INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR

THE INTERMEDIATE LANGUAGE FOR PICTURES

TEUS HAGEN, PAUL J. W. TEN HAGEN, PAUL KLINT and HAN NOOT
Mathematical Centre
2de Boerhaavestraat 49, Amsterdam, the Netherlands

The Intermediate Language for Pictures (ILP) determines the structure of a graphics system in which pictures are represented as ILP programs. ILP defines extensions to general purpose programming languages. Input and output is defined as generating and interpreting ILP programs. ILP ensures that a uniform concept is used during design. System modules can be applied to symbolic ILP programs for testing. Applications can be defined in terms of extensions to the, indeed easily extendible, ILP. New language constructs present in ILP are emphasized.

1. INTRODUCTION

The Intermediate Language for Pictures (ILP), has been designed as part of a research project on computer graphics. The graphics system [1] being constructed will be the kernel of a satellite system for the manipulation of structured data.

ILP fits into the structure of the system in the following ways:

- All pictures are represented as ILP programs. ILP programs must therefore be stored, retrieved, classified and communicated.
- A high level graphical language is obtained by embedding ILP in an existing high level general purpose programming language.
- The various drawing devices are logically connected by defining a conversion between ILP and device code. This is also true for input devices. As an important consequence, full symmetry between input and output is obtained.

These applications require that ILP is a data structure rather than a programming language. However, one may equally well consider these data structures as programs, which, when executed (interpreted), produce the picture as output.

For the designers of the system ILP plays a key role both as a design method and as a means of communication. ILP is treated as a programming language in the sense that a complete definition of syntax and semantics for it is given. It can be represented in symbolic form just as any other programming language. A compiler and interpreter for it are being developed. All these activities are at the same time contributions to the graphics system. This is illustrated with a few examples:

- There is a one to one correspondence between the syntax and the data structure representation (of a picture). This syntax therefore defines both the form of the internal representation and the extensions to high level languages.
- The semantics of the language define the machine independent part of pictures and the effect of ILP on an abstract drawing machine. Each physical device is connected by defining a correspondence to this abstract machine.
- System modules can be tested separately by trying them on symbolic ILP programs. In this way, cooperation with other modules can be simulated and test results are available in (human) readable form.

The most important achievement, however, is that a conceptual uniformity is maintained throughout the system.

During the definition of the language, the designers found that ILP is a means to isolate and characterize the essentials of computer graphics. It is also possible to compare the complexity of various operations on graphical data by expressing these operations as transformations on ILP programs.

2. THE BASIC STRUCTURE OF ILP

ILP is described in [2]. Only a simplified version is presented here.

Interpreting data can cause the interpreter to perform two kinds of actions, namely, external actions and changes of state of the interpreter. We therefore choose data entities that correspond with these two kinds of actions, external action specifiers or "actions", and state specifiers which are called "attributes". Furthermore, one needs an operator to connect attributes with actions, expressing the fact that the actions should be carried out in the state as described by the attributes.

The basic construction for the connection operator has the form:

WITH A DRAW P ,

where: "A" denotes a collection of attributes, "P" denotes a set of actions, called a picture, and "WITH...DRAW..." denotes the connection operator. The construction as a whole is again a picture.

For both pictures and attributes there exist primitives and complex constructions built by composition. ILP is a low level language in the sense that, apart from a number of language primitives, only a few very elementary means of composition are provided. They serve two main purposes:

- Data must be represented in a compact way. To this end a subroutine like construction exists both for pictures and attributes. Common state information need be specified only once, and the WITH...DRAW construction, which can be nested, makes it possible to create locally, partial exceptions to the current state.

- Data must be structured to suit the anticipated manipulations. The possibility to insert empty cells in both picture lists and attribute lists allows the skeleton for a data structure to be specified, in which the actual values can be supplied later. The composition rules simple as they are can be used to build arbitrary directed graph structures.

The two basic entities, "picture" and "attribute", have the following syntax:

```
<picture>:
        <picture element> | <pname> |
        '{'<pictures>'}' |
        <subspace> |
        WITH <attribute> DRAW <picture> ;

<attribute>:
        [ABS | REL] <basic attribute> ;

<basic attribute>:
        <attribute class> |
        <aname> | '{' <attributes> '}' ;
```

The non composite constructions are "picture element" and "attribute class". They will be discussed later. "pname" and "aname" are names of pictures and attributes, respectively. An ILP program consists of a set of named pictures and a set of named attributes. A named picture is called a root picture if its name is external (global), it is called a subpicture otherwise. Interpretation of an ILP program is started in a root picture. We shall use the word "subpicture" to include "root picture" as a special case. A named attribute is called an attribute pack.

```
<root picture>:
        PICT <dimension> <pname>
                <picture> ;

<subpicture>:
        SUBPICT <dimension> <pname>
                <picture> ;

<attribute pack>:
        ATTR <dimension> <aname>
                <attribute> ;
```

The subroutine and bracket mechanism can specify (directed) attribute graphs and picture graphs. The WITH...DRAW operator combines picture and attribute graphs into one picture graph. The concept of a picture graph will be used to give the basic semantic rules. The initial node of the graph is a root picture. The direct descendants of that node are the pictures that constitute the body of the root picture. Picture elements are terminal nodes. The non terminal nodes are the other alternatives of the syntax rule for pictures. The WITH...DRAW nodes always have two descendants namely an attribute and a picture. An attribute node only contains attributes as descendants.

At the present time recursive calls are not allowed, because neither pictures nor attributes contain any form of conditions. This and other constructions, like assignment and parameters, have been left out of the language for two reasons:

- ILP programs as such, will usually be produced by programs rather then by human beings; programming convenience is therefore less important.

- ILP will be embedded in high level languages where all these constructions are already present.

The interpreter is traversing the nodes of the graph (or more precisely: the tree obtained by expanding this acyclic directed graph) in preorder [3]. Each time an attribute node is encountered, the attribute (which may be an entire subgraph) is interpreted, resulting in a so-called "state component". This new component is mixed with the current state into a new current state. In principle, effects of these state components are accumulated. The picture node of the same parent node is then interpreted in this new state. Upon return to the parent node the original state is restored. Each time a picture element is encountered the element at hand is transformed according to the current state. The resulting element is next converted into a sequence of machine dependent actions.

According to this scheme, further semantics specify the following items:

- How attributes are converted into states and state components.

- How two states are mixed.

- How a state is converted into drawing state descriptions.

- How attributes transform picture elements.

- How picture elements will be converted into drawing machine instructions.

The attributes of child nodes have priority over those of parent nodes; e.g., they are applied first and moreover, they specify whether the parent attributes will be applied at all. The latter is controlled by the tags "ABS" and "REL" respectively, which may be prefixed to a list of attributes. This mechanism implements the concept of specifying a common state with local exceptions (ABS) or adjustments (REL). ABS indicates that the new attribute replaces those of the parent. REL indicates composition of attributes.

The set up so far can be used for a large family of special purpose languages, of which the graphical languages form only a small part.

There are many applications for this attribute concept. For example: find the graph that is equivalent to a given graph but optimal with respect to compactness (number of attribute nodes) or execution speed (number of state transitions). Each subpicture can be executed in its own private state regardless of the state of the caller and hence always cause the same effect.

The flexibility is also proven by the wide variety of attributes that fit into this scheme. In fact, it is especially designed to allow extensions by adding new attribute classes in order to realize a particular application.

3. THE INTERPRETATION OF ATTRIBUTES AND PICTURES

We now specify the remaining semantic items.

The syntax rule for non composite attributes is:

```
<attribute class>:
        <coordinate mode> |
        <transformation> |
        <detection> | <style> |
        <control> | <pen> |
        <visibility> ;
```

Attributes are divided into "attribute classes". In the process of mixing attributes, only attributes from the same class are involved. The result of mixing attribute primitives from a single class is called an "attribute class value" (or class value for short). Each primitive attribute itself is a particular instance of a class value. The converse however, is not true, i.e. not every class value can be expressed by means of one attribute primitive of that class.

A state component is a list of class values which contains at most one class value for each attribute class.

The attribute graph that has to become a state component is elaborated as follows. First all references (anames) to attribute packs are replaced by the corresponding attribute pack. The combining operation starts bottom upwards. Each attribute list that contains no sublists between brackets is converted into a state component:

- The primitive attributes are sorted by class without disturbing the suborder in each class, e.g.:
  $\{a_1;a_3;b_1;c;b_2;a_2\} \Rightarrow \{a_1;a_3;a_2;b_1;b_2;c\}$.
- Next the attributes of one class are combined (concatenated) into one class value, e.g.:
  $\{(a_1{*}a_3{*}a_2);\ (b_1{*}b_2);\ (c)\}$.

According to the syntax, each primitive attribute can have at most one ABS/REL prefix. These prefixes are applied as follows:

$$A * REL\ a * B = A * a * B .$$
$$A * ABS\ a * B = a * B .$$

Here "A" and "B" denote a sequence of attribute primitives of the same class and "*" denotes the mixing operator. The resulting value is further treated as a class value. The combination process is repeated, taking class values instead of primitive attribute values. Parentheses are removed by distributing the prefix (if any) over the individual class values.

By repeatedly applying the combination rule for state components, each time going up one level, a single state is finally obtained.

Combining a state component belonging to a WITH...DRAW construction with the current state is a special case of the mixing process defined above.

The application of a state to a picture element takes two major steps. First the picture element performs a state selection. Next the state finally obtained is effectuated.

The general form of a picture element is:

"type" <attribute matches> "type values"

This is the primitive form of the general construction

WITH A DRAW P ,

preceded by some type specification. The attribute matches select a state component from the current state as follows: A single attribute match is a boolean value. Each attribute class has a corresponding attribute match. For each class the corresponding match value specifies whether this class participates in the state component or not. The state component is completed to a full state by adding default values for each missing class. The default values may depend upon the type of the picture element. This mechanism is the ultimate consequence of providing a common state with individual exceptions.

The five attribute matches and their corresponding attribute classes are:

match	class	comment
CM/~CM	<coordinate mode>	Free or fixed coordinate mode.
TF/~TF	<transformation>	
DT/~DT	<detection>	Selection by pointing or not.
ST/~ST	<style>	
PN/~PN	<pen>	Pen functions (colour etc.).
CM/~CM	<coordinate mode>	Free or fixed coordinate mode.
VS/~VS	<visibility>	The picture element is visible or not.

The effect of individual class values on picture elements can be described in terms of control information for the abstract drawing machine or as a transformation of an ILP program, e.g.:

WITH a DRAW p_0 <=> $\{p_1; \ldots ; p_n\}$

Where $p_0, \ldots, p_n$ are picture elements.

The effect of picture elements themselves on the abstract machine is defined in a way similar to attribute classes. The meaning of some picture elements is defined in terms of other (more) primitive elements but picture elements are never

composite values.

4. PICTURE ELEMENTS

Picture elements are syntactically described by:

```
<picture element>:
        <coordinate type> |
        <curve> | <text> |
        <library> | NIL ;
```

We will now discuss the various picture elements.

4.1 Coordinate type

The type-tags for which the type-values must be coordinates are:

```
<type>:
        POINT | LINE | CONTOUR ;
```

They occur in the following syntax rules:

```
<coordinate type>:
        <type> <attribute matches>
                '{' <coordinates> '}' ;

<coordinate>:
        PP | EP |
        <attribute matches>
                <dimensional value> ;
```

The primitive action embodied by a coordinate type can be described as follows. The row of coordinates specifies a series of positions. The coordinates are relative to the current origin in the coordinate mode FIXED and relative to the current pen position in the coordinate mode FREE. A type-tag may add a first and last position as follows. POINT adds nothing. LINE adds the pen position at the beginning. CONTOUR adds the first position at the end. PP and EP may be used anywhere in the sequence. They indicate the pen position at the beginning of the picture element (EP: element position) or at the beginning of the enclosing picture (PP: picture position). Note that EP as the first coordinate of a CONTOUR results in a contour that starts and ends in the initial pen position.

What is actually drawn going from one position to the next, depends on the type-tag and the attributes. The pen functions that define the colour, line width, etc. are applied only in the state PN. In a similar way, the attribute match ST specifies whether the current style functions or the default style function for that type will be applied.

Many pictures are most naturally described in a space of a certain dimension. The ILP "subspace" mechanism makes it possible to temporarily change the dimension of the space in which a picture is being constructed. If a picture must lie in a given plane, the plane can be chosen as subspace and as a result, all redundant coordinates in the picture specification must be omitted. Syntactically a subspace is specified as follows:

```
<subspace>:
        SUBSPACE <dimension>
                <new axes> ;
```

The dimension of the subspace is specified by "dimension" while the subspace itself is specified by "new axes", which consists of dim+1 vectors. The first vector specifies the origin, the remaining vectors specify the orientation of the axes and the coordinate units of the subspace.

4.2 Text

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

```
<text>:
        TEXT <attribute matches>
                '{' <strings> '}' ;

<string>:
        <attribute matches>
                <proper string> ;
```

The type-value of TEXT is a row of strings. Each element in the row may have its own private escape characters.

Characters are grouped in alphabets of at most 256 tokens. We assume that there are at least 64 printable characters in the system. With the help of 2 escape characters it is possible to specify all 256 tokens. A change of alphabet is possible by means of attributes. In principle an unlimited set of alphabets can be used in an ILP program.

Whether an attribute from a given class applies to a TEXT element or not, depends on the definition of the attribute itself. Attributes which are applied exclusively to TEXT elements are called "typographic functions" and are a subclass of the style functions.

4.3 Curve

A CURVE value consists of a row of curve descriptions. Each curve description generates a row of coordinates. The effect of a CURVE value can now be defined as the effect of a LINE with the same attribute matches and with the generated coordinates as type values.

The functions are devided in two types: parameter and non parameter functions. For a parameter function one must specify an interval and two or three functions of one variable. The coordinates produced are of type (x(t), y(t) [, z(t)]), where t steps through the interval. The step size can be calculated by the function itself, can depend on a given device or be given as one of the arguments.

Non parameter functions or system functions are collected in a system function library. Each function has its own name and parameter format. The functions only produce coordinates and have no side effects whatsoever on the drawing device or environment. The parameters are handed over to the system routine without any modification by current attributes.

4.4 Library

LIBRARY is followed by a row of names of external subpictures. The effect of a LIBRARY call is that a sequence of primitives is produced. The way these primitives are produced inside the LIBRARY

function is not specified in ILP.

Inclusion of a part of a picture program in a LIBRARY gives it the predicate "symbol". For all possible operations on picture programs, symbols are indivisible primitive units.

5. BASIC ATTRIBUTES

The order in which attribute classes are listed in the syntax rule (section 3) is also the order in which they are applied to picture elements.

We will now briefly discuss examples of attributes for the classes transformation, style and detection. These examples illustrate that a large variety of unrelated attributes fits into the same scheme.

5.1 Transformations

The effect of transformations is that of coordinate transformations well known in computer graphics.

```
<transformation>:
        <rotate> | <scale> |
        <translate> | <matrix> |
        <projection> | <affine> |
        <homogeneous matrix> |
        <port> ;

<port>:
        <window> |
        <window> ',' <view port> ;
```

Homogeneous matrix is the general transformation, written as a (n+1,n+1)-matrix value (n is the dimension). One may also construct such a homogeneous matrix value with the help of sub-matrices like rotation, scaling and projection.

"Port" has two aspects. As "window", it performs clipping along the border of the window. Mixing of windows means intersecting them. In combination with "view port" it defines a (matrix) transformation to screen coordinates. Clipping is performed first.

At present, only rectangular windows are allowed. The combination of two (matrix, window) pairs is, in general, not possible (rotation). In the absence of rotation, combination can be described as:

$$(M_1,W_1) * (M_2,W_2) = (M_1*M_2, W_1*M_2(W_2)).$$

Here M(W) means the rectangle transformed by M. M_1*M_2 implies matrix multiplication. W_1*W_2 implies intersection of windows. If there is rotation one must retain both pairs. Thus, in general, a transformation class value consists of a sequence of (matrix, window) pairs.

5.2 Style

Style functions describe what kind of lines and characters (and in a future extension of the language what kind of shades and grey scales) are to be produced by the drawing machine. Considering the enormous variety of styles that can be produced by drawing machines, the style function package has to be extendible.

The three classes of style functions that exist so far, e.g., line styles, point styles and typographic styles are mutually unrelated. Line styles are applied to coordinate values with type-tag LINE or CONTOUR, point styles to those with type-tag POINT and typographic styles to TEXT. A class value for styles contains a line style, point style and typographic style. Combination means replacement of the corresponding functions (REL) or replacement of all functions (ABS). In the latter case not specified styles enforce default values.

We will discuss line styles only.

```
<line style>:
        PERIOD
            '(' <period description> ')'
        MAP '(' <value> <reset> ')' |
        THICK '(' <value> ')' ;
```

The attribute can produce a large variety of dotted and dashed lines. Period is a basic pattern which is repeatedly produced going along the line.

```
<period description>:
        <dash> | <dash> ',' <gap> |
        <dash> ',' <gap> ',' <dash> ;

<dash>:
        DOT | <value> ;

<gap>:
        <value> ;

<reset>:
        RESETLINE | CONTINUE |
        RESETCOORDINATE ;
```

The period is defined on a straight line piece of 100 units in length: Hence $dash_1 + gap_1 + dash_2 + gap_2 = 100$. Gap_1 through gap_2 may be omitted, implying that the first missing one adds up to 100. If dash has value DOT, a point is produced on the spot which has a length of 0 units with respect to the period.
Examples:
PERIOD (100) => solid line (e.g. the default style for LINE).
PERIOD (25, 50) => dashed line with gaps equal to dashes. It starts, however, with a half dash.

Map defines the length of the period in coordinate distance units. Reset is a value telling whether a period has to be restarted or continued when a new coordinate value or a line is encountered.

5.3 Detection

The detection attribute provides the primitives for interactive work with pictures. Basically its effect is that it isolates parts of the picture

For each node in the graph (which is not an end node) the detection attribute tells whether that node can be detected by an external selection process (detector) or not.

The detection attribute has the following syntax:

```
<detection>:
        DETECT <dname> <proper string> |
        SETEL <dname> <proper string> |
        UNDETECT <dname> ;
```

A detector has its own name (dname). There is also a common detector which has no name. Switching from one detector to another is possible by external action. Whenever a node is detected, the string (if any) that is attached to it can be returned to the user for identification.

Combining detection attributes means selection of a set of nodes, called the "detectant set", and selection of a preferred node, called the "detectant". Detailed manipulation on the detectant set and the detectant is possible with DETECT, SETEL, UNDETECT and the prefixes ABS or REL. If present, the prefix ABS makes the set empty and the detectant nil. REL leaves them as they are. DETECT adds the node to the specified set and makes the node detectant. SETEL only adds the node to the specified set. With the prefix REL, UNDETECT makes the node and all its ancestors undetectable for the detector specified.

The attribute match DT switches on the detection mechanism as a whole. A single primitive can be isolated by selecting a detector not present in its parent node.

The detection mechanism can differentiate between each incarnation of a subpicture. It also can detect the subpicture itself, i.e., all incarnations.

6. CONCLUSION

We attempted to show that ILP provides a general but simple scheme in which a large variety of language constructions can be fitted. It must be admitted that not everything we wanted to include in the language can be modelled this way.

The scaling (by transformations) of line styled picture elements produces the same line style pattern although the line itself changes. To remove this restriction interaction between attribute classes (style and transformation) would be required.

The attribute mechanism allows non pictorial information to be associated with pictures.

Our main goal now is to gain experience with ILP as quickly as possible by applying it in the way mentioned in the introduction.

REFERENCES

[1] P.J.W. ten Hagen, P. Klint, H. Noot and T. Hagen, Design of an interactive graphics system, MC Report IW36/1975, Mathematical Centre, Amsterdam, 1975.

[2] T. Hagen, P.J.W. ten Hagen, P. Klint and H. Noot, ILP, Intermediate Language for Pictures, MC Report IW68/1977, Mathematical Centre, Amsterdam, 1977.

[3] Donald E. Knuth, The art of computer programming, Vol 1/ Fundamental algorithms, Addison-Wesley, Reading, Massachusetts, 1968, 305-347.