *attribute_classes* are introduced, priority rules become extremely important. It can be considered, for instance, to give certain type 3 classes priority over certain type 2 classes. This would allow, the modification of type 2 elements by type 3 elements.

As an illustration of the semantics of type 1 *attributes* consider the ILP statement:

WITH { A ; B } DRAW P .

The meaning is, that first P is subjected to attribute B, and then the result to attribute A. Especially when the *attributes* are transformations, this application order is different from the one mostly used in computer graphics systems.

### 3.4.3. State component and state

Before we describe the semantics of individual *attributes*, we give definitions for the concepts of state component and state.

If the mixing process is applied to the *attribute* of one single "WITH...DRAW" node, the result is called a <u>state component</u>.

Let A be a "WITH...DRAW" node in a picture tree. The result of mixing all *attributes* encountered on the path from the root up to and including A, is called a <u>state</u> S.By definition S is the state for all *pictures* corresponding to nodes that are:

-- descendants of A, and

-- not reached from A via "WITH...DRAW" nodes.

It should be noted, that state components and states both consist of *attribute_class* values and application universe elements.

Let, during elaboration, $W_1$ and $W_2$ be two successively encountered WITH...DRAW nodes. The state calculated from $W_1$ and its predecessors, is called the <u>current state</u> of the elaboration process, from the moment the *attribute* of $W_1$ has been elaborated, until the start of the elaboration of $W_2$.

Finally, we signal an important property. Let $S_1$ be the current state prior to elaboration of a *picture* $P_1$. Let $P_1$ contain a "WITH...DRAW" node W and let $S_2$ be the state component of W. Let $P_2$ be a *picture* contained in W. The current state, during elaboration of $P_2$ is obtained by mixing $S_1$ and $S_2$. Clearly an efficient implementation of ILP programs requires an efficient stack mechanism for states.

### 3.4.4. Transformations

Transformations are type 1 attributes (see 3.4.2.2.). From a semantic point of view, they are applied one after the other, although in an actual implementation, matrix transformations (see below), will probably be concatenated. The result of applying a transformation T to a *picture_element* P can be described as an ILP program P' that consists of a linear list of transformed *picture_elements*. All transformations except *port* are matrix transformations.

The semantics of matrix transformations have some general aspects that will be discussed first.

When a matrix transformation is applied to a *coordinate_type picture_element* (see 3.5.1.) the resulting ILP program P' consists of one *picture_element* of the same category as the original *picture_element*.

*picture_elements* (*text* excluded), either contain a row of *coordinates* (e.g. *line*) or generate a sequence of *coordinates* (*generator*). A *coordinate* contains a *dimensional_value* which, if the dimension of the environment is n, consists of the row of *values* $[v_1, v_2, \ldots, v_n]$. With such a *dimensional_value*, the corresponds a column vector v with n+1 components, defined as:

$$v = \left|\begin{matrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \\ 1 \end{matrix}\right|$$

In the sequel this extended form (i.e. homogeneous coordinates) will be used).

With every matrix transformation either a n,n-matrix, or a (n+1),(n+1)-matrix can be associated, where n is again the dimension of the environment. n,n-matrices will be extended to (n+1),(n+1)-matrices by first extending every row with a rightmost element with value zero, and then adding an extra (bottom) row of n+1 elements which are all zero, except for the rightmost one, which has value one.

Hence every matrix transformation is represented by a (n+1),(n+1)-matrix A. To vector v corresponds a transformed vector w, defined by:

$$w = A * v$$

where "*" denotes ordinary matrix multiplication. Because column vectors are used, the order of multiplication must be matrix times vector.

To vector w corresponds a *dimensional_value*

$$\left[ \frac{w_1}{w_{n+1}}, \frac{w_2}{w_{n+1}}, \ldots, \frac{w_n}{w_{n+1}} \right]$$

which is called the transformed of *dimensional_value* $[v_1, \ldots, v_n]$. The result of applying a matrix transformation to a *picture_element* is now obtained by replacing all (generated) *dimensional_values* by their transformed *dimensional_values*.

REMARK

> The detailed definition of the various *transformations* that will be given below, does not strictly apply to *picture elements* of type *text*. Although it seems intuitively clear what is meant by, for instance, an arbitrary clipped or rotated *char*, we cannot formally tell, which *char* is the result of the *transformation*. Furthermore, it is likely that permitting arbitrary *transformations* on *text* objects, precludes the use of character hardware and a reasonably efficient implementation. Hence a "subset" of *transformations* must be defined from which an element will be automatically selected when a general *transformation* is applied to *text*. A further problem with *text* is the interaction between elements from different *attribute_classes*. Whether, for instance, a character whose center lies near a *window* boundary (see 3.4.4.8.) has to be clipped, depends on the value of the *typographic attribute* (which influences its shape and size). A solution might seem the choice of priorities, but this leads to conflicts: If *line_style* must not be transformed, *transformations* must have priority over *typographic*. If characters must be properly clipped at *window* boundaries, their *size* must be evaluated before the *port* transformation is applied, e.g. *typographic* must have priority over *transformations*.

The *attribute_match* corresponding to *transformations* is TF. ~TF switches off all *transformations* or, one might say, replaces the overall transformation by the default transformation (see below). TF reinstalls the overall transformation.

The *transformations* are listed in the following syntax rules:

```
transformation:       rotate |
                 scale |
                 translate |
                 matrix |
                 projection |
                 affine |
                 homogeneous_matrix |
                 port $
rotate:          ROTATE value
                         AROUND invariant $
scale:           SCALE dimensional_value $
translate:       TRANSLATE dimensional_value $
matrix:          MATRIX matrix_value $
affine:          AFFINE matrix_value
                         dimensional_value $
projection:      PROJECT eye_position
                         ON projection_space $
homogeneous_matrix:
                 HOMMATRIX homogeneous_matrix_value $
port:            window |
                 window , viewport $
```

### 3.4.4.1. Rotation

An elementary rotation in n-dimensional Euclidean space can be specified by:

-- Selection of a plane V in the n-dimensional space.

-- Selection of a point P in this plane.

-- Definition of a rotation angle phi.

The matrix R:

$$\begin{vmatrix} \cos \text{phi} & \sin \text{phi} & 0 & \cdots & 0 \\ -\sin \text{phi} & \cos \text{phi} & & & \\ 0 & & 1 & & \\ \cdot & & & \cdot & \\ \cdot & & & \cdot & \\ \cdot & & & \cdot & \\ 0 & & & & 1 \end{vmatrix}$$

describes this elementary rotation under the condition that a new set of coordinate axis $x_1, \ldots, x_n$ is chosen with:

-- The origin coincident with P.

-- $x_1$ and $x_2$ contained in V.

When the matrix that transforms the original coordinate axis into the set $x_1, \ldots, x_n$ is given by T, the rotation in the untransformed coordinate system is given by

$$T^{-1} * R * T$$

A rotation in n-dimensional Euclidean space can be considered as the product of a number of elementary rotations.

In ILP, an elementary rotation is syntactically specified by:

```
rotate:        ROTATE value
                        AROUND invariant $
invariant:     ( dimensional_values ) $
```

The rotation angle is determined by *value*, while the rotation plane and point are specified by *invariant*. The *invariant* contains a number of *dimensional_values* which is one less then the dimension of the environment. The first *dimensional_value* specifies the rotation point P, the following define (n-2) independent vectors orthogonal to the rotation plane. Rotation takes place clockwise (defined with respect to the normal from the origin to the plane), through a number of degrees, specified by *value*.

In the two dimensional case, the set of n-2 vectors is empty, in the three dimensional case it is the familiar axis of rotation. As a consequence, in the two or three dimensional case a general rotation can be specified by one single *rotate*.

REMARK

> It should be clear that we are confronted with a tradeoff here: if the dimension of the environment is less then four, it is economical to specify a plane by its normals, if the dimension is more then four, specifying the plane with two vectors contained in it is cheapest. We have chosen the first alternative.

### 3.4.4.2. Scale

By scaling, the *values* of the *dimensional_value* of a *coordinate* are changed independent of each other. Scaling can be represented by a diagonal matrix. The syntax rule is:

*scale*: SCALE *dimensional_value* $

Each *value* in the *dimensional_value* specifies a diagonal element of the unextended transformation matrix.

### 3.4.4.3. Translate

A translation maps all points in user space on points displaced by a fixed amount. Translation is syntactically described by:

*translate*: TRANSLATE *dimensional_value* $

Each *value* in the *dimensional_value* specifies the displacement along the corresponding coordinate axis.

In an n-dimensional environment a translation, characterized by *dimensional_value* $[v_1, \ldots, v_n]$, is represented by a $(n+1),(n+1)$-matrix with diagonal elements of unit value, the rightmost element of the k-th row $(k = 1, \ldots, n)$ with value $v_k$, and all other elements zero.

### 3.4.4.4. Matrix

A matrix transformation specifies a linear transformation of the user space. A matrix transformation is syntactically described by

*matrix*: MATRIX *matrix_value* $

Each *dimensional_value* in the *matrix_value* (see 3.3.1.) specifies a column in the transformation matrix. As a consequence of the general dimension rule (see 3.3.1.), a matrix contains a number of rows and columns equal to the *dimension* of the environment.

### 3.4.4.5. Projection

Projection is syntactically described by:

*projection*: PROJECT *eye_position*
ON *projection_space* $
*projection_space*: *dimensional_value* |
ORIGIN *dimensional_value* $
*eye_position*: *dimensional_value* |
PARALLEL *dimensional_value* $

The space on which projection takes place, is specified by *projection_space*. It is a space of dimension one less than the environment, perpendicular to the the vector specified by *dimensional_value* in *projection_space*. Nevertheless, the projected image has the dimension of the environment, but there exists a linear relation between its coordinates, for instance, $x_n = 0$. If only a *dimensional_value* is present in the specification of the *projection_space*, the space contains the end point of the vector defined by *dimensional_value*. If the keyword ORIGIN is used it contains the origin of the current coordinate system.

The type of projection is determined by the specification of *eye_position*. If a *dimensional_value* is used, a central projection is applied, with the point with coordinates corresponding to *dimensional_value* as centre. In the other case, projection is parallel, to a direction defined by *dimensional_value*.

Let the coordinate axis of an n-dimensional Euclidean space be $x_1, \ldots, x_n$. A projection with the point $(x_1 = x_2 = \ldots = x_{n-1} = 0, x_n=c)$ as centre, on the space $x_n=0$ is given by the $(n+1),(n+1)$-matrix P.

$$\left|\begin{array}{ccccc} 1 & & & & \\ & \cdot & & & \\ & \cdot & 0 & & \\ & 0 & \cdot & & \\ & & 1 & & \\ 0 & \ldots & 0 & 0 & 0 \\ 0 & \ldots & 0 & -1/c & 1 \end{array}\right|$$

In this matrix, constant c equals the distance from the *eye position* to the *projection space*. Parallel projection is obtained by replacing $-1/c$ by zero. Let $T_1$ be the transformation that translates a projection space to the origin, $T_2$ the transformation that rotates the normal on this space to the direction of coordinate axis $x_n$. The projection is then given by the matrix:

$$T_1^{-1} * T_2^{-1} * P * T_2 * T_1$$

In the following example a three dimensional environment is assumed, with coordinate axes denoted by x, y and z.

PROJECT { 1, 1, 1 } ON { 0, 0, 1 }
defines a central projection on the plane z=1. With the point x=1, y=1, z=1 as projection centre.

PROJECT PARALLEL { 0, 0, 1} ON ORIGIN { 0, 0, 1}
defines a projection parallel to the z-axis on the plane z=0.

### 3.4.4.6. Affine

*affine*: AFFINE *matrix_value*
*dimensional_value* $

An affine transformation is represented by a square matrix with:

- A number of rows (columns), one more than the dimension of the environment.
- A bottom row with all elements zero, except for the rightmost, which has value one.

If the dimension of the environment is n, *matrix_value* specifies a n,n-matrix A, *dimensional_value* a column of n elements. The resulting affine matrix is:

```
<------------ n + 1 ----------->

|  --------------------   ---  |
| |                    | |   | |
| |                    | |dim| |
| |   matrix value     | |   | |
| |                    | |val| |
| |                    | |   | |
|  --------------------   ---  |
| 0      ...          0    1   |
```

### 3.4.4.7. Homogeneous matrix

In an n-dimensional environment a *homogeneous_matrix* transformation is represented by a (n+1),(n+1)-matrix. Every element of this matrix is explicitly specified with the help of the following syntactical construction :

```
homogeneous_matrix:
                    HOMMATRIX homogeneous_matrix_value $
homogeneous_matrix_value:
                    [ homogeneous_dimensional_values ] $
homogeneous_dimensional_values:
                    homogeneous_dimensional_value |
                    homogeneous_dimensional_values ,
                              homogeneous_dimensional_value $
homogeneous_dimensional_value:
                    [ values ] $
```

The *homogeneous_matrix_value* consists of (n+1) *homogeneous_dimensional_values*, which each specify a column of the matrix. Every *homogeneous_value* consists of (n+1) *values*, which each specify an element of the column.

### 3.4.4.8. Window and viewport

The *port* transformation is the only mapping from user coordinates into user coordinates, that cannot be described by a matrix.

The transformation is syntactically described by:

```
port:          window |
               window , viewport $
window:        WINDOW ( dimensional_value ,
                        dimensional_value ) $
viewport:      VIEWPORT ( dimensional_value ,
                        dimensional_value ) $
```

*window* and *viewport* select rectangular areas in user space.

The *dimensional_value* pairs in both the *window* and *viewport* definition, determine the end points of a principal diagonal of the *window* and *viewport* areas. As a consequence of the general dimension rule (see 3.3.1.), the dimension of a *window* or *viewport* is equal to the dimension of the environment in which the *window* and *viewport* are specified.

The selected areas are fully determined by the requirements that they are block shaped, and that they have their edges parallel to the coordinate axis.

When *coordinate_mode* (see 3.4.8.) has value FREE, the relative position of the *ports* and the free coordinates may not be known from context. The *dimensional_values* in *port* denote absolute positions in the current coordinate system. Selection of a new origin at the untransformed pen position with the help of the SUBSPACE mechanism, solves this problem.

If the *port transformation* does not contain a *viewport*, only a clipping boundary is defined. Only parts of the picture, that lie inside the window, are preserved. Without going into detail, we summarize in the table below, for every type of *picture_element* the possible elements of the result set (see 3.4.2.2.), if this set is not empty.

| *picture_element* | result set elements |
|---|---|
| POINT | zero or one POINT |
| LINE | zero or more LINE's |
| CONTOUR | zero or more LINEs or CONTOUR |

The *picture_element generator* ultimately generates elements contained in this table, which determines its behaviour under the *port* transformation. The effect on TEXT has not yet been decided upon.

If the *port* contains a *viewport* (which must be preceded by a *window*), first the *window* is applied, whereafter the matrix transformation, that maps the *window* area onto the *viewport* area is applied.

The following observations can be made:

-- The effect of the application of a number of *windows* (separated by matrix transformations) is identical to the effect of clipping to the intersection of the leftmost *window* and the (transformed) further *windows*.

-- When two *window*, *viewport* pairs are applied, the visible part of the *viewport* area of the second pair, is always contained in the *viewport* area of the first pair.

### REMARK

Is it useful to incorporate negative windows, i.e. windows that remove all *picture_elements* <u>inside</u> the window area?

The nesting of windows allows the construction of window areas of very complicated shapes. How inefficient becomes the clipping algorithm?

### 3.4.4.9. Subspace and transformations

The major similarity between *subspaces* and *transformations* is that the effect of any matrix *transformation* (except *homogeneous_matrix* or *projection*), can also be achieved by a *subspace* transformation.

The basic differences between *subspace* and matrix *transformations* are the following:

- *subspace* transformations transform coordinate systems and hence influence all *dimensional_values* in a *picture* that begins with a *subspace*. Note that the *dimensional_values* used to specify matrix *transformations* are affected too.
- Matrix *transformations* transform actual coordinates of "objects" defined in a given coordinate system. Hence they are only applied to the *dimensional_values* of *coordinate_values*. For instance, *dimensional_values* used to specify other *transformations* are not affected.
- A *subspace* can reduce the dimension.
- A *subspace* transformation is not affected (switched off) by a prefix "ABS", which occurs in a *transformation attribute*.
- A *subspace* transformation is not affected by the *attribute_match* for *transformations*.

### 3.4.5. Style functions

#### 3.4.5.1. Introduction

Style functions describe what kind of lines, points and characters (and in the future shades and greyscales) are to be produced by a drawing machine. The description is as machine independent as possible. In view of the enormous variety of drawing machines, the *style*-function package has to be extendible and is inevitably incomplete.

The given functions are all specified in such a way that the same *style* functions produce similar results on all drawing machines, that is, if they are expressible in terms of the existing hardware. However no functions exist in ILP to express the quality required of the result of application of a *style attribute*. *) When necessary an extra software layer has to be provided to produce or approximate *styles* for which no direct hardware functions are available. Since *style* has more to do with taste and clearness of expression than with accuracy, it will cause no trouble when *style* is not defined with mathematical precision (as would be the case with, say, *transformations*).

The three classes of *style* functions that exist so far, e.g. *line_style*, *point_style* and *typographic* are mutually unrelated. The syntax for *style* is:

*style*: *line_style* |
*point_style* |
*typographic* $

*style* is a type 2 *attribute_class* Its class value is a 12-tuple with atoms represented by PERIOD, MAP, THICK, FONT, SIZE, ITALIC, BOLDNESS, POINTSTYLE FONT, POINTSTYLE SIZE, POINTSTYLE ITALIC, POINTSTYLE BOLD, POINTSTYLE *token*. Let $C_1$, ..., $C_{12}$ and $C_1'$, ..., $C_{12}'$ denote *style* class values. Then the concatenation rules for *style* are:

---

*) It can be considered to parameterize quality outside ILP by providing a quick-and-dirty, and a high-quality mode for the representation of the same *style*.

$\mathrm{REL}(C_1, \ldots, C_{12}) <> \mathrm{REL}(C_1', \ldots, C_{12}') ->$
$\mathrm{REL}(C_1\%C_1', \ldots, C_{12}\%C_{12}')$

where <> denotes concatenation and % is defined by:

$C_i\%C_i' -> C_i$, when $C_i'$ is a unit atom,
$C_i\%C_i' -> C_i'$, otherwise.

Hence in the "ABS" case, *style* is completely redefined, partly by unit atoms. In the "REL" case, only explicitly specified atoms are changed.

### 3.4.5.2. Linestyle

*Line_style* conforms to the syntax:

*line_style*: PERIOD ( *period_description* ) |
MAP ( *value reset* ) |
THICK ( *value* ) $

*line_styles* are applied to *picture_elements* of type LINE. They are also applied when the LINE is produced indirectly, through a *contour*, or a *generator* .

The *line_style* determines what will be drawn along the straight lines that connect the successive positions of the *picture_elements*.

The *line_style* can produce a large variety of dotted and dashed lines. The definition of such a pattern goes in two steps.

### 3.4.5.2.1. Period definition

PERIOD describes a basic pattern which is repeatedly produced going along the *line*.

*period_description*: *dash* |
*dash* , *gap* |
*dash* , *gap* , *dash* $
*dash*: DOT |
*value* $
*gap*: *value* $

The period is defined on a straight line piece of 100 units in length, which is filled out by:

$\mathrm{dash}_1 \quad \mathrm{gap}_1 \quad \mathrm{dash}_2 \quad \mathrm{gap}_2$

Hence $\mathrm{dash}_1 + \mathrm{gap}_1 + \mathrm{dash}_2 + \mathrm{gap}_2 = 100$. $\mathrm{Gap}_1$ through $\mathrm{gap}_2$ may be omitted, implying that the first missing one adds up to 100. $\mathrm{Gap}_2$ always is omitted. If *dash* has value DOT, a point is produced on the spot with has a length of 0 units with respect to the period. This concept DOT is the same, as the one used in *point_style*, see 3.4.5.4..

Examples:

| | |
|---|---|
| PERIOD (100) | Solid line. |
| PERIOD (DOT) | One point at the beginning of each period. |
| PERIOD (0,100) | Blank (invisible) line. |
| PERIOD (50) | Dashed line with gaps equal to dashes. |
| PERIOD(25,50) | Dashed line with gaps equal to dashes. It starts however, with a half dash. |

### 3.4.5.2.2. Map definition

The *value* of MAP specifies the actual length of the pattern described by *period_description*. This length is defined in user unit coordinates, valid at the root. A pattern of the given actual length is rolled along the line, to produce the style.

*reset*: RESETCOORDINATE |
CONTINUE |
RESETLINE $

The three different values for *reset* tell, whether the periodic pattern has to be continued from one LINE to the next (value: CONTINUE), to be reset at the start of every new LINE (value: RESETLINE) or to be reset whenever a new *coordinate* within a LINE is encountered (value: RESETCOORDINATE).

REMARK

It might be considered to link the length unit, used in the *style attribute* to the metric of the *subspace* in which this *attribute* is used. This will lead to complications for non-orthogonal *subspaces*, however.

### 3.4.5.2.3. Thickness

The value of THICK determines the linewidth, when drawing LINEs. It is expressed in the same unit as used in the map definition for linestyles (see above). Thick lines are cylindrical. They are drawn with constant diameter. Thick lines are not modified by *projection transformations*, i.e. they do not become conic.

REMARK

Is it a good decision to use the same unit for period description and thickness?

The connection method for thick lines and the visual representation of their end points is not yet defined, but several problems may be exspected in this area.

### 3.4.5.3. Typographic style

The *typographic style* is in fact nothing else than a means to specify a given character set out of the available sets.

```
typographic: TYPFAULT |
             font |
             size |
             italic |
             bold $
```

Characters are grouped in sets of at most 256 tokens, called a basic set.

A basic set can contain tokens of any kind, up to complete *pictures*. In ILP they will, however, be considered as characters. Their internal structure is inaccessible and can therefore not be manipulated.

A *font* consists of a basic set plus a description how the character data are to be interpreted, and what the effect of *size*, *italic* and *bold* is, on the individual tokens. In view of the use of *typographic* for *pointstyles* also a default token for DOT must be given. A *font* is selected by the *font attribute*. The tokens can be modified, by explicitly specifying *size*, *italic* and *bold*.

It is clear, that the *typographic attribute*, allows the specification of an unlimited collection of characters. TYPFAULT is shorthand for selection of *font*, *size*, *italic* and *bold*. Its effect is device dependent. It denotes a character set, whose elements can be drawn as efficient as possible on the device at hand, if necessary disregarding high quality demands (see 3.4.5.1.).

### 3.4.5.4. Point style

The syntax rules for point style are:

```
point_style: DOT |
             POINTSTYLE typographic |
             POINTSTYLE marker $
```

*marker* selects a token from the font specified by POINTSTYLE *font*. This token is modified by POINTSTYLE *size* etc. At point positions, this token is displayed, drawn in a centered fashion. It will be drawn in the $x_1$, $x_2$ plane of the current coordinate system, with its "bottom line" parallel to the $x_1$ axis. When the alternative DOT is used, a device dependent "point" will be displayed. DOT is shorthand for a device dependent character set (*typographic*) and for a specific token (the point) out of this set. When only POINTSTYLE *typographic* or POINTSTYLE *marker* is specified, the other atom of *point_style* has its default value (see 3.4.12.).

### 3.4.6. Pen functions

Pen functions determine the reproduction method to be used when a *picture_element* is drawn. As a consequence, pen functions influence only the final appearance of a drawing but do not affect the structural information contained in it. The effect of pen functions can not be described in terms of ILP primitives. *pen* is a 3-atomic type 2 *attribute*, its concatenation rules are analogous to those of *style*.

The syntax for *pen* is:

*pen*: PENFAULT |
*contrast* |
*intens* |
*colour* $

Just as in the case of TYPFAULT, PENFAULT selects device dependent values for *contrast*, *intens* and *colour*.

#### 3.4.6.1. Contrast

The syntax of *contrast* is:

*contrast*: CONTRAST ( *value* , *value* ) $

It is assumed, that any physical drawing device can draw with a minimal and a maximal intensity, which are the end points of its physical intensity range. (The maximal intensity always represents "light", the minimal "dark", i.e. on a plotter, these two intensities are determined by the reflectivity of the paper, respectively the blackness of the ink.)

For every device a mapping must be defined from the interval [0,100] (the contrast range) to the physical intensity range. *contrast* specifies a subrange of the contrast range, i.e. fixes indirectly the lowest and highest physical intensity, that can be used.

Examples:

CONTRAST ( 0 , 100 ): highest possible contrast.
CONTRAST ( 50 , 50 ): no contrast, one intensity.

### 3.4.6.2. Intensity

Intensity is syntactically described by:

*intens*: INTENS ( *value* ) $

and determines the brightness of the registration method. *value* may have as value a real number from the interval [0,100]. The corresponding physical intensity used by the drawing device is determined as follows. There is a linear mapping from the intensity range [0,100], to the contrast range [a,b] (0 <= a <= b <= 100), specified by *contrast*. So, a value in the intensity range determines a value in the contrast range, which determines the physical intensity, via the mapping from the contrast range to the physical intensity range.

Examples:

INTENS(100) = maximal intensity
INTENS(0) = minimal intensity

There is an important distinction between invisible lines (i.e. lines drawn with value INVISIBLE for *visibility*) and lines with zero intensity. In the former case the order in which the invisible lines are drawn is not defined and consecutive invisible lines may even be replaced by one invisible line. In the latter case the drawing order is completely defined and the kind of optimizations just mentioned are not allowed.

### 3.4.6.3. Colour

On a mono colour (black/white) drawing device, *contrast* and *intensity* are sufficient for the specification of the different shades of "grey" in the drawing.

Examples:

| | |
|---|---|
| INTENS ( 100 ) | white. |
| INTENS ( 50 ) | grey. |
| INTENS ( 0 ) | black. |

(These examples assume a contrast range with length not equal to zero.)

On a multi-colour device, the contributions of the three primary colours (red, yellow, blue) to the total intensity, specified by *contrast* and *intensity* are defined by *colour*. *colour* is syntactically described by:

*colour*: COLOUR ( *value* , *value* , *value* ) $

The ratio between the three *values* is the ratio between the primary colour intensities; *values* may denote arbitrary real numbers.

Examples:

COLOUR ( 100 , 0 , 0 ):
red, with an intensity equal to the total intensity.

COLOUR ( 0 , 10 , 10 ):
green; Yellow and blue each have half of the total intensity.

COLOUR ( 1000 , 1000 , 1000 ):
white; Red, yellow and blue each have one third of the total intensity.

### 3.4.7. Detection

In this section it will be shown how *attributes* can be used to model the characteristics of a detection mechanism. *detection* provides external references to parts of the *picture*. It divides the *picture* in units that may be subjected to further manipulations.

REMARK

> The *detection attribute* provides the bridge between the interactive and not interactive parts of ILP. It is clear that this bridge should be designed carefully and that it affects both parts of the language. At this moment, only the not interactive part of ILP is defined. Major problems are involved in the design of this bridge if the interactive function of ILP in a computer graphics system is taken into account:
>
> -- A labelling or addressing scheme must be designed to allow selection of any part of the ILP graph structure.
>
> -- Modification operations on the ILP graph structure must be defined, which result in a compact representation of the modifications (design goal).
>
> -- A description method for moving pictures must be designed.
>
> -- The above requirements must be reconciled with the concept of i/o symmetry (design goal).

The important facility of picture manipulation must be designed with the help of ILP primitives. We want to apply ILP to structure this part of the graphics system just as it structures the basic graphics operations. In accordance with the overall functions of ILP, it is therefore required to solve these problems in such a way, that an ILP graph structure can be manipulated inside ILP itself. At present this is not the case. Some manipulation on these graphs can, however, be described in this report through the description method for the semantics of ILP, for which an (informal) metalanguage is used. More general manipulations, like for example edit operations, can be described neither inside ILP, nor in the metalanguage. One could invent another metalanguage for that purpose. It is far better however, to extend ILP with appropriate constructions to achieve i/o symmetry. The detection mechanism, only solves the first of the four problems: An addressing scheme for picture nodes is given.

Three entities are required to describe detection. A detector is an external process (which for example can involve lightpen, tracking cross or even some combination of these), that is used to select nodes in the ILP data structure. A detector has a name which is part of the environment when this detector is active. Nodes in the data structure must define by which named detectors they can be selected and for each of these, which identification string must be returned to the user if selection occurs. Thus detectors with different names can be used to search the data structure. The remaining entities are the detectant set and possibly a detectant.

Only *picture_elements* can be pointed at. Nevertheless, all nodes on the path from *root_picture* to this particular *picture_element* must be potential candidates for selection. The detectant set is a subset of these nodes, and the detectant (if defined) is a preferred element of this subset. They are formed by applying combination rules to the *detection attributes* (see below). Whenever a node is detected, the string associated with it (for the currently active detector) can be returned to the user. This provides him with a facility for identification of the various detection points. During elaboration the detectant set and detectant are constructed, and preserved in the state. Their value can be returned to the user or to the application program, when, during elaboration of a *picture_element*, this element is subjected to a selection action. Initially detector and detectant are undefined and the detectant set is empty.

The *detection attribute* has the following syntax:

```
detection: DETECT detector proper_string |
           SETDEL detector proper_string |
           UNDETECT detector $
detector:  empty |
           dname $
```

The *proper_string* is the label returned to the user when the node is detected. Each *detector* is identified by a name (*dname*). There is a common *detector* which has no name. Switching from one *detector* to another is possible by external action which consists of selecting a new name or the common *detector*.

In a short-hand notation, and disregarding *dnames* and *proper_strings*, the value of the detection *attribute* at each node can be any of the following six:

| | |
|---|---|
| AD | absolute detectable |
| RD | relative detectable |
| AS | absolute detectant set element |
| RS | relative detectant set element |
| AU | absolute undetectable |
| RU | relative undetectable |

We will call these values the principle values of the detection *attribute*. The absolute/relative value originates from the prefix that can be attached to all *attributes*. As will be seen from the following, prefixing a *detection attribute* at a node with ABS, acts as putting up a fence at that node: when pointing at some *picture_element*, all nodes beyond the fence on the path from *picture_element* to *root_picture* are undetectable for all detectors.

Detection is a type 3 *attribute*. The application universe elements are collections of pairs (DS,DT), where DS is an arbitrary detectant set and DT is either empty or an element from DS, in which case DT is the detectant. The default application universe element is the pair:

(empty detectant set, empty).

The unit application universe element is identical to the default element. The application universe element corresponding to a single *detection attribute_class* element consists of a detectant set containing only a reference to the node where the *attribute* occurred, and a detectant which is either empty or has that same reference as value.

The combination rules work as follows. The LIN list is scanned from left to right. Whenever a new *detection attribute* is encountered, the corresponding application universe element is combined with the application universe element, obtained from its predecessor in the list. The combination rule actually taken depends on the principal value of the new attribute as listed below. This process starts, using the unit application universe element.

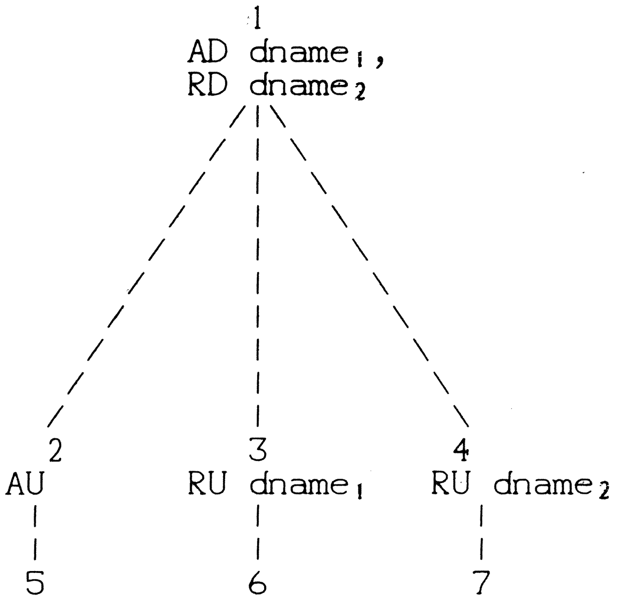
AD the node is detectable and becomes the detectant and only element of the detectant set, for the detector specified in the *attribute*. The detectant sets of all other detectors are made empty, and the detectants are made undefined.

| | |
|---|---|
| AS | the node is detectable and becomes the only element of the detectant set of the detector specified. The node is however not the detectant of that set. The detectant sets of all other detectors are made empty and the detectants are made undefined. |
| AU | the node is undetectable. The detectant sets of all detectors are made empty and the detectants are made undefined. |
| RD | the node is detectable and is made the detectant of the current state for the detector specified. It supercedes as detectant any ancestor which was specified as such before. All other information remains unchanged. |
| RS | The node is detectable and is added to the detectant set of the current state for the detector specified, without becoming the detectant. Everything else remains as it was. |
| RU | The node is undetectable for the detector specified and so are all its ancestors. Detectant sets and detectants of all other detectors remain as they were. |

It should be noted that, in accordance with the general semantics of prefixes, the use of "ABS" leads to the reinitialization of the detectant set and the detectant, whereby all previous values (even of other detectant sets) are totally disregarded. The effect of principle value "RU" may seem Relatively Unexpected, because it can modify the effect of previously encountered *detection attribute_class* elements anywhere on an element path. *detection* is the only *attribute_class* discussed so far, where "REL" can have this property. This is caused by the fact, that it is the only type 3 class, encountered in this report.

Example:

Consider the following ILP graph, in which nodes 1 through 4 are WITH...DRAW nodes and nodes 5 through 7 are *picture_element* nodes.



Node 1 can be detected by the detectors named dname1 and dname2. It is impossible to detect this node by selecting *picture_element* 5, selection of *picture_element* 6, respectively 7, only leads to detection of node 1, when the detector dname2, respectively dname1 is active.

The nodes of this graph are visited during elaboration in the order: 1, 2, 5, 3, 6, 4, 7. When detector "dname1" is active, the application universe elements contained in the state of these nodes are shown in the following application universe element table:

| node | application universe element of state | |
|---|---|---|
| | detectant set | detectant |
| 1 | empty | none |
| 2 | 1 | 1 |
| 5 | empty | none |
| 3 | 1 | 1 |
| 6 | empty | none |
| 4 | 1 | 1 |
| 7 | 1 | 1 |

If detector dname2 is active instead of dname1 this table is valid after the rows for node 7 and node 6 have been interchanged.

Example:

1
AD $dname_1$,
RD $dname_2$
|
|
2
RD $dname_1$,
RS $dname_2$
|
|
3
RS $dname_1$
|
|
4
AS $dname_1$
|
|
5

The application universe element table would be:

| node | application universe element of state | | | |
|---|---|---|---|---|
| | dname1 | | dname2 | |
| | det set | detectant | det set | detectant |
| 1 | empty | none | empty | none |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 1 2 | 2 | 1 2 | 1 |
| 4 | 1 2 3 | 2 | 1 2 | 1 |
| 5 | 4 | none | empty | none |

## REMARK

So far we have not related the pointing action to visibility aspects. Apart from detectable, each primitive can also be visible or invisible. Many hardware pointing devices (e.g. lightpen) identify detectability and visibility. We have deliberately chosen for the separate concepts, because we can give a meaningful interpretation for each combination of (in)visibility and (un)detectability. For instance, in order to change an invisible move, one must first identify it.

## 3.4.8. Coordinate mode

The *coordinate_mode attribute_class* is specified by the syntax rule:

*coordinate_mode*: FIXED !
FREE $

When the *coordinate_mode* has value FIXED, positioning information represented by a *dimensional_value* is taken to mean an absolute position. When it has value FREE, the absolute position is found, by adding the *dimensional_value* to the untransformed pen position (see 3.5.1.). *coordinate_mode* is a type 2 *attribute*, which is concatenated according to:

$$\text{REL CM}_1 <> \text{REL CM}_2 \text{ -> REL CM}_2,$$

where $CM_2$ is not the unit element. (Remember, that at the time of concatenation, all remaining *attributes* have prefix "REL"). In other words, at any time during elaboration, the part of the state, representing *coordinate_mode*, has simply the value that has last been encountered.

## 3.4.9. Control

The syntax for *control* is:

*control*: MACHINEDEPENDENTCONTROL *proper_string* $

*control* is an instrument for the specification of drawing machine dependent control information, like paper feed, clear screen and so on. In general nothing can be said about the oddities of machine typical control information. Hence only a, further unspecified, *proper_string*, is transmitted to the drawing machine.

The concatenation rule for *control* amounts to string concatenation.

## 3.4.10. Visibility

The *attribute visibility* has the syntax:

*visibility*: VISIBLE !
INVISIBLE $

When the state of a *picture* contains value INVISIBLE for the *visibility attribute_class*, this *picture* will not be drawn during elaboration. Nevertheless it will be elaborated, to update the environment properly. The current pen position (see 3.5.1.) must be updated, and the *detection attribute_class* elements must be evaluated, since invisible *pictures* may be detected.

The concatenation rule is:

$$REL\ V_1 <> REL\ V_2 \rightarrow REL\ V_2$$

where $V_1$ and $v_2$ are *visibility attribute_class* elements, and $V_2$ is not the unit element.

### 3.4.11. Attribute matches

How *attribute_matches* contribute to a state has formally been described in 3.4..

Conceptually, *attribute_match*es are a primitive form of the WITH...DRAW construction, operating on the *picture_element* level. They inhibit or permit the effect of all elements of their class that lie on the element path of the picture element. If an inhibiting match is used, these elements are replaced by the default element of the *attribute_class* (see 3.4.12.). *picture_element*s may contain two levels of *attribute_matches*. The matches of the first level are written directly following the *picture_element* tag (e.g. LINE). The matches of the second level are written directly preceding *picture_element* values like *dimensional_values*, *curve_values* etc. The first level of matches apply to all *picture_element* values unless a second level match of the same class is specified. In that case only, the *picture_element* value directly following has the second level match for that class. All *attribute_match*es not specified on any of the two levels are taken to be non inhibitive, i.e. those that leave the current state unchanged. In this way, the concept of a global state with local exceptions is also realized at the *picture_element* level.

The correspondence between *attribute_matches* and *attribute_classes* is given in the following table:

| match | class |
|---|---|
| TF | transformation |
| DT | detection |
| ST | style |
| PN | pen |
| CM | coordinate mode |
| VS | visibility |

### 3.4.12. The default attribute

With every *attribute_class* corresponds a default element, according to the following table:

| class | | default value |
|---|---|---|
| *transformation* | | unit matrix transformation |
| *detection* | | UNDETECT, i.e. undetectable |
| *control* | | MACHINEDEPENDENTCONTROL "", i.e. the empty string |
| *pen* | | PENFAULT |
| *coordinate_mode* | | FIXED, i.e. abs. positioning |
| *style* | *line_style* | PERIOD (100),i.e. solid line<br>MAP(1, RESET)<br>THICK(THICKFAULT) |
| | *typographic* | TYPFAULT |
| | *point_style* | DOT |
| *visibility* | | VISIBLE |

Apart from style values, the defaults are self explanatory. The defaults for style are as follows. Default *linestyle* is a solid line, when however the *period* is specified explicitly, default *map* is such, that the pattern is reset for every new LINE. The default for THICK is denoted by THICKFAULT, which stands for the most convenient thickness, available on the device, on which the drawing defined by the ILP program is to be drawn. Hence, THICKFAULT is device dependent. The default value for *typographic* is TYPFAULT which is discussed in 3.4.5.3.. However, *typographic* has the atoms *font*, *size*, *italic* and *bold*. When certain atoms are specified, but others not, the latter again take device dependent values. The default for *point_style* is DOT, which denotes a device dependent spot. The default value for POINTSTYLE *typographic* is the same as for ordinary *typographic*. The default POINTSTYLE *token* depends on the selected *font*, but will be a 'point' when the font contains one. The default for *pen* is PENFAULT (see 3.4.6.). If only one atom of *pen* is specified, the other again assumes a device dependent value.

## 3.5. Picture Elements

A *picture_element* is a language primitive of ILP. Each ILP-program eventually specifies a list of *picture_elements* (end nodes of the graph represented by the ILP program). A *picture_element* is syntactically described by:

```
picture_element: coordinate_type |
                 text |
                 generator |
                 NIL $
```

We will now discuss the various *picture_elements*.

### 3.5.1. Coordinate type

The syntax rules for *coordinate_type* *picture_elements* are:

```
coordinate_type: type attribute_matches
                      ( coordinates ) $
```

Such a *picture_element* consists of a *type*, *attribute_matches* and *coordinates*, in conformity with the syntax rules:

```
type:            POINT |
                 LINE |
                 CONTOUR $
coordinates:     coordinate |
                 coordinates
                      , coordinate $
coordinate:      attribute_matches
                      coordinate_value |
                 attribute_matches
                      ( coordinate_values ) $
coordinate_values: coordinate_value |
                 coordinate_values ,
                      coordinate_value $
coordinate_value: dimensional_value |
                 PP |
                 EP $
```

The *attribute_matches* in the syntax rule for *coordinate_type* are the first level matches. Those in the rule for *coordinate* are the second level matches.

PP and EP are special *coordinate_values*, defined as follows.

At any moment during elaboration, the most recently visited point in the unit user space (see 3.3.1.) is called the transformed pen position. *) The transformed pen position is stored in the environment as a *dimensional_value*. The untransformed pen position, which is also part of the environment, is defined as the result of applying to the transformed pen position, the inverse of every matrix transformation and *subspace* transformation contained in the current state. Hence, whenever during elaboration of a *picture*, the untransformed pen position is used (implicitly or explicitly), it has the *coordinate_value* compatible with the *coordinates* occurring in the *picture*.

With the help of the untransformed pen position, two special *coordinates* are defined: EP and PP. EP is mnemonic for <u>element position</u> PP for <u>picture position</u>. During elaboration of a *picture_element*, EP denotes the value of the untransformed pen position just prior to the elaboration of this element. PP denotes the value of the untransformed pen position just before the start of the elaboration of the smallest *picture* enclosing the *picture_element* in which PP is referenced.

Before the elaboration of a *root picture* starts, transformed pen position, untransformed pen position, PP and EP all have as value the unit cube origin coordinates. Each time a *subspace* selection occurs, transformed pen position is set to the value of the *subspace* origin in user unit space coordinates. Untransformed pen position, PP and EP then have the *subspace* origin in *subspace* coordinates as value. This effect is similar to the effect of coordinate transformations. For this reason *subspaces can be converted to coordinate transformations (which can not be inhibited, see 3.4.2.1.). PP allows among other things the specification of subpictures* that leave the pen position where it was at the start, by adding a *picture* like

WITH INVISIBLE DRAW POINT PP

as the last element to the *subpicture*.

Upon return from *subspace picture*, the transformed pen position remains where it is left by the *picture*. The untransformed pen position is expressed in terms of the restored coordinate system and PP is restored from the environment. EP needs not to be restored at all. It can only

---

*) The untransformed pen position (as well as the other pen positions) are abstract entities, corresponding with a point in user space, and not with a physical object like a plotter pen. This difference is particularly important when *port* transformations are used, or the *visibility attribute_class* has value INVISIBLE.

be used inside *picture_elements*. Hence, it will be copied from the most recent untransformed pen position at the beginning of that *picture_element*.

The primitive action embodied by a *coordinate_type picture_element* can be described as follows. First of all the row of *coordinates* specifies a series of positions. The positions are found in either of two ways, depending on the value of the *coordinate_mode* (see 3.4.8.):

-- In the FIXED-state, the *coordinate_values* are absolute values with respect to the current origin.

-- In the FREE-state, the *coordinate_values* are offsets from the untransformed pen position (incremental mode).

This series of positions is the same for all *types*. The *type* is used to specify a "polygon", that contains these positions as vertices. The first and last vertex of the polygon however are different for different *types*. Let the untransformed pen position before the elaboration of some *picture_element* be x. Let the series of positions be represented by $c_1$, $c_2$, ..., $c_n$. Then the polygon to be drawn is:

-- In case of *type* POINT: $c_1$-$c_2$- ... -$c_n$ .

-- In case of *type* LINE: x-$c_1$- ... -$c_n$ .

-- In case of *type* CONTOUR: $c_1$-$c_2$- ... $c_n$-$c_1$ .

The possibility x-$c_1$- ... -$c_n$-x, can be obtained by adding the special *coordinate* denoted as EP to the head of the row of *coordinates* of *type* CONTOUR. This produces a closed polygon with the original pen position as the first (and last) value, e.g.:

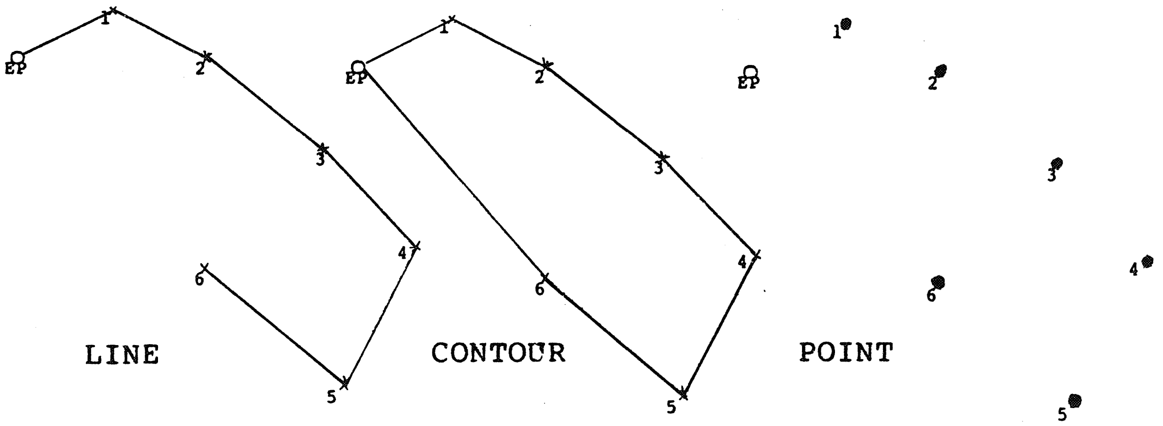
```
WITH FREE DRAW
CONTOUR CM (EP;[0, 1];[1, 0];[0, -1])
```

specifies a square that begins and ends in the untransformed pen position, valid at the start of the elaboration of this *picture*. If we negate CM in this example, we also get a closed polygon which starts and ends in the pen position. However, we cannot say what the shape will be until we know the pen position.

Next all *transformations* of the current state are applied to the *dimensional_values* of the *coordinates*. This establishes which positions the pen will visit while a POINT, LINE or CONTOUR is elaborated. What is actually drawn, and what route is actually taken, going from one position to the next, depends on the *type* and the *attributes*. The *attribute_class visibility*, and its match VS specify, whether anything will be drawn at all. In the state INVISIBLE, the route is followed as a sequence of invisible moves. In the state VISIBLE, it depends on the value of the *attribute_classes style* and *pen*, and their matches ST and PN, how the moves will actually be drawn.

There is, apart from the initial vertex, a second fundamental difference between POINTs and LINEs. For LINEs the route between successive positions defined by the *coordinates* is always a straight line, which will be drawn according to the current *style* functions. The route between POINT positions is undefined. For this reason it is impossible to apply any *line_style* function to the route between these points. It is not defined in which order the positions have to be visited, with the exception of the last one. Hence the only *style* functions for POINTs are those which specify by which symbol (centered around the "point" position), the POINT will be represented. On the other hand, it is possible to specify a *line_style* for LINEs which shows the positions as points. In that case the initial untransformed pen position is always included. With respect to *style* functions the CONTOUR behaves in a LINE-like manner.

In the next example the same row of *coordinates* is drawn as POINT, LINE, CONTOUR and EP-contour respectively. In each case the initial pen position, marked as O is the same.



### REMARK

A contour has the property that its beginning and end coincide. The coordinates of a contour are not necessarily coplanar. Is it useful to add contours which consist of coplanar points?

It might be advantageous to add a precision specification to *coordinates*. The denotation of *values* remains the same for all precisions but their implementation may be different (floating point, long and small integer etc.). In this manner *values* which lie in a given range can be stored more efficiently. Precision is at present not a part of ILP.

### 3.5.2. Text

Objects with *type* TEXT enable the production of texts as part of a *picture*. The syntax rules are:

```
text:           TEXT attribute_matches
                        ( strings ) $
strings:        string |
                strings , string $
string:         attribute_matches proper_string |
                attribute_matches
                        ( proper_strings ) $
proper_strings: proper_string |
                proper_strings , proper_string $
```

The value of *text* is a row of *strings*, which are build up from tokens. Tokens are selected from fonts. Each font contains at most 256 tokens. If the size of the character set of some device is smaller than 256, a device dependent escape mechanism is required to provide token values in the range [0-255]. Change of font is possible by means of the *typographic style attribute*. In principle an unlimited set of fonts can be used in an ILP program.

REMARK

At present the *font* selection mechanism and the escape mechanism contain some questionable choices. Some unsolved problems are:

-- Should the escape mechanism be incorporated in ILP or is it better to leave its specification to an implementor of ILP, or even to the implementor of the conversion routine from ILP to a specific device? In the latter case the assumption is made that token values from 0 to 256 can be represented symbolically by characters in an ILP program. This can be solved for binary ILP-files, but how is portability of symbolic ILP-files achieved?

-- Should ILP be extended with a facility to compose tokens and *fonts*?

An important aspect of *text* values is the way they are positioned, since nowhere in a *text* value, a *coordinate* can be specified, the position must be deduced from the current environment. No explicit page or layout *attribute* exists. *text* values are always positioned parallel with the x-axis and relative to the pen position. No limit is set to the maximal size of *text* values. Layout characters have a meaning, relative to the pen position (EP) of the current *text* value or relative to the current line of text. If *text* and other *picture_elements* are mixed, layout characters can not have a meaning, relative to previous *text* values.

The above characterization of *text* is still primitive. Extensions of the *text* mechanism can be expected.

### 3.5.3. *Generator*

So far we have encountered primitives with explicit values. The remaining three *types* are generators of values.

A *generator is syntactically described by:*

```
generator:      symbol |
                curve |
                template $
```

The semantics of a *picture_element* of type *generator* are defined as follows.

Each *generator* contains a number of *gnames*. When a *generator* is encountered by the elaboration process, this process activates some external mechanism for every *gname* of the *generator*. Each mechanism generates an ILP graph, corresponding to a *picture*, whereafter these graphs are combined in a new graph of the same type. This graph replaces the picture node corresponding to the *generator*, after which the elaboration process continues with the subgraph just inserted. The *picture* generated, however, is considered as one indivisible action. This means that manipulations can only be defined for that *picture* as a whole. In particular *detection* of parts of the elaborated *picture* is impossible.

REMARK

> If one removes the requirement of indivisible action, the *generator* no longer constitutes a *picture_element*, but a new type of *picture*. The possibility to introduce a more general *picture* construct, together with

the (orthogonal) concept of declaring any *picture* as indivisible is still under investigation.

To guarantee, that the result of the replacement is again a correct graph, two demands must be met:

-- The generated ILP graph must be complete, i.e. it may not contain references to undefined nodes. To facilitate statical checking of this property, the following rule must be obeyed. The *picture* corresponding to the graph may not contain *pnames* or *anames* of objects, defined in the *picture_program* that contains the *generator*, unless these references (*pnames* or *anames*) are passed as *template_parameters* (see 3.5.3.2.2.).

-- In the *picture* describing the generated graph, all generated *dimensional_values*, *matrix_values* etc. must have dimensions in accordance with that of the environment and eventually generated *subspaces*.

*generators* provide a library facility. Because the nature of the library elements is not defined inside ILP, they are implementation, and application dependent. Nevertheless, the interface between the library and ILP (the *generator*) is defined inside ILP, and hence does not depend on a specific implementation.

REMARK

Apart from the library aspect, *generators* form, together with the detection mechanism, the connection between the interactive part of future ILP versions and the current ILP. Because the detection mechanism (to be used, among other things for picture identification), cannot reach below *picture_element* level, a *generator* is an instrument for creating parts of pictures with highly controlled detection and hence modification possibilities. For instance, a library element that does not contain a *detection attribute*, can be considered as a "frozen" picture part.

### 3.5.3.1. Symbol

The syntax for *symbol* is:

```
symbol:        SYMBOL gnames $
gnames:        gname |
               gnames , gname $
```

Every *gname* of a *symbol* corresponds with a *root picture* in a previously defined ILP program. In this case, the picture graph is generated as follows.

Every *gname* represents a picture graph, as defined in 3.2.. If the *symbol* contains more then one *gname*, all picture graphs are combined into one, by creating a picture node, having all these graphs as direct descendants. The (left-right) order of the descendants corresponds to the textual order of the *gnames*. In this case it is necessary, that all *gnames* correspond to *root pictures* of the same dimension.

### 3.5.3.2. Curve and template

The generation mechanism activated by a *curve* or *template* can be of arbitrary nature, as long as it produces picture graphs of the correct kind. The only demand is, that the mechanism is a program that can be invoked by the elaboration process and generates an ILP picture graph accessible to it. The distinction between *curves* and *templates* lies in the structure of the picture graphs they produce.

REMARK

To facilitate programming of *curves* and especially of *templates*, a set of basic tools must be designed, that generate ILP constructions in a standardized manner.

### 3.5.3.2.1. Curve

The syntax for *curve* is:

```
curve:           CURVE type attribute_matches
                          ( curve_generators ) $
curve_generators: curve_generator |
                 curve_generators , curve_generator $
curve_generator: attribute_matches
                 curve_determinator |
                 attribute_matches
                          ( curve_determinators ) $
curve_determinators: curve_determinator |
                  curve_determinators
                          , curve_determinator $
curve_determinator: gname |
                 gname ( interval ,
                          curve_parameters ) |
                 gname ( curve_parameters ) $
interval:        UNIT |
                 ( value , value ) $
curve_parameters: curve_parameter |
                 curve_parameters
                          , curve_parameter $
curve_parameter: value |
                  dimensional_value $
```

The semantics of *curves* will be described in terms of elements from ILP programs rather then in terms of the corresponding graphs. This will lead to a clearer description. In case of a *curve* , an object of type *dimensional_values* corresponds to every *gname*. In other words, every *gname* represents a mechanism for the generation of *dimensional_values*. These *dimensional_values*, together with the *attribute_matches* of the *curve_determinator* containing the *gname* can be combined into an object of type *coordinate*. Then, using the *attribute_matches* (if present) of the *curve* a *picture_element* of type *type* can be formed out of these *coordinates*. The order of the *coordinates* in the *picture_element* corresponds to the textual order of the *gnames*. The *picture_element* thus constructed, is equivalent with the generated picture graph, that will replace the *generator* node.

The parameters of a curve can be (at most) one *interval*, and a number of *values* or *dimensional_values*. If there is an *interval*, we have a parameter curve. The *interval* is the domain of a parameter t. The *dimensional_values* of the generated *picture_element*, correspond to different values of t, when t steps through the interval. The stepsize can be calculated by the *curve* itself, can depend on a given device, or can be a parameter to the curve (a *value*). The other parameters (*curve_parameters*) are either *values* or *dimensional_values*. Their number and meaning is specific for each particular *gname* . *dimensional_values* could for instance be used, to define some fixed points on-, or tangents to the curve.

### 3.5.3.2.2. Template

The syntax for *template* is:

```
template:          TEMPLATE ( template_generators ) $
template_generators: template_generator |
                   template_generators
                        , template_generator $
template_generator: gname |
                   gname ( template_parameters ) $
template_parameters: template_parameter |
                   template_parameters
                        , template_parameter $
template_parameter: value |
                   dimensional_value |
                   pname |
                   aname |
                   dname $
```

A *template generator* may produce an ILP picture graph of arbitrary structure. Because of the fact, that this picture graph not necessarily represents a *picture_element*, the syntax rules for *template* do not contain *attribute_matches*.

Each *gname* identifying a generation mechanism has its own specific set of parameters, described by *template_generator*. *pnames* or *anames* used as parameters must correspond to *root_pictures*, *sub_pictures* or *attribute_packs* defined in the ILP program containing the *template*. These parameters specify the references corresponding to *pnames* and *anames*, allowed in the generated graph. Name conflicts must be avoided by using unique names.

The picture graphs generated, (one for every *gname*) are combined in one single picture graph, in the same way as is done for *symbols*.

## 4. Design goals and evaluation

In this chapter, the design criteria of ILP are considered and an analysis is given, to show whether and if so, how the stated goals are achieved.

### 4.1. Design goals

Five major design goals can be distinguished:

- Compactness of picture representations, to reduce the enormous amounts of data which are normally required for the representation of pictures.
- Mutual independence of attributes, to isolate the effects of individual attributes and forbid side effects caused by attributes from one class on attributes from another class.
- Symmetry of input and output, which obviates the need for separate languages for input and output descriptions.
- Embedding, which allows the incorporation of ILP in other (high level) programming languages.
- Self modification of ILP programs, which allows the description of changes in a picture in ILP itself.

Compactness of picture representations can be achieved in several ways:

- Multiple occurrences of the same subpicture are included only once in the data structure.
- Only necessary coordinate values need to be specified, i.e. in a two dimensional space two numbers are sufficient to determine a coordinate value.
- Coordinate values are packed, i.e. a priori knowledge of the range in which coordinate values lie is used to determine the most compact representation of coordinate values.

In ILP only the first two methods are used explicitly. The first is realized via the subpicture, root picture and attribute pack mechanisms. The second is realized with the subspace mechanism. Note that the dimension of each coordinate value can be determined statically. The third method can be applied by an optimizing compiler.

Apart from the influence of these explicit methods, the ILP attribute mechanism has the beneficial effect of factoring out common subpictures, since the same subpicture can be drawn in contexts with completely different attribute values.

Independence of attributes restricts the ways in which attributes can influence each other. This restriction has several advantages:

-- The semantics of individual attributes can be studied in isolation, thus obviating the need to consider complex interactions between attributes.

-- The attributes are easily extensible, since new attribute can (by definition) not influence the already existing attributes.

The restriction of attribute independence seems to be justified, if the already considerable complexity of the semantics of independent attributes is taken into account. On the other hand certain useful applications of attribute interaction are forbidden by this restriction. Line style that adapts itself to transformations is an example.

Symmetry of input and output, means that the same intermediate representation is used both for drawing and reading pictures. The advantage of this method is obvious: only one intermediate representation is required. As simple as this criterion can be formulated, so difficulty can it be achieved. Especially on the input side, a completely new set up is necessary, if input can only be provided in the form of ILP primitives like picture elements, primitive attributes or references to pictures.

Embedding means incorporation of ILP in existing programming languages. In other words, ILP can be used as a model for a graphics system, which can be incorporated in an existing programming language. Though the embedding methods may be different, the various user interfaces and the underlying model graphics system remain the same. Algol68G is an example of such an embedding, in which Algol68 serves as a host language. A major consequence of this embedding strategy is that many features (variables, loop constructions) need not be included in ILP since the host language provides such facilities.

Self modification means that, with the help of a local editor for building and changing ILP constructions, very elaborate edit operations can be described in ILP itself. Not only the resulting picture, but also the way it was constructed can be remembered, if necessary.

For both editing and modifying, a sophisticated reference mechanism is indispensable. It is felt that the attribute mechanism can be used to model such a reference scheme. The detection mechanism is the first (and sofar the only) step in this direction.

### 4.2. Omissions

Several features and concepts are not incorporated in ILP. Some are not yet understood well enough (time, modifications), others are not included in accordance with the embedding strategy. Some of the omitted features are:

-- variables, recursion, loop constructions. The host language constructions are used.

-- Subpictures with parameters, which could be used to further compress the picture data.

-- Modifications of pictures. It is not yet clear how modification operations on the ILP data structure must be described in ILP itself and how selective modifications (changing one line in a subpicture) must be realized.

-- Time and moving pictures. The problems are comparable to those for picture modifications.

-- Surfaces. The present contours can be used to delimit a surface, but better tools are needed.

-- Surface style, the equivalent of line style and point style. Greyscale forms an example.

-- Association of non graphical information with a picture.

### 4.3. Evaluation

Some lessons can be learned from the design of ILP.

The level of intermediate representation as provided by ILP seems adequate. Attention is focussed on a restricted problem area and many problems related with high level graphics languages and machine dependent issues can be (partly) ignored.

A careful description of the semantics of drawing operations and attributes reveals problems which were not recognized before. Such an analysis required a considerably greater effort, than was anitcipated.

ILP provides an uniform interface, during the design phase of a graphics system. This implies that every change in modification of ILP must be reflected in all interfaces between system modules. Note that only the interface is fully specified and that implementation techniques may differ from module to module.

## REFERENCES

[1] W.M.Newman, R.F.Sproull,
Principles of Interactive Computer Graphics,
Mc-Graw-Hill, 1973.

[2] Jose L.Encarnaçao,
Computer-Graphics,
Programmierung und Anwendung von Graphischen
Systemen, R.Oldenburg Verlag, 1975.

[3] Donald E.Knuth,
The art of computer programming,
Vol 1/Fundamental algorithms,
Addison-Wesley, 1968, pg 305-357.

[4] P.J.W. ten Hagen, P.Klint,
H.Noot & T.Hagen,
Design of an interactive graphics system,
Report IW 36/75,
Mathematical Centre, Amsterdam, 1975.

## Appendix 1 Syntax

The syntax rules are given in BNF. Non terminals are denoted in the form *non_terminal*. The syntax is context free. The non terminal that is defined in a rule is separated by a colon (:). Alternatives are separated by a bar (|). The end of a rule is marked with the symbol $. Terminal symbols are either special single characters from the following list:

( ) { } , . ;

or they are delimeters denoted in bold capitals e.g. **TERMINAL**. The non terminals not defined in this syntax are all defined in Appendix 2. They constitute the so called lexical units.

The syntax as presented is directly fed into the parser generator for ILP. For this reason usual notational conventions to make the syntax look more compact, have been omitted.

*picture_program*: *pictstruct* |
*picture_program* *pictstruct* $

*pictstruct*: *named_picture* |
*attribute_pack* $

*named_picture*: *root_picture* |
*subpicture* $

*root_picture*: **PICT** *dimension* *pname*
*picture* . $

*dimension*: **DIMLESS** |
*dim* $

*dim*: ( *value* ) |
*empty* $

*subpicture*: **SUBPICT** *dimension* *pname*
*picture* . $

*attribute_pack*: **ATTR** *dimension* *aname*
*attribute* . $

```
picture:  pname |
          picture_element |
          { pictures } |
          subspace picture |
          WITH attribute
               DRAW picture $

pictures: picture |
          pictures ; picture $

picture_element: coordinate_type |
          text |
          generator |
          NIL $

coordinate_type: type attribute_matches
               ( coordinates ) $

coordinates:   coordinate |
          coordinates
               , coordinate $

coordinate:    attribute_matches
               coordinate_value |
          attribute_matches
               ( coordinate_values ) $

coordinate_values: coordinate_value |
          coordinate_values ,
               coordinate_value $

coordinate_value: dimensional_value |
          PP |
          EP $

dimensional_value: [ values ] $

dimensional_values: dimensional_value |
          dimensional_values ,
                dimensional_value $

matrix_value:  [ dimensional_values ] $

values:   value |
          values , value $

type:          POINT |
          LINE |
          CONTOUR $

subspace: SUBSPACE dim new_axes $
```

```
new_axes: position ( shift axes ) $

shift:          dimensional_value $

position: CURRENT |
          ORIGIN $

axes:           empty |
          , dimensional_values $

generator:      symbol |
          curve |
          template $

symbol:   SYMBOL gnames $

gnames:   gname |
          gnames , gname $

curve:          CURVE type attribute_matches
                ( curve_generators ) $

curve_generators: curve_generator |
          curve_generators , curve_generator $

curve_generator: attribute_matches
          curve_determinator |
          attribute_matches
                ( curve_determinators ) $

curve_determinators: curve_determinator |
                 curve_determinators
                , curve_determinator $

curve_determinator: gname |
          gname ( interval ,
                curve_parameters ) |
          gname ( curve_parameters ) $

interval: UNIT |
          ( value , value ) $

curve_parameters: curve_parameter |
          curve_parameters
                , curve_parameter $

curve_parameter: value |
                 dimensional_value $

template: TEMPLATE ( template_generators ) $

template_generators: template_generator |
          template_generators
                , template_generator $
```

```
template_generator: gname |
          gname ( template_parameters ) $

template_parameters: template_parameter |
          template_parameters
               , template_parameter $

template_parameter: value |
          dimensional_value |
          pname |
          aname |
          dname $

text:          TEXT attribute_matches
               ( strings ) $

strings:  string |
          strings , string $

string:   attribute_matches proper_string |
          attribute_matches
               ( proper_strings ) $

proper_strings: proper_string |
          proper_strings , proper_string $

attribute_matches: empty |
          attribute_matches
               deny attribute_match $

deny:     empty |
          ~ |
          NOT $

attribute_match: TF
          DT |
          ST |
          PN |
          CM |
          VS $

attribute:     ABS basic_attribute |
          REL basic_attribute |
          basic_attribute $

basic_attribute: attribute_class |
          aname |
          { attributes } |
          NIL $

attributes:    attribute |
          attributes ; attribute $
```

```
attribute_class: transformation |
          detection |
          style |
          control |
          pen |
          coordinate_mode |
          visibility $

transformation:    rotate |
          scale |
          translate |
          matrix |
          projection |
          affine |
          homogeneous_matrix |
          port $

rotate:   ROTATE value
               AROUND invariant $

invariant:     ( dimensional_values ) $

scale:         SCALE dimensional_value $

translate:     TRANSLATE dimensional_value $

matrix:   MATRIX matrix_value $

affine:   AFFINE matrix_value
               dimensional_value $

projection:    PROJECT eye_position
               ON projection_space $

projection_space: dimensional_value |
          ORIGIN dimensional_value $

eye_position:  dimensional_value |
          PARALLEL dimensional_value $

homogeneous_matrix:
          HOMMATRIX homogeneous_matrix_value $

homogeneous_matrix_value:
          [ homogeneous_dimensional_values ] $

homogeneous_dimensional_values:
          homogeneous_dimensional_value |
          homogeneous_dimensional_values ,
               homogeneous_dimensional_value $

homogeneous_dimensional_value:
          [ values ] $
```

```
port:               window |
            window , viewport $

window:     WINDOW ( dimensional_value ,
                 dimensional_value ) $

viewport: VIEWPORT ( dimensional_value ,
                 dimensional_value ) $

style:      line_style |
            point_style |
            typographic $

line_style:     PERIOD ( period_description ) |
            MAP ( value reset ) |
            THICK ( value ) $

pen:        PENFAULT |
            contrast |
            intens |
            colour $

period_description: dash |
            dash , gap |
            dash , gap , dash $

dash:            DOT |
            value $

gap:        value $

reset:      RESETCOORDINATE |
            CONTINUE |
            RESETLINE $

contrast: CONTRAST ( value , value ) $

intens:     INTENS ( value ) $

colour:     COLOUR ( value , value , value ) $

typographic: TYPFAULT |
            font |
            size |
            italic |
            bold $

font:       FONT ( value ) $

size:       SIZE ( value ) $

italic:     ITALIC ( value ) $
```

```
bold:        BOLD ( value ) $

point_style: DOT |
             POINTSTYLE typographic |
             POINTSTYLE marker $

control:     MACHINEDEPENDENTCONTROL proper_string $

coordinate_mode: FIXED |
             FREE $

visibility: VISIBLE |
             INVISIBLE $

detection: DETECT detector proper_string |
             SETDEL detector proper_string |
             UNDETECT detector $

detector: empty |
             dname $

empty:             $
```

## Appendix 2 Lexical units

```
value:       unsigned_value |
             + unsigned_value |
             - unsigned_value $

unsigned_value: unsigned_integer |
             decimal_fraction |
             unsigned_integer exponent_part |
             decimal_fraction exponent_part $

decimal_fraction: unsigned_integer . unsigned_integer $

exponent_part: e + unsigned_integer |
             e - unsigned_integer $

unsigned_integer: digit |
             unsigned_integer digit $

aname:            name $

pname:            name $

gname:            name $

dname:            name $

name:        letter |
             name letter |
             name digit $

proper_string: " any_sequence_of_symbols_not_containing_" " $

letter:      a | b | c | d | e | f | g |
             h | i | j | k | l | m | n |
             o | p | q | r | s | t | u |
             v | w | x | y | z |
             A | B | C | D | E | F | G |
             H | I | J | K | L | M | N |
             O | P | Q | R | S | T | U |
             V | W | X | Y | Z $

digit:       1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 $

marker:      " any_symbol,_except_" " $
```

# INDEX