stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IN 14/77 AUGUSTUS

T. HAGEN

INTERMEDIATE DATA STRUCTURE (IDS)

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 Reviews-categories: 8.2

Intermediate Data Structure (IDS)

by

T. Hagen

ABSTRACT

A survey is given of a data structure representation (called IDS) of ILP (Intermediate Language for Pictures). Examples of this representation are given together with algorithms for the interpretation and manipulation of these complex data structures. An implementation of ILP in terms of a virtual machine with a data structure like IDS is outlined.

KEY WORDS & PHRASES: *Computer graphics, graphical data structure, picture representation.*

CONTENTS

1. GOALS AND CRITERIA

This report is closely related to the report "ILP, Intermediate Language for Pictures" [1]. ILP concepts and definitions are not repeated here. To facilitate referencing, the same notation as in [1] is used.

A first goal is to define IDS (Intermediate Data Structure for ILP) and to propose an implementation. The intention of this proposal is to provide a classification of the problems as presented by some ILP design decisions. ILP concepts which have an important influence on the semantics are emphasized. A distinction is made between the logical units of IDS and the actual "hardware" representation. In this way a detailed and directed discussion of the implementation is made possible.

The main goal in the design of IDS is to provide a one to one correspondence between ILP and IDS. The special properties of ILP that must be preserved are:

- The compactness of picture representation.
  Multiple occurrences of parts of a picture (subpicture) are included only once in the data structure.

- The subspace mechanism.
  This mechanism allows omission of irrelevant value, e.g. in a subspace with a smaller dimension then the surrounding space, one (or more) coordinates can be omitted.

- ILP provides a library facility and predefined (curve) generators.
  IDS must contain sufficient information to support these facilities.

Modification or edit operations on ILP programs must be possible. IDS should be flexible enough to allow such modifications.

IDS will be used as input for picture processing and picture manipulation programs. These programs will bypass a lot of the detailed information of ILP. Traversing of the data structure is facilitated by providing a uniform and simple main data structure. This overall structure gives little, but sufficient information for traversing the data structure and provides the descriptors necessary for storage allocation.

2. INTRODUCTION TO THE DATA STRUCTURE

First we define the basic elements of IDS. The most elementary unit of information is called an elementary data item. One or more elementary data items that belong together logically are called a data block. One or more data blocks may be collected in a record. All records of the IDS for one ILP program are collected in a picture file. An actual implementation requires that a correspondence between elementary data items, records, files and the underlying hardware representation is defined.

Elementary data items are the smallest information units that are distinguished. An address is associated with every elementary data item. Records are identified by the address of their first elementary data item. An address of an IDS record is called a pointer. The first elementary data item in the first data block of a record always is a pointer, called link. The second elementary data item in the first data block is called the type of a record. A list of data blocks with a fixed internal structure as depending on the type is the data body of a record. The third elementary data item in the first data block is the length of the data body. The first data block of a record i.e. link, type and length is called the header of a record. All records have this structure:

- header
  -- link
  -- type
  -- length
- data body
  -- data block(s)

In the sequel we will reserve the name data block for data blocks in the data body only. The explicit names link, type or length will be used to refer to components of the record header. A record is self descriptive in the sense that its data blocks can be interpreted completely using the information in the record header. In the case where data blocks of one (record) type may vary in length, the length of such a block is specified as a part of the data block.

So far we have mainly stressed the layout of IDS records, and not considered their contents. The information contained in an ILP construction can be build up from a number of elementary ILP data items. The IDS must provide a means to store these elementary data items together with their relations. The definition of ILP is such that elementary ILP constructions are well separated from relations between those constructions.

3. IDS-ILP STRUCTURE

An ILP program is equivalent with a directed graph without cycles. All arcs of such a graph are represented in IDS by means of links. The direct descendants of a given node in the graph are collected in a linked list of records (see figure 3.1). The link in the header of the record is used for that purpose.

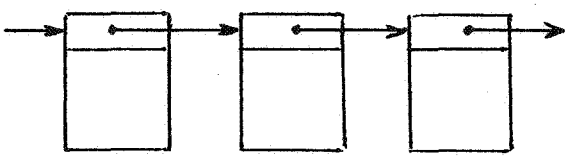


figure 3.1

Each linked list of this nature is prefixed with a so called link back record. The link of this record points to the first record of the list. The link of the last record of the list points to the link back record. The link back record actually turns the list into a ring. Moreover, the data body of a link back record contains a pointer, called back link, to the parent node (see figure 3.2).

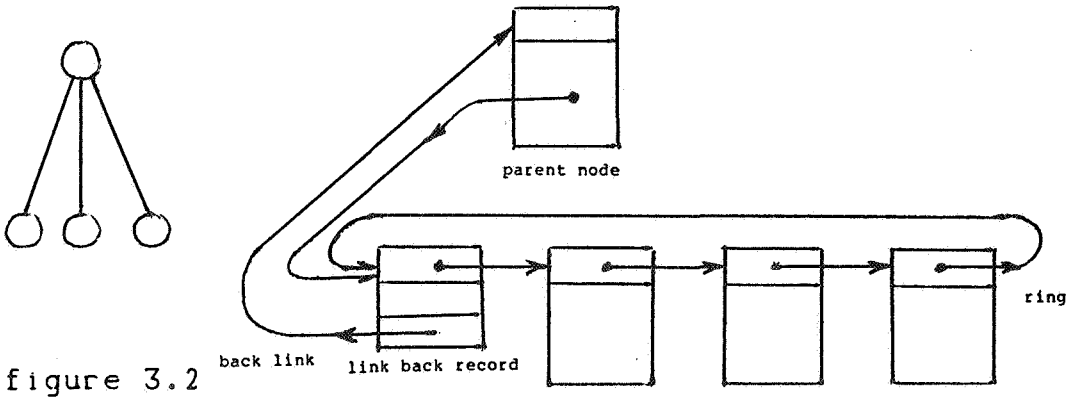


figure 3.2

Direct descendants of an ILP node are either of type *picture* or of type *attribute*. In IDS the direct descendants are linked in two separate type of rings called *picture* ring and *attribute* ring respectively. The record, which corresponds to the parent node, has a pointer to link back record of the ring of its direct descendants.

The structure of ILP is such that a distinction can be made between elementary constructs (the end nodes of the graphs) that do not contain arcs, elements that contain only arcs, and elements that contain external references and (not necessarily) other data. To reflect this structure, IDS contains data records, pointer records and mixed records. End nodes that specify an external reference, provide infor-

mation for other end nodes, which refer to that external reference. A *generator* requires, for instance, besides a dimension and a name, also a description of parameters. This global information, is collected in a mixed record, called the name record. The information in the end node that is unique (in case of a *generator*: the actual parameter values and a pointer to the corresponding name record) is found in another mixed record, called reference name record. A similar separation of common and unique information is formed in the case of *subpictures* and *attribute packs*. Here also name and reference name records are used. Data records always correspond to end nodes in the ILP graph. Nodes which are not end nodes are expressed in pointer records. We will now discuss the various types of records in more detail.

3.1. Pointer records

A pointer record always connects rings. The data body consists of data blocks of one pointer. An example of a pointer record is the link back record. All other pointer records correspond to ILP constructions. These are:

- Constructions that associate data with a *picture*. The first pointer points to the associate ring, the other pointer to a *picture* ring. The following association constructions exist:

  -- Withdraw record
  associates an *attribute* ring with a *picture* ring.

  -- Subspace record
  associates a subspace transformation with a *picture* ring. The transformation is placed in a ring, called *subspace* ring.

  Remark
  To associate non pictorial information with a *picture* a construction must be available to associate this information with a *picture* ring. This associate record should belong to the group of pointer records. However, there is no ILP construction for such an association of data with a *picture*.

  Withdraw and subspace records always belong to a *picture* ring (*picture* node).

- Bracket records.
  The data body consist of one pointer to a ring (descendant nodes) of the same type as the ring of which this bracket record is an element (node).

3.2. Mixed records

A construction like *attribute* (*attribute pack*) and *picture* (*subpicture* or *root picture*) has a name and a dimension. For multiple referenced objects common information can be collected in a name record. ILP constructions which refer to these objects have a corresponding record in IDS, called reference name records.

3.2.1. Name records

The name records which correspond to the name of an ILP construction are:

- root picture record or rpname record.
- subpicture record or spname record.
- attribute pack record of aname record.

The data body of these name records consist of one data block containing:

- a pointer to the *picture* or *attribute* ring.
- the dimension of the *picture* or *attribute*.
- the number of references to this object.
- the name, which consists of the actual length of the name, followed by the characters of the name.

Note that the name record corresponds to a node in the graph.

The global information of *detectors* and *generators* is also collected in a name record. These name records do not correspond to nodes in the graph. The first elementary items of the data block are the same as in the previous name records. However, the pointer is nil and the dimension is zero for *detector* types. If necessary, the data block is completed with elementary items, which represent the number of parameters and the types of the parameters. Apart from the name records just mentioned, the following name records exist:

- detector name or dname record.
  This name record has no parameter descriptors and the dimension is zero.
- symbol name or sname record.
  This name record has no parameter descriptors.

- curve name or cname record.
  This name record has parameter descriptors, i.e. the number of parameters, followed by the type of the parameters.

- template name or tname record.
  This name record has parameter descriptors as described by the cname record.

The type of a parameter in a *generator* can be *value*, *dimensional value*, *pname*, *aname* or *dname*.

3.2.2. Reference name records

An ILP construction that represents an explicit reference corresponds to the reference name records in IDS. The *pname* and *aname* in ILP are examples of such a reference. The corresponding reference pname and reference aname records contain one pointer to the name record of the reference. *Detection* in ILP (detectable, detectant element and undetectable) has its IDS counterpart in a reference dname record. The data block contains a pointer to the name record, which contains the name of the detectant and the string of the *detection*. The reference name record, which corresponds to a reference of a *generator* (reference sname, cname and tname record) has a number of data blocks according to the number of references in this *generator*. Each data block contains the parameter values, (possibly) the *attribute matches* and a pointer to the name record, which contains the name and the types of the parameters.

3.3. Data records

In a ring only data records are found which correspond to ILP elements of the type of the ring. The structure of the data blocks of the data record depends on the type the data record. In a *picture* ring there are always data records which correspond to *picture elements* and in an *attribute* ring the data records always correspond to *attribute* elements. The data records always correspond to end nodes (leaves) of an ILP graph. In IDS the following types of rings exist:

- *picture* ring

- *attribute* ring

- *subspace* ring

3.3.1. *Picture* ring data records

The elementary ILP constructions that correspond to data records are *coordinate type*, *text* and **NIL**. The **NIL** type data record is characterized by a **NIL** type and an empty data body (length of the data body is 0). For the other two elementary ILP constructions data blocks are formed in the following way. For every *coordinate* or *string* the *attribute match* for every *attribute class* is computed. In case of the *coordinate type* element successive *coordinates* are searched, which have equal *attribute matches*. These *attribute matches* are placed in front of such a row. In this way a semantically equivalent element is constructed. A data block consists of *attribute matches* and a *string* or a row of *coordinates*. Note that the two levels of *attribute matches* of ILP have been combined into one level.

3.3.2. *Attribute* ring data records

The *attribute* **NIL** data record is the same as the *picture* **NIL** data record. The *attribute class* elements can specify a large amount of data record types. They are split according to their *attribute classes*:

transformation
: *Scale*, *translate*, *matrix*, *affine*, *project* ORIGIN, *project* PARALLEL, *port* and *homogeneous* have a data body with one data block: a matrix. The structure of this matrix is the same as described by *picture coordinate* matrix.

style
: The *attribute class* elements fall into three groups of elements:

  line style
  : The data body consists of one data block of one real value or three small integers (*period*).

  point style
  : The data body consists of one data block of one elementary item. The following items are possible: small integer for a font, real for a value (size, italic, bold) and character value for a character (dot, marker).

  typographic style
  : The data body consists of one data block of one elementary item. The following items are possible: real for a value (size, italic, bold) and a small integer for a font.

The **TYPFAULT** data record does not contain any data blocks (length of the data body is 0).

pen

The **PENFAULT** data record does not contain any data blocks (length of the body is 0). All the different combinations (seven) of *contrast*, *intens* and *colour* have a corresponding data type. According to that data type a data body consist of one data block of real values.

control

This data record has a data body consisting of one data block: The length of the *string* and the characters of the *string* (string data block).

coordinate mode and *visibility*

These data records have an empty data body. The binary value of these *attribute*s is stored as part of the type of the data records.

3.3. ABS/REL *basic attribute*

Every *basic attribute* has a prefix **ABS** or **REL** (default is **REL**). In IDS a mark **ABS** or **REL** is added to the type of all records (except the link back record) in an *attribute* ring. Reference aname records, bracket records and the data records in an *attribute* ring all have this mark.

4. IDS COMPARED WITH OTHER DATA STRUCTURES

IDS has no means to manipulate its own data structures. In this respect IDS (like ILP) differs from most associative data structure languages. Here we will only compare the data structure itself.

IDS is a hierarchical data structure constructed with the help of one way single rings. One way means that the elements in the rings can only be found by traversing the ring (say) clock wise. Single means that an element of the ring does not participate in other rings as well. This structure, the simplest form of ring structure, is more than sufficient to represent an acyclic directed graph. In fact linear lists instead of rings would have been sufficient. The ring structure however is added in order to be able to return to the parent ring element (node) in an efficient way.

Multi list structures as present in sophisticated graphics systems like the Graphics System of General Motors and Univac are not included. Structures equivalent with these can be generated dynamically under control of attributes (detection) or by external means through association rings (for instance, to describe the topology). IDS represents purely graphical information for which these more complicated features are not necessary and moreover, non graphical information must be attached to IDS structures explicitly, leaving IDS data unchanged.

IDS meets the following requirements as generally accepted for graphics systems. Elementary pieces of a picture (like text or line) are stored as one unit separable from similar units. Their representation is a simple sequence of all relevant data in one record. All external references to IDS objects are eventually transformed into pointers to a record on the intended level in the hierarchy. From there all related records can be reached (e.g. all elements of the same ring, the parent or all descendants, etc.). Adding and deleting records is possible and efficient, due to the fact that the logical order is completely independent from physical arrangements.

IDS is too complicated for direct use as display file. A copy of IDS containing only picture elements collected in subroutines (if the display processor can handle these) must be produced. Depending on the *detection attribute* values,

these display files will contain references to the IDS data structure. After each update of IDS, the display file is (perhaps partly) regenerated. Essential is the concept that the display file is considered as part of the "hardware". In order to display the picture, which amounts to the generation of a display file, interpretation of the IDS data structure is required. The instruction set of such an interpreter together with the IDS data constitute a language comparable to other languages for handling complicated data structures.

5. IDS COMPARED WITH ILP

ILP and IDS are equivalent in the sense that there is a mapping defined from ILP to IDS and also from IDS to ILP. An ILP program mapped in this way onto an IDS data structure and back to ILP does, in general, not produce exactly the same symbolic ILP description. However, the picture produced by this program will be exactly the same as the picture produced by the original program.

An ILP program forms a symbolic description of data structures that are used to produce pictures. Since this description is in symbolic form, no commitment has to be made how these data structures should be represented on a specific computer. IDS is functionally equivalent to an ILP program. But implementation of IDS requires choices for the representation of all IDS constructions and data structures. This makes IDS more machine dependent then ILP.

IDS must realize the ILP concepts concerning compact representation. To this end the *subpicture* and *attribute_pack* are stored only once in IDS. This is comparable to the way in which these entities are treated in ILP. Moreover, to minimize referencing overhead, the machine independent but complicated naming strategy of ILP is, whenever possible, directly mapped onto the record link mechanism. This certainly provides a limited "name space", but the name space in IDS can be made sufficiently large by using an address dictionary or similar technique.

The subspace mechanism of ILP, which allows spaces of minimal dimension, is realized in IDS via a *subspace* ring.

Numerical values in ILP can have arbitrary size and precision. In IDS numbers are divided in classes with different range and precision. Rows of *coordinates*, *transformations* and *subspace* transformations are packed in a matrix (see APPENDIX I). These packed values may form less accurate but more compact representations.

With regard to the introduction of new *attributes* or even *attributes* over *attributes*, IDS is as flexible as ILP. All these new facilities can be implemented using the existing record ring structure.

IDS is designed to facilitate traversing the data structure. Moreover, edit or modification operations on the data structure are straightforward.

Information concerning multiple referenced objects (detectors, pictures etc.) is collected in name records. Specific properties of such objects are also stored in the

name records. The introduction of name records makes the interaction of external processes (outside IDS and ILP) with ILS straightforward. Compare this situation with the case of an ILP program where this interaction information must be collected during a traversing of the ILP program.

6. IMPLEMENTATION ASPECTS OF IDS

The correspondence between ILP constructions and IDS records is described in chapter 3. We will now introduce data records of IDS which have no counterpart in ILP. All IDS records are combined together with some extra data information in a picture file. This extra information concerns the links and physical addresses of the records (address dictionary) and expedites the access to a data structure by hashing techniques. Combining two picture files must be done in a unique way. At the end of this chapter an algorithm is given for the combination of two picture files.

6.1. The combination records

Many types of data records contain a small amount of information. To reduce the storage overhead caused by the explicit linkage of data records, it is possible to combine data records in a combination (data) record. The data body of this data record consists of sequentially ordered data records without their links. This is a way of bracketing that does not exists in ILP.

6.2. The continuation records

The capacity of the data body of records is limited. Since the data body of a **LINE** data record may be arbitrary long, it is some times necessary to divide the *coordinates* over two or more records. Splitting of the large **LINE** itself is not possible, since this may affect line styles. The only solution is to allow the continuation of a data body over several records. This facility is provided by continuation records.

6.3. The picture file

All aspects of IDS introduced in this paragraph do not influence the logical set up of IDS. Therefore they are left out of the discussion concerning this set up.

The IDS records corresponding to one ILP program are collected in a picture file. The number of records can be so large that we must consider problems concerning the size of the link and multi volume files. A mechanism must be provided that computes the physical record address from a link (value). A possible scheme is to attach unique indices to all records and define a mapping from index to an address dictionary, which in turn contains the physical address of the corresponding record. In this way the link size is reduced. In many cases the hardware enhances this type of dictionary system.

ILP provides a library facility and predefined generators. Name records realize this facility in IDS. Basically the name records are the contact points for internal as well as for external references (and hence, for the correlation between those). This is the first step towards interaction with IDS.

A fast way of fetching the name information is achieved by storing the name records sequentially in the file. This is called the name record part of the file. The speed of fetching named data can be improved by means of hash techniques. This requires a hash table to be part of the picture file and that a hash coding scheme is chosen and implemented. Hash coding must be transparent to the logical structure of IDS.

A picture file has the following structure:

- An identification mark that it is a picture file.
- The size of the address dictionary.
- The size of the name records part.
- The hash table.
- The address dictionary.
- The name record part.
- The other records.

A further consequence of the address dictionary and hashing technique is that two IDS files can be merged efficiently. The names from one data structure (file) are merged in the hash table and linked to the name record lists. The address dictionaries are set one after the other. When the records of the second file are being copied, the link(s) of each record are updated to point to the new dictionary and its physical address in the address dictionary is corrected.

The following algorithm can be used to merge two picture files. From the name a key is computed. This key corresponds to an entry in the hash table. The hash table consist of links, one for every entry. If more names have the same key, the corresponding name records are put into one list. The link of the hash entry points to this list. The links of the records are numbers of the entries of the address dictionary.

Algorithm: Merge two picture files.

Step 0. Make a start: Copy the first part one of the picture files (called the first).

Step 0.1. Copy the hash entries of the first file.

Step 0.2. Copy the address dictionary of the first file behind it.

Step 0.3. Copy the address dictionary of the second file behind it.

Step 0.4. Copy the name records of the first file behind it and mark these entries in the address dictionary.

Step 1. Merge the hash table of the first and second file into the new hash table.

Step 1.0. Fetch the next name record (initially the first) of the second file. If there are no more name records do step 2.

Step 1.1. Compute the hash entry of the name record. If the name is not new do step 1.2.
The link becomes the contents of the hash entry (set the name record as the first record in the list). The hash entry becomes the entry in the dictionary. Update and mark the address dictionary to the physical address and copy the name record. Do step 1.0.

Step 1.2. If the name is an *aname* or *pname* then the pointer in the data block must be updated, so do step 1.3.
Update and mark the address dictionary to the physical address of the name record of the first file. Do step 1.0.

Step 1.3. If the name records of both files have a pointer to a ring then announce an error and do step 1.0.
Update the address dictionary to the physical address of the record, which had a pointer unequal nil. Mark the entry. Delete the other name record or make this name record nil.
Do step 1.0.

Step 2. Update the physical addresses of the unmarked entries in the address dictionary of the first file.

Step 3. Merge the other records.

Step 3.1. Copy the rest of the records of the first file.

Step 3.2. Copy the next (initially the first) record of the second file.
If there is none then the merge is ended.

Step 3.3. The link of the record becomes the sum of the link entry and the number of entries of the address dictionary of the first file.

Step 3.4. If the record is a link record, then the pointers in the data body are updated according to the link in step 3.3,
else do the next step.

Step 3.5. The address dictionary entry corresponding to the physical address of the record of the second file is updated to its new physical address.
Do step 3.2.

7. THE VIRTUAL IDS STACK MACHINE

The semantics of ILP are described in chapter 3 of [1] by means of an interpretation process, called elaboration. In the following an implementation of this process on a virtual stack machine is outlined.

An ILP program consists of a description of the way a picture is to be drawn (*attributes*) and one or more actions (*picture_elements*). The *attributes* together with all information that changes dynamically as a result of actions is called the environment, in which the elaboration takes place. The part of the environment that is determined by *attributes* is called the full state. The additional information needed to turn a full state into an environment is called state extension. Examples of information from the state extension are the picture position, element position and pen position. The contribution to the state of the *attributes* between **WITH...DRAW** "brackets" is called a state component. The contribution of the *attributes* of an element path (see 3.4.1. [1]) derived from an ILP program is called state (i.e. all the state components on the element path). By combining a state with default attributes for all the *attribute_class* elements which were omitted a full state is obtained. According to *attribute_classes* the (full) state can be split into (full) state classes.

The state can be described in terms of an ILP *attribute pack*, named state.

```
ATTR state {
        {"transformations"};
        {"coordinate_mode"};
        {"pens"};
        {"styles"};
        {"visibility"};
        {"detections"};
        {"controls"}
}.
```

Here "...(s)" denotes the *attribute*(s) to describe the state class. Elaboration of the *picture_element* is equivalent with executing the ILP statement

```
WITH state DRAW picture_element .
```

Executing this statement has the following effects. The

state is combined with default *attributes* to a full state. Next the mode of a drawing device is updated according the corresponding full state. The *picture_element* is changed through the environment into zero or more *picture_elements*, which are fed in the device. Finally the state extension is updated.

Traversing the ILP graph implies that at all possible levels the current state must be saved. The state has to be changed temporarily, and must next be restored upon return from each descendant. Moreover, calling subpictures requires that return information is stored, preferably on the stack. The machine which perform these actions by traversing the IDS of a picture is called the IDS virtual stack picture machine or (picture) machine for short.

As we have seen before IDS is structured in such a way that logical ILP units are represented as IDS records. The machine knows three separate address spaces:

- Instruction space, containing IDS.
  In this space every record corresponds to a (virtual) machine instruction.

- The stack.
  The stack contains all temporary values that change temporarily. State values, which change temporarily are stacked in records, called stack records.

- Register space, which contains two kinds of registers:
  Link registers always contain a pointer. Examples are the program instruction register, pointing at the record in the instruction space that is currently being executed, and the stack register, pointing at the top of the stack.
  Value registers contain dynamically changing global values. An example is the value of the pen position.

Each instruction of the picture machine can be characterized by:

- push a value, a value of some kind is saved on the stack, or

- pop a value, a value of some kind is restored from the stack, or

- execute, a value is produced or consumed, either as input, output or in a register.

Each instruction will be described as a micro program of micro instructions. The micro instructions can be divided in push, pop and execute micro instructions.

The micro programs for execute instructions can be divided in three categories, depending on the ring in which the instruction record occurs. The execute instructions of the three rings use separate registers and data (as part of the instructions). Moreover, they are of such a different nature that they do not share common instructions of a lower level. For this reason, these instructions are conceptually executed by three different so called comachines, which in principle can run in parallel. These comachines need to synchronize only when a save or restore operation is required.

We will now explain how the environment (i.e. state and state extension) is represented in terms of the three address spaces. Next it is outlined how the state mechanism operates on records through their links. This knowledge is used to explain each record instruction at the micro code level.

The state extension is stored in the register space. For every state there exists a linked list of *attribute* records on the stack. A state descriptor, which has pointers to every state class list, is part of the state extension. Apart from this state descriptor, the state extension contains the following position values: element position (EP), transformed element position (TEP), transformed picture position (TPP), the pen position (PEN) and the transformed pen position (TPEN). Note that TEP, TPP and TPEN are values in the original user space. The register PR (program instruction pointer) points to the IDS record to be executed. The top of the stack is pointed to by the stack pointer (SP). The initial value of SP at the start of a picture comachine is kept in the initial stack pointer (ISP).

The state class values of the state can be found directly via the state descriptor. Executing the records of an *attribute* ring amounts, in general, to combining an *attribute* with the state class value. For every class this combining is done on a linked list of records, which describe the *attribute_class* value. The records of the lists are kept on the stack (linked lists of stack records). Some class values require a second level of linked lists, because each class value is made up of a number of independent atomic values. Each atomic value is represented as a second level list. Combining an *attribute* with a state consist of combining the *attribute* with a particular list. Combining can produce a new stack record at the head of the list or a complete new list on the stack. The state class descriptor pointer points (indirectly) to this new list.

We will now discuss the record instructions of the different comachines at the micro code level. Note that initially the picture comachine is started with PR pointing to

the rpname record of the picture to be elaborated.

7.1. The picture comachine

7.1.1. The data records

For the picture comachine the execution of all data records shows a similar scheme. The sequence of micro instructions for the execution of a data record has a common start and end:

```
begin
    if type of PR is not nil then
       begin
           complete(state);
           execute(PR);
           update(state extension);
               $ EP := PEN; TEP := TPEN $
       end;
    PR := link of PR $ next record $
end
```

A comment in a micro program starts and ends with a $ symbol. Let PD be a pointer to a data block in the record pointed to by PR. The micro instructions of the procedure next data block (PR) calculates the pointer to the next data block in the same record. Execute(PR) corresponds to executing the following micro instructions:

```
begin
    PD := next data block(PR);
    while PD is not nil do
       begin
           select state(attribute matches of PD);
           apply(state);
           PD := next data block(PR)
       end
end
```

7.1.2. The pointer records

If a pointer record associates a ring of a certain type with a picture ring, micro instructions are required to start the comachine of that type. IF the comachine has completed its instructions the state extension is updated. The values of the state extension, which must be saved, are pushed on the stack and the picture ring pointed to by the picture pointer is executed. The micro instructions consist of three blocks of micro instructions:

```
begin
    begin $ block 1 $
        push(link of PR); push(TPP);
        push(state descriptor);
        push(ISP); ISP := SP;
        comachine(link of (associate pointer of PR))
                  of associate type;
        update(state extension)
    end;
    begin $ block 2 $
        TPP := TEP;
        comachine(link of (picture pointer of PR))
                 of picture type;
    end;
    begin $ block 3 $
       SP := ISP; pop(ISP);
       pop(state descriptor);
       pop(TPP);
       pop(PR)
    end
end
```

The various pointer records are:

The bracket record
The micro program, which corresponds to this record, is:

```
begin
    push(link of PR); push(TPP);
    execute block 2;
    pop(TPP); pop(PR)
end
```

The subspace record
The associated *subspace* ring contributes a transformation to the state and a dimension to the state extension. Hence the state class *transformation* pointer and the dimension must be saved instead of the state descriptor. The associated type is *subspace*, so the subspace comachine is started executing the *subspace* ring. The micro program consists of executing block 1, 2 and 3.

The link back record
This is a return instruction to the calling comachine.

7.1.3. The mixed records

The various mixed records are:

The rpname and spname record
The root picture and subpicture records have a simple micro program:

```
    begin
    push(TPP);
    execute block 2 $ of pointer record
        micro instruction program $;
    pop(PR) $ if PR is nil stop machine $
end
```

The reference rpname and reference spname record
The micro program, which corresponds to this record is:

```
begin
    push(link of PR);
    PR := reference pointer of PR
end
```

The reference *generator* name record
A reference name record starts an external machine for every data block in the record. The name and the type of input for that external machine are given in the name record. The external machine builds an IDS in memory. A pointer to this IDS is returned to the picture machine. The picture machine executes this new structure, after which the IDS is removed. The reference name records starting such external machines are:
The reference sname record
PD is a pointer to a data block. Next data block (PR) gives the next data block of PR (initially the first, finally nil). The micro program is:

```
begin
    while next data block(PR) is not nil do
       begin
       PI := start external
           machine(PD);
       stack(TPP);
       execute block 2 $ of the
           pointer record micro program $;
       pop(TPP)
       end
end
```

The reference tname record
The micro instructions differ from the previous ones in that the external machine builds an IDS with pointers to the original picture (IDS) program. PD is as described above. The micro program is:

```
begin
    while next data block(PR) is not nil do
       begin
       PI := start external machine(PD);
       push(TPP);
       comachine(PI) of picture type;
       pop(TPP)
       end
end
```

The reference cname record

The external machine builds one data record with a *coordinate_type* and *attribute_matches* according to the specification in the data block. The micro program is:

```
begin
    PI := start external
       machine(description of PD,
           coordinate type of PD,
           parameter values of PD);
    execute(PI);
end
```

7.2. The attribute comachine

The actions of this machine are controlled by records from the *attribute* ring. The actions mainly consist of combining *attribute*s and the state into a new state. We will give an example of combining an *attribute* to a state class for the state class *detection*. The state class record, called detection record, has a data body with data blocks, one data block for every detector. The data block consists of three pointers. The first pointer points to the dname record (the name of the detector), the second pointer points to the detectant set (a list of reference dname records) and the third pointer points to the detectant element (one record of the detectant set or a record of the list of reference dname records). How *detection attribute*s are combined is outlined in the explanation of the instruction corresponding to the reference dname record.

The way *attribute* elements of a given class are combined is controlled by ABS and REL. Every record (*attribute*) in an *attribute* ring has an ABS/REL mark. The ABS/REL mark of a record, which contains a pointer to another *attribute* ring supersedes the ABS/REL mark of the data records of this *attribute* ring. This superseding holds only for the first data record for every state class. After combining the first data record of one state class, the other data records can be combined with a REL mark. To denote that the state class is changed during the execution there is a change mark for every state class. In the state exten-

sion there are for every state class two marks: The ABS/REL superseding mark and the changed/not changed mark. These marks are called the mark field. The mark field is stacked on entering an *attribute* ring and combined with the stacked field mark on return from an *attribute* ring. The combination of the marks of the field mark is done according to the following scheme:

old marks: stacked marks:	ABS/ not changed	REL/ not changed	REL/ changed
ABS/not ch.	ABS/not ch.	ABS/not ch.	REL/changed
REL/not ch.	REL/not ch.	REL/not ch.	REL/changed
REL/changed	REL/changed	REL/changed	REL/changed

ABS/changed is handled as REL/changed, as explained above, hence ABS/changed is no part of the scheme. The field mark is part of the state extension and for that reason a register value.

7.2.1. The pointer records

The various pointer records are:

The bracket record

The micro program is:

```
begin
    push(PR);
    push(field marks);
    set change marks of field marks to unchanged;
    if ABS/REL mark of the field marks is ABS then
       set mark to ABS
    else if mark of record is ABS then
       set mark to ABS
    else set mark to REL;
    comachine(pointer of PR) of attribute type;
    compute field marks $ see scheme above $;
    pop(PR)
end
```

The link back record

The corresponding instruction is a return instruction to the calling comachine.

7.2.2. The mixed records

The various mixed records are:

The aname record

The micro program is:

```
begin
    comachine(pointer of PR) of attribute type;
    pop(PR)
end
```

The reference aname instruction
The micro program is:

```
begin
    push(link of PR);
    PR := attribute pointer of PR
end
```

The reference dname record
The micro instructions of this record depend on the following type of reference dname record:

Absolute detectable
The dname reference record is pushed. The link of the stacked record becomes nil. The pointer of the corresponding detectant set and detectant in the detection record are set to the stacked record. The other pointers of the detection record are made nil.

Absolute detectant set element.
The dname reference record is pushed. The link of this stack record becomes nil and the pointer of the corresponding detectant set of the detectant record is set to the stacked record. All the other pointers of the detection record are made nil.

Absolute undetectable
All the pointers of the detection record are made nil.

Relative detectable
The dname reference record is pushed. The link of this stack record becomes the pointer of the corresponding detectant set. The pointers of the detectant set and the detectant of the detection record are set to the stacked record.

Relative detectant set element
The dname reference record is pushed. The link of this stack record becomes the pointer of the corresponding detectant set of the detection record. This pointer of the detection record is set to the stacked record.

Relative undetectable
The pointers of the corresponding detectant set and detectant in the detection record are made nil.

The last micro instruction of the micro program is:

PR := link of PR

7.2.3. The data records

The micro instructions of the data record consist of combining the data record to a state class. A new list of data records, which describe the state class is formed on the stack. The combining micro instructions differ for each state class or even for each atom and the type of the corresponding *attribute* in the *attribute_class*. Such a list of actions will not be given here.

7.3. The subspace comachine

The subspace selection contributes a transformation to the transformation state class, a new pen position (invisible move of the pen) and a (new) dimension. The transformation can be prepended to the list of *transformation* state class stack records. The transformation can not be combined with other transformations in the list while the ABS/REL *attribute* mark and the *attribute_match* TR may not cancel this transformation. Hence efficiency is improved if a pointer to the preceding subspace transformation is added in the subspace stack record. Finally the state extension is updated and the next record in the ring is executed (PR:= link of PR). This next record is a link back record and will stop the comachine (return to the picture comachine).

LITERATURE

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

[2] Robin Williams,
A Survey of Data Structures for Computer
Graphics Systems, Computing Surveys,
Vol. 3, No. 1, March 1971.

[3] J.C. Grag,
Compound data structure for computer aided
design; a survey, Proc. A.C.M.,
National Meeting, 1967.

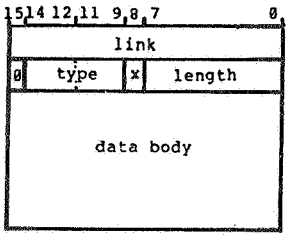
[4] George G. Dodd,
Elements of Data Management Systems,
Computing Surveys,
Vol. 1, No. 2, June 1969.

[5] Dennis M. Ritchie,
C Reference Manual,
Bell Telephone Laboratories, January 1974.

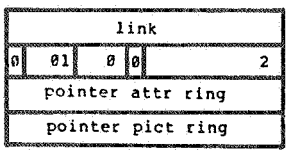
APPENDIX I IDS on a PDP 11/45

This is a survey of the structure of IDS records for a PDP 11/45. The first two words of the record form the header of the record. The first word (first and second byte) of the header are the link. The third byte is the length of the data body in units of words. The fourth byte has three fields of bit(s):
bit 0 is the ABS/REL mark.
bit 1 - bit 6 is the type.
bit 7: If this bit is 0 the record is of type pointer or mixed. If this bit is 1 the record is of type data.

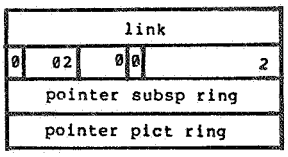
Records:



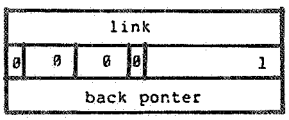
Pointer records:



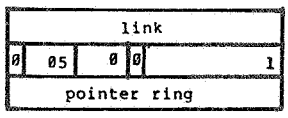
withdraw record



subspace record



link back record



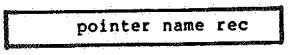
bracket record

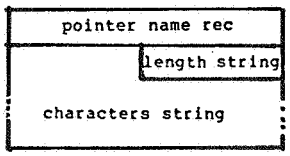
Mixed records:
Name records:

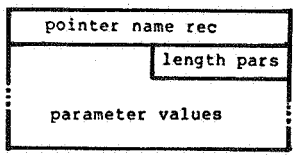
attribute subtypes:			picture subtypes:	
aname	detectable	001x	sname	101x
	set element	010x	cname gname	110x
	undetect	011x	tname	111x
			rpname	1001
			pname	1000

(a,p,d,s)name record (c,t)name record ref. name record

Reference name data blocks:

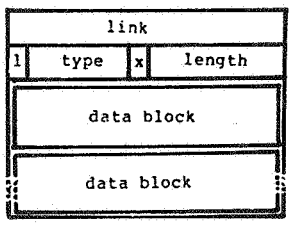


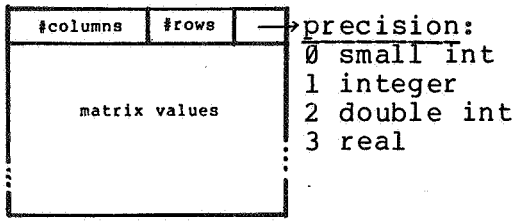


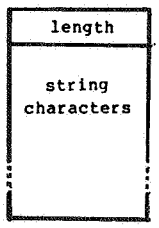


(a,p,s)name ref. dname ref. (c,t)name ref.

Data records: Some structures in a data block:





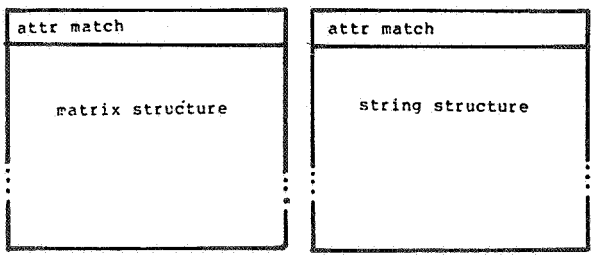


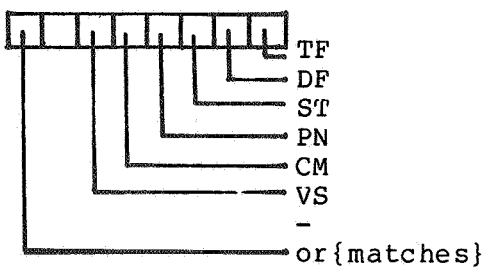
data record matrix structure string structure

Picture data records:

types	okt. code	data block structure	
line	02	attr. matches	matrix
contour	04	attr. matches	matrix
point	06	attr. matches	matrix
text	010	attr. matches	string
nil	0	-	-

Data blocks:





coord.type text

Attribute data records:

Type:

attr.type mode: ABS (1) / REL (0)

Attribute classes:

Transformation (00xxxx):

subtype	bin.code	oct.code	data block structure
nil	00 0000	0	-
rotate	00 0001	01	real+matrix
scale	00 0010	02	matrix
translate	00 0011	03	matrix
matrix	00 0100	04	matrix
affine	00 0101	05	matrix
project origin	00 0110	06	matrix
project parallel	00 0111	07	matrix
window	00 1000	010	matrix
window,viewport	00 1001	011	matrix
homogeneous	00 1010	012	matrix

Style (01xxxx):

subtype	bin.code	oct.code	data block structure
linestyle:			
period	01 0110	026	3*byte
map reset coord.	01 1101	035	real
map reset line	01 1110	036	real
map continue	01 1111	037	real
thick	01 0111	027	real
point style:			
dot	01 0100	025	byte
marker	01 0101	024	byte
typ.fault	01 0000	020	-
font	01 0001	021	byte
size	01 0010	022	real
italic	01 0011	023	real
bold	01 0100	024	real
typographic style:			
typ. fault	01 1000	030	-
font	01 1001	031	byte
size	01 1010	032	real
italic	01 1011	033	real
bold	01 1100	034	real

Pen (111xxx):

subtype	bin.code	oct.code	data block structure
fault	111 000	070	-
contrast	111 001	071	2*real
intens	111 010	072	real
colour	111 100	074	3*real
col.+intens	111 011	073	3*real
cont.+colour	111 101	075	5*real
int.+colour	111 110	076	4*real
cont.+int.+col.	111 111	077	6*real

Coordinate mode (10000x):

subtype	bin.code	oct.code	data block structure
fixed	10000 0	040	-
free	10000 1	041	-

Visibility (10001x):

subtype	bin.code	oct.code	data block structure
visible	10001 0	042	-
invisible	10001 1	043	-

Control (100100):

subtype	bin.code	oct.code	data block structure
control	100100	044	string

Combination record:
The type of this data record is 100111 (octal 047). The data body consists of data blocks, structured as a data record without a link.

Continuation record:
Continuation data records are only permitted in the ‹picture› ring. The type of this data record is the same as the ‹picture› data record which precedes this record. The distinction is made by the ABS/REL mark bit in the type. ABS (1) means that the data record is a continuation of the previous data record.

Picture file:

name	size
picture file identification	1 word "p"
address dictionary table size	1 word "a"
name record size	1 word "n"
hash table	"h" words "h" entries
address dictionary	a words a/2 entries
name records	n words
other records	x words

APPENDIX II IDS in the language C

This is a survey of the IDS structure in the language "C" [5].

```
\
/* record structure */
\define   ltype(p,q)     p<<4&q
struct    body{
     int  data[];
};
struct    record{
     struct    record    *link; /* ref record */
     char length;         /* \words in data body */
     char type;           /*bit[0] ABS/REL; bit[1:6] type;
                     bit[7] link (0) / data (1) record */
     struct    body databody; /* data */
};

/* link records */
\define   withdraw  ltype(01,0)
struct    withdrawbody{
     struct    record    *p_ring; /* ref picture ring */
     struct    record    *a_ring; /* ref attribute ring */
};

\define   subspace  ltype(02,0)
struct    subspacebody{
     struct    record    *p_ring; /* ref picture ring */
     struct    record    *s_ring; /* ref subspace ring */
};

\define   assoc     ltype(03,0)
struct    asocbody{
     struct    record    *p_ring; /* ref picture ring */
     struct    record    *assoc_ring; /* ref association ring */
};

\define   bracket   ltype(05,0)
struct    bracketbody{
     struct    record    *ring; /* ref ring */
};

\define   backlink  ltype(0,0)
struct    backlinkbody{
     struct    record    *back; /* ref ring */
};
```

```
/* name subtypes */
%define   sname     012  /* symbol name */
%define   cname     014  /* curve name  */
%define   tname     016  /* template name */
%define   rpname    011  /* root picture name */
%define   pname     010  /* subpicture name */
%define   aname     00   /* attribute name */
%define   dname     02   /* detector name */
%define   detect    02   /* detectable  */
%define   detset    04   /* detectant set element */
%define   undet     06   /* undetectable */

/* (ref) name type */
%define   name 07
%define   refname   06

/* string and matrix structure */
struct    string{
     char nrchrs; /* number of characters in string */
     char chars[]; /* actual string data */
};

/* precision */
%define smallint        0
%define integer         01
%define longint         02
%define realvalue       03

%define   descript(c,r,p)     (c|0377)<<8+(r|077)<<2+(p|03)

struct    matrix{
     int  m_descr; /* descript(%columns,%rows,%precision) */
     char matrixdata[]; /* actual values */
};

%define   maxlength 15
struct    ident{
     char lngth; /* length of the name */
     char chrs[maxlength]; /* actual name */
};

/* name record bodys */
struct    namebody{ /* root/subpicture, attribute, symbol, detector */
     struct    record    *ring; /* ref picture/attribute ring */
     char dim; /* dimension */
     char nrrefs; /* %references */
     struct    ident     identity; /* name */
};
```

```
/* parameter types */
%define   interval  01
%define   value           02
%define   dimval(dim)    04+(dim!03)
%define   refpname  010
%define   refaname  012
%define   refdname  013

struct    gen_body{
     struct    record    *nil; /* ref routine entry */
     char dim; /* dimension */
     char nrref; /* %references */
     struct    ident     identity; /* name */
     char nrpar; /* number of parameters */
     char par_type[]; /* parameter types */
};

/* reference name record bodys */
struct    dnameblock{
     struct    record    *namerec; /* reference to the namerecord */
     struct    string    dstring; /* detectant string */
};

struct    ctnameblock{
     struct    record    *namerec; /* reference to the name record */
     char parlength; /* length of the parameter values */
     char parameter[]; /* parameter values */
};

struct    apnamebody{
     struct    record    *namerec; /* reference to the name record */
};

struct    snamebody{
     struct    record    *snamerec[]; /* references to the name record *
};

struct    dnamebody{
     struct    dnameblock     dblock[];
};

struct    ctbody{
     struct    ctnameblock    ctblck[];
};

/* data records */
%define   dtype(p)  (p<<1)&0200
%define   nil  dtype(0,0)
```

```
/* picture data records */

/* attribute matches */
%define   tf    01
%define   dt    02
%define   st    04
%define   pn    010
%define   cm    020
%define   vs    040

/* coordinate type */
%define   line dtype(01)
%define   contour   dtype(02)
%define   point     dtype(03)

struct    coorddatablock{
     char attr_matches;
     struct    matrix    coords;
};

struct    coordbody{
     struct    coorddatablock cblock[];
};

%define   texttpe   dtype(05)
struct    textblock{
     char attr_matches;
     struct    string    text;
};

struct    textbody{
     struct    textblock tblock[];
};

/* attribute data records, types */
%define   ttype(p,q)     (p&q)<<1
/* transformations */
%define   trans     0
%define   rotate    ttype(trans,01)
%define   scale     ttype(trans,02)
%define   translate ttype(trans,03)
%define   matrix    ttype(trans,04)
%define   affine    ttype(trans,05)
%define   proj_org  ttype(trans,06)
%define   proj_par  ttype(trans,07)
%define   window    ttype(trans,010)
%define   w_view    ttype(trans,011)
%define   homo ttype(trans,012)

/* style */
%define   style     020
```

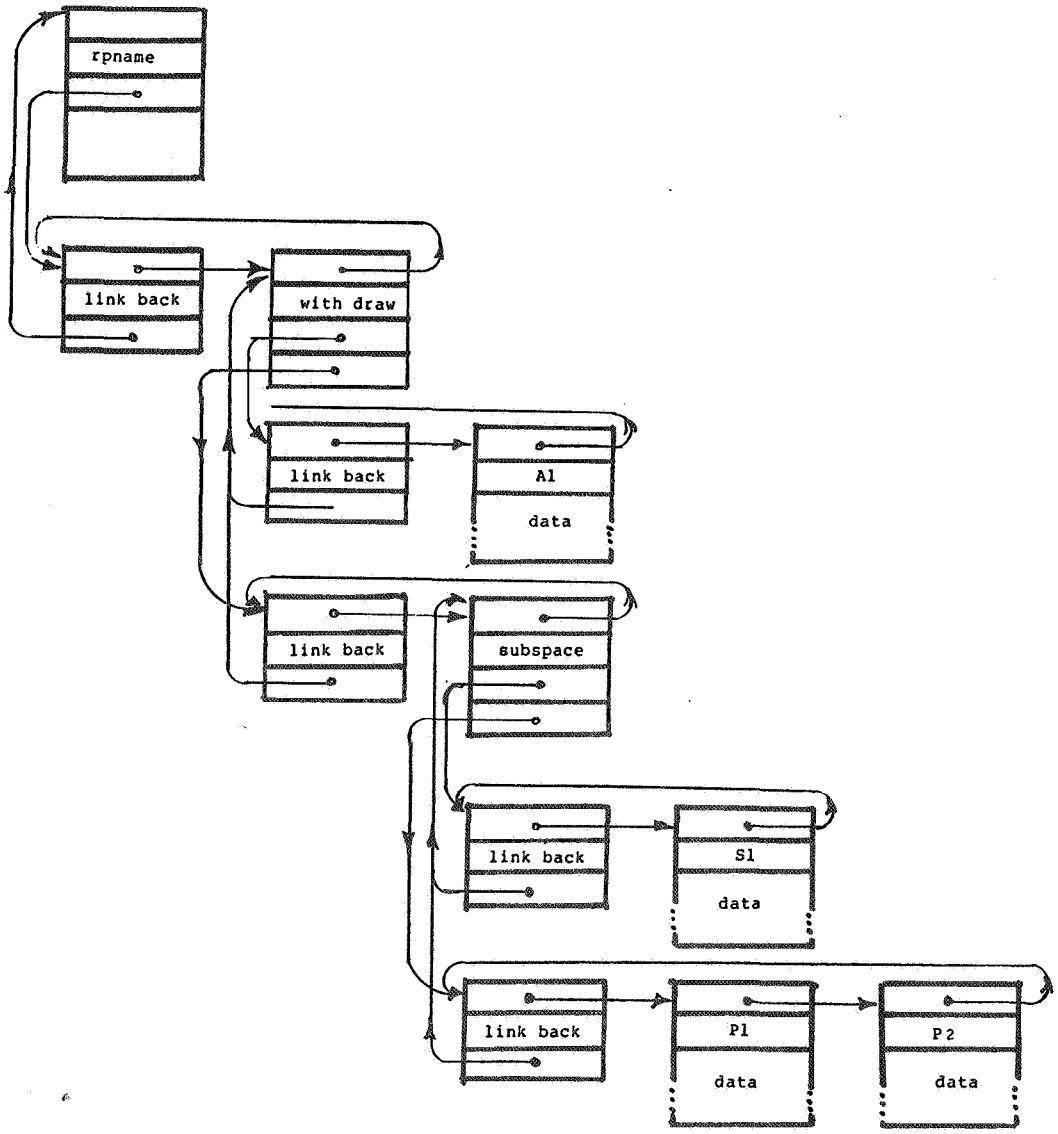
```
/* line styles */
%define   lperiod   ttype(style,06)
%define   lmaprc    ttype(style,015) /* map reset coordinate */
%define   lmaprl    ttype(style,016) /* map reset line */
%define   lmapc     ttype(style,017) /* map continue */
%define   lthick    ttype(style,07)
/* point style */
%define   pdot ttype(style,05)
%define   pmarker   ttype(style,04)
%define   ptfault   ttype(style,0)   /* type fault */
%define   ptfont    ttype(style,01)      /* type font */
%define   ptsize    ttype(style,02)  /* type size */
%define   ptital    ttype(style,03)  /* type italic */
%define   pbold     ttype(style,04)  /* type bold */
/* text style */
%define   ttfault   ttype(style,010) /* type fault */
%define   ttfont    ttype(style,011) /* type font */
%define   ttsize    ttype(style,012) /* type size */
%define   ttital    ttype(style,013) /* type italic */

/* pen */
%define   pen  070
%define   pfault    ttype(pen,0)
%define   pcontr    ttype(pen,01)
%define   pinten    ttype(pen,02)
%define   pcolour   ttype(pen,04)
%define   pcolint   ttype(pen,03)
%define   pconcol   ttype(pen,05)
%define   pintcol   ttype(pen,06)
%define   pcoint    ttype(pen,07)

/* coordinate mode */
%define   fixed     ttype(0,040)
%define   free ttype(0,041)

/* visibility */
%define   vis  ttype(0,042)
%define   invis     ttype(0,043)

/* control */
%define   control   ttype(0,44)

/* combination record */
%define   comb dtype(047)

/* picture file */
struct    addentry{
     int  *address;
     char extent;   /* address extention */
     char flags; /* system */
};
```

```
struct    pictfile{
     int  picture; /* picture file */
     int  addsize; /* address dictionary size */
     int  namesize; /* name records size */
     int  *hashtable[]; /* hash table */
     struct    addentry  addrss[]; /* address dictionary */
     struct    record    nrec[]; /* name records */
     struct    record    rec[]; /* records */
};
```

APPENDIX III An ILP and IDS example

An example of an ILP program and its corresponding IDS "program". Elements (end nodes) of the *attribute*, *subspace* and *picture* are denoted by a, s and p respectively followed by an index.

```
PICT pa
     WITH a1
     DRAW SUBSPACE s1
          WITH a2
          DRAW { p1 ; p2 } .
```



INDEX