stichting mathematisch centrum

MC

AFDELING INFORMATICA IW 102/78 DECEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

H.J. SINT

ALGOL68G-0: PICTURES REPRESENTED IN ALGOL 68

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 68A10, 68A30

ACM-Computing Reviews-categories: 4.22, 4.34, 8.2.

Algol68G-0: Pictures represented in Algol68

by

H.J. Sint

ABSTRACT

A method is explained to construct a set of Algol68 declarations which are, in a well defined sense, equivalent with a given context free grammar.

The method is applied to the graphics language ILP. The result is a library prelude which contains mode declarations allowing the representation of graphical data, and procedures to translate data thus represented to ILP and vice versa.

KEYWORDS & PHRASES: *Computer graphics, ALGOL68, language extension.*

1. Introduction

An important goal of the graphics project currently conducted at the Mathematisch Centrum is the design of a well structured, high level graphics language. The philosophy underlying this part of the project, can be summarized by the following statements:

- The high level graphics language (HLGL for short) should borrow its control structure entirely from an existing programming language, the host language.

- The graphical part of the HLGL should consist of an interface with a low level, but still device independent, intermediate graphics language (IL).

- The IL should contain graphical elements only. Familiar constructs like assignments, conditional expressions, loops and procedures are not included.

- The interface between the IL and the HLGL should be two-way: The HLGL should be able to produce as well as to accept and process programs in the IL.

Moreover, a first host language (Algol68) and implementation technique (writing an extension in the form of a library prelude) were chosen [1].

The first step toward the realisation of these ideas has been the design of an intermediate language, ILP (Intermediate Language for Pictures), defined in [2].

In this report the interface between ILP and Algol68 is described. Its main purpose is to establish a formal relationship between ILP and the Algol68 extension defined at the end of the report. This is achieved by first presenting a general method for embedding a context free language into Algol68, and then applying that method to ILP, with some extras to handle the non context free features. Readers who are only interested in the language defined can turn immediately to section 3.3. They are warned however that only the kernel of the high level graphics language Algol68G is defined here, providing nothing but a set of graphical modes and translation procedures to and from ILP.

2. <u>The embedding of a context free language into Algol68</u>

2.1. <u>Preliminaries</u>

If one uses a manipulator for some language L, which consists of a set of operations on sentences of L, one would like to be sure that whatever sequence of operations is applied, the final result is still a sentence of L.

We propose to use the Algol68 mode mechanism to ensure that a set of manipulator operations is closed under function composition. This is achieved by defining some mode M, such that there is a one to one correspondence between values of M (acceptable to M, in Algol68 jargon) and sentences of L, and two procedures to translate sentences of L to their corresponding M value and vice versa. A manipulator of L can then be written completely in terms of operations on objects of mode M, according to the following scheme:

$$\begin{tikzcd}
m \in M \arrow[r, "O_1"] & \cdots\cdots \arrow[r, "O_n"] & m' \in M \arrow[d, "\text{translator} \\ M \longrightarrow L"] \\
s \in L \arrow[u, "\text{translator} \\ L \longrightarrow M"'] & & s' \in L
\end{tikzcd}$$

(Of course, without further knowledge about the manipulator operators themselves, nothing beyond the protection provided by the Algol68 mode checking mechanism is guaranteed. If m' turns out to be uninitialized for instance, the second translator will crash. But if the manipulator terminates without crashing, then it is certain that the output of the second translator is in L).

This chapter formalizes this method for the case that L is non ambiguous and context free.

The obvious way to produce the desired mode declarations is to associate a new mode with <u>each</u> symbol X in a grammar producing L, and to make sure that there is a one to one correspondence between the values of that mode and the terminal productions of X. But here we get in trouble. If X is a terminal, we want a corresponding mode having only <u>one</u> value. Algol68 provides exactly one such mode, namely VOID, and no way to construct any other. In practice getting around this (not very orthogonal) restriction is not much of a problem; we can use other modes which have more values (two for instance) and then simply treat these values as if they were indistinguishable. It does however obstruct a clean definition of an embedding and an easy correctness proof of the algorithm that constructs such an embedding from a given grammar. Therefore, we will define and use an extension of Algol68 which has the required feature.

We introduce an additional PLAIN mode, (denoted by ONE), and specify that (like VOID) this mode has only one value (denoted by ONLY). Unlike VOID however, this mode can be used freely for the construction of other modes. One can write for example

```
MODE A = STRUCT(ONE a);
MODE B = STRUCT(ONE b);
```

thus obtaining two different modes both having one value. We will denote this extension by Algol68'.

2.2. Definition

An embedding of a non ambiguous, context free language L into Algol68' consists of at least the following components:

- A declaration for a mode M;
- A declaration for a procedure <u>ltoa68</u> of mode PROC(STRING)REF M;
- A declaration for a procedure <u>a68tol</u> of mode PROC(M)STRING;

with the following properties: (1)

1. For all strings w ∈ L, ltoa68(w) ∈ M ;
2. For all strings w ∉ L, ltoa68(w) :=: NIL;
3. For all strings w1, w2 ∈ L, w1 = w2 <=> ltoa68(w1) = ltoa68(w2);
4. For all values m ∈ M there is a string w ∈ L such that a68tol(m) = w;
5. For all values m ∈ M and all strings w ∈ L, a68tol(m) = w <=> ltoa68(w) = m.

In the next section we present an algorithm that constructs an embedding for some language L, given a grammar G that produces L. In principle this algorithm consists of three parts:

1 A component that constructs a set of mode declarations, forming a mode tree with root M;

2 A component that constructs a translator from mode M to L;

---

(1) The relations "=" and "∈" cannot be defined as Algol68 operators. In section 2.4.2 we will give formal definitions. ∈ means something like "accepted by" as used in [3].

3 A component that constructs a translator from L to mode M.
This translator has two tasks:

- It has to decide whether a given sentence belongs to L, i.e. it must contain a parser.

- It has to construct an appropriate value of mode M for each sentence that does belong to L.

We have chosen to separate the first task - parsing - from the remainder of the algorithm. Parser generating is a thoroughly studied problem that is not of particular interest in this context. Most parser generators moreover impose restrictions on the input grammar which also restrict the class of languages handled; and these restrictions are not required for any other part of the algorithm. Therefore, we will initially ignore the problem of generating a parser altogether and only construct a translator from derivation trees to M.

In section 2.4. we will consider the correctness of the algorithm. In section 2.5. we will present a generator for a recursive descent parser written in Algol68 which produces a derivation tree in the form required by the algorithm.

2.3. An algorithm producing an embedding from a CFG

2.3.1. Input

The algorithm requires as input a context free grammar G of the following form:

$G = \{T, N, P, S\}$
where $T = \{T_i\}_{i=1}^{t}$ is a set of terminals.
ε can be a member of T.

$N = \{N_i\}_{i=1}^{n}$ is a set of nonterminals,

$P = \{N_i \rightarrow \{\{E_{ijk}\}_{k=1}^{l_{ij}}\}_{j=1}^{m_i}\}_{i=1}^{n}$
is a set of production rules,
m_i is the number of alternative productions of the i-th non terminal,
l_{ij} is the number of symbols in the j-th alternative of the i-th nonterminal,

$\{E_{ijk}\}_{k=1}^{l_{ij}}$ is the j-th alternative of the i-th nonterminal,
E_{ijk} is the k-th symbol of that alternative and a member of $N \cup T$.

a_{ij} will be used as an abbreviation for $\{E_{ijk}\}_{k=1}^{l_{ij}}$,

$S \in N$ is the start symbol.

The algorithm explicitly requires that each non terminal occurs only once on the left hand side; i.e. that all alternative productions of a non terminal are gathered into one member of P.
G must moreover meet the following requirements:
- G is not ambiguous,
- G can not give rise to productions of the form $N_i \xrightarrow{*} N_i$.

2.3.2. Notation

The algorithm as presented here is a slightly less formal but slightly more readable transscription of a complete Algol68 program, without the parser generator and all administrative procedures. The notation used is a mixture of Algol68 and calls to two special, macro like procedures. The arguments of those procedures are delimited by brackets {}, to distinguish them from ordinary procedure calls.

The procedure gen serves as a pair of quasi-quotes. Gen literally copies its argument to an output file, with the exception of elements preceded by a dot (.), and of calls to the second macro. The dotted elements (most of them are procedure calls) are first evaluated by the Algol68 system, the result (if necessary converted to a string) replaces the dotted element in the output of gen. The calls to the second macro are first evaluated and then put in the input of gen.

The other macro used is denoted by <u>var → lim:</u>.This macro first has the Algol68 system retrieve the value of lim (which should be an integer) and then copies its argument that number of times in succession, leaving everything (including dotted elements) as it was except that the parameter "var" is replaced by 1 in the first copy, by 2 in the second, etc. Moreover, it skips the last character before the } the last time (which is what you most of the time want for separators). An instantiation of this operator occurs in the second example given below: k → times.

To discriminate as clearly as possible between the (Algol68-like) language in which the algorithm is written, and the Algol68 statements it produces, we use different conventions: In the algorithm bold words will be surrounded by single quotes, and subscripts will be used instead of indices (a_{ij} instead of a[i,j]). In the output bold words will appear in CAPITALS and variables will be indexed instead of subscripted.

Comment on the algorithm is given between #-signs.

A few examples will hopefully clarify this notation:

if i=1, T[1] = "a", B[1] = "A", and double is a procedure
PROC double = (STRING s)STRING: s + s;
then gen{MODE .B_i = STRUCT(ONE .double(T_i); } will generate
```
        MODE A = STRUCT(ONE aa);
```

```
If  times = 3,
then gen{IF x = "" THEN proc0(x)
            k  → times:
            {ELIF x = .(k * "a") THEN proc.k (x) }
         FI}
```

will generate

```
IF x = "" THEN proc0(x)
   ELIF x = "a" THEN proc1(x)
   ELIF x = "aa" THEN proc2(x)
   ELIF x = "aaa" THEN proc3(x)
FI
```

There is a straightforward translation between the macro-notation and Algol68 programs. The Algol68 equivalent of the last example for instance is:

```
 'string' ifs := "IF x = """" THEN proc0(x)" + newl;
 'for' k 'to' times
 'do' ifs +:= "ELIF x = " + k * "a" + " THEN proc" + whole(k,0)
              + "(x) " + newl
 'od';
  ifs +:= "FI";
  pprintt(ifs);

 # pprintt is a function which interpretes the character newl as a newline #
```

This illustrates that programs in the macro-processor language are more readable than their algol68-equivalent.

The algorithm uses the following (administrative) procedures:

- <u>tag</u> and <u>boldtag</u> have one argument and return a unique Algol68 tag and boldtag associated with that argument:
  tag(s1) = tag(s2) <=> s1 = s2;
  boldtag(s1) = boldtag(s2) <=> s1 = s2.

These two procedures achieve consistent naming.

- s (abbreviation for suffix) has a triple (i,j,k) as argument and returns some suffix uniquely associated with that triple. s has moreover the following property: If $a_{ij} = a_{pq}$, then s(i,j,k) = s(p,q,k) for all $k <= l_{ij}$.
  The triple (i,j,k) must be seen as pointing to a specific occurrence of a symbol in the grammar. Use of s guarantees that different (and reproducable) field selectors are associated with different occurrences of the same symbol in the same right hand side alternative. In section 2.4.1. we will state some additional properties of tag, boldtag and s.

- '<u>old</u>' is an operator which has an a_{ij} as argument and returns a boolean: 'true' if the same production was found before in the grammar, i.e. if there is an a_{pq} such that $p < i$ and $a_{ij} = a_{pq}$.

Appendix 1 contains an example of the output produced by the Algol68 version of the embedding algorithm.

2.3.3. The algorithm

```
1. # Generate mode declarations:                                        #

   # Invent for each terminal a unique mode having only one value.      #

   'for' i 'to' t
   'do' gen{MODE .boldtag(T_i) = STRUCT(ONE .tag(T_i) )} 'od';

   # For each right hand side alternative a structured mode is genera- #
   # ted, provided that it consists of more than one symbol and that   #
   # it did not occur in the grammar previously.                        #
```

```
'for' i 'to' n
'do' 'for' j 'to' $m_i$
'do' 'if' $l_{ij}$ > 1 'and' 'not' 'old' $a_{ij}$ 'then'
gen{
    MODE .boldtag($a_{ij}$) =
        STRUCT($k \rightarrow l_{ij}$: {REF .boldtag($E_{ijk}$) .tag($E_{ijk}$)s(i,j,k) ,}
   }
     'fi'
'od';

# For nonterminals with one production an equivalence declaration    #
# is generated. For nonterminals with more productions, a union is   #
# generated.                                                          #
'if' $m_i$ = 1 'then'
     gen{MODE .boldtag($N_i$) = .boldtag($a_{i1}$)}
'else'
     gen{MODE .boldtag($N_i$) = UNION($j \rightarrow m_i$: {.boldtag($a_{ij}$),} );}
'fi'
'od';
```

2. # generate a translator from Algol68 to L(G): #

```
# For each terminal an output procedure is generated which simply    #
# returns that terminal. The algorithm is supposed to be able to     #
# handle ε correctly:                                                 #

'for' i 'to' t
'do'
gen{
    PROC a68to.tag($T_i$) = (.boldtag($T_i$) .tag($T_i$) )STRING: ".$T_i$";
   }
'od';
```

```
# For each right hand side alternative consisting of more than one #
# symbol, a procedure is  generated which  calls the procedures    #
# generated for each symbol and concatenates the results:          #

'for' i 'to' n
'do' 'for' j 'to' $m_i$
'do' 'if' $l_{ij}$ > 1 'and' 'not' 'old' $a_{ij}$ 'then'
gen{
    PROC a68to.tag($a_{ij}$) = (.boldtag($a_{ij}$) .tag($a_{ij}$))STRING:
    k→$l_{ij}$: {a68to.tag($E_{ijk}$)(.tag($E_{ijk}$).s(i,j,k) OF .tag($a_{ij}$) ) +};
   }
'fi'
'od';

# For all nonterminals for which a united mode was invented, a     #
# procedure is generated which calls the procedure corresponding   #
# to the mode of its argument, and passes on its result.           #
# If there was only one right hand side, the procedure body is a   #
# simple procedure call.                                           #

'if' $m_i$ = 1 'then'
gen{
    PROC a68to.tag($N_i$) = (.boldtag($N_i$) .tag($N_i$)) STRING:
    a68to.tag($a_{i1}$)(.tag($N_i$) );
   }
   'else'
gen{
    PROC a68to.tag($N_i$) = (.boldtag($N_i$) .tag($N_i$)) STRING:
    CASE .tag($N_i$) IN
    j→$m_i$:
    { (.boldtag($a_{ij}$) .tag($a_{ij}$) ): a68to.tag($a_{ij}$)(.tag($a_{ij}$)),}
    ESAC;
   }
'fi'
'od';
```

3. # Generate a translator from L(G) to Algol68:

As said before, we will for the moment assume that a parser for L(G) is somehow available. This parser should accept a string s as input and produce a value of mode REF PARSETREE: the nil-reference if $s \notin L(G)$, a non-nil value if $s \in L(G)$; where

```
MODE PARSETREE = STRUCT(STRING node,
                        FLEX [1:0] REF PARSETREE descendants);
```

The node field of a PARSETREE is either a terminal, a right hand side production or a nonterminal.
If the node field is a terminal, the descendants field is the empty array (lower bound 1, upper bound 0).
If the node field is a right hand side production a_{ij}, the descendants field contains l_{ij} pointers to the parsetrees for all symbols in a_{ij}.
If the node-field is a nonterminal, the descendant field contains one element: a pointer to the parsetree for the successful alternative.

Example: A parser for the language produced by $S \rightarrow aSb \mid c$, produces the structure shown in figure 1 for the string "aacbb". #

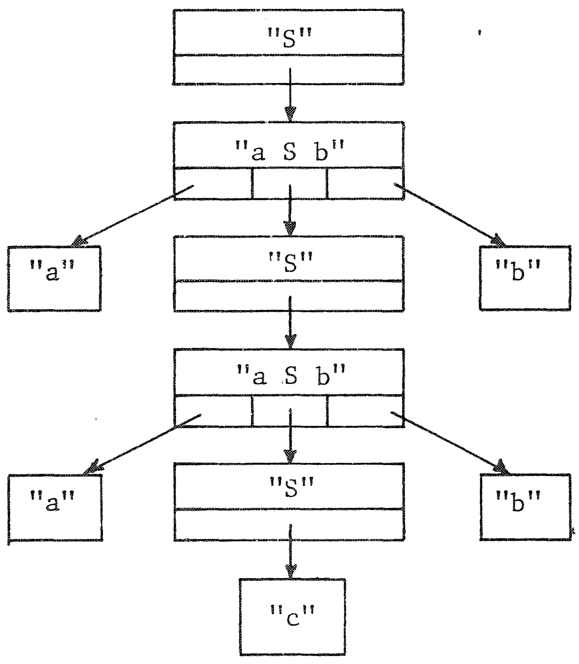

figure 1

```
# Generate a translator from mode PARSETREE to Algol68:                  #

# For each terminal, a procedure is generated which returns the only     #
# value of the mode associated with that terminal.                       #
'for' i 'to' t
'do'
gen{
    PROC .tag($T_i$)toa68 = (REF PARSETREE p) REF .boldtag($T_i$):
    (HEAP .boldtag($T_i$) t;
     .tag($T_i$) OF t := ONLY;
     t);
   }
'od';

# For each right hand side alternative consisting of more than one      #
# symbol a procedure is generated, which for each symbol calls the      #
# procedure  associated  with it and  assembles the results into a      #
# structured value.                                                      #

'for' i 'to' n
'do' 'for' j 'to' $m_i$
'do' 'if' $l_{ij}$ > 1 'and' 'not' 'old' $a_{ij}$ 'then'
gen{
    PROC .tag($a_{ij}$)toa68 = (REF PARSETREE parsetree) REF .boldtag($a_{ij}$):
    HEAP .boldtag($a_{ij}$) :=  (
     k $\rightarrow$ $l_{ij}$: { .tag($E_{ijk}$)toa68((descendants OF parsetree)[k]),}
    );
   }
'fi'
'od';

# For each nonterminal a procedure is generated which finds out,        #
# which alternative production of that nonterminal actually occurs      #
# in the parsetree, and then calls the procedure associated with that #
# alternative. If there was only one production, no testing, derefe-   #
# rencing and uniting is needed: the right procedure can be called      #
# immediately and the result passed on.                                  #
```

```
'if' $m_i$ = 1 'then'
gen{
    PROC .tag($N_i$)toa68 = (REF PARSETREE parsetree)REF .boldtag($N_i$):
    .tag($a_{i1}$)toa68((descendants OF parsetree)[1]);
   }
'else'
gen{
    PROC .tag($N_i$)toa68 = (REF PARSETREE parsetree)REF .boldtag($N_i$):
    (REF PARSETREE child = (descendants OF parsetree)[1];
     HEAP .boldtag($N_i$) :=
     j → $m_i$-1:
     {IF node OF child = .$a_{ij}$  THEN  .tag($a_{ij}$)toa68(child) ELSE }
      .tag($a_{im_i}$)toa68(child)
     j → $m_i$-1:
     {FI }
    );
   }
'fi'
'od';

# Finally, a procedure is generated which accepts a string as input,  #
# calls the parser, returns NIL immediately if the parser returns     #
# NIL, otherwise calls tag(S)toa68 with the result as argument and    #
# returns the result of that call.                                    #
gen{
    PROC ltoa68 = (STRING w) REF .boldtag(S):
    (REF PARSETREE parsetree = parser(w);
     IF parsetree :=: REF PARSETREE (NIL)
        THEN NIL
        ELSE .tag(S)toa68(parsetree)
     FI);
   }
```

2.4. Correctness of the algorithm

There are two different kinds of considerations concerning the correctness of the algorithm. First, we must convince ourselves that syntactically correct Algol68' programs are produced. Second, we must convince ourselves that the produced programs indeed constitute an embedding.

2.4.1. Syntax of produced programs

We have not tried to proof formally that the produced programs are syntactically correct. We have considered a lot of potential errors and have taken measures to circumvent them. Programs produced by the Algol68 version of the algorithm are accepted by the CDC Algol68 compiler, (after we define "MODE ONE = BOOL" and "ONE ONLY = TRUE", surround the program by a BEGIN/END pair and insert a dummy statement before this final END).

We give a list of potential errors, and an informal explanation how they are circumvented. RR refers to [3].

1 No undefined modes:
One mode is defined for each terminal, nonterminal and unique right hand side. The procedure "boldtag" always has an argument belonging to one of these three categories and hence always produces a mode indicator for a mode defined somewhere in the program (though not necessarily before its first application).

2 No redefined modes:
Terminals and nonterminals are unique. "boldtag" is defined in such a way that it never generates a reserved tag (i.e. any representation of a bold symbol listed in RR 9.4.1.), nor will it generate "ONE" or "PARSETREE". The use of the operator 'old' guarantees that no mode for a right hand side production is generated twice.

3 No circular and/or infinite modes:
There are no productions of the form $N_i \xrightarrow{*} N_i$. If N_i is nevertheless

recursive, we always have the case
$N_i \xrightarrow{*} X_1 N_i X_2$, with X_1, $X_2 \in (N \cup T)^*$ and not both empty.
There are $N_j \in N$, and Y_1, $Y_2 \in (N \cup T)^*$, such that
$N_i \xrightarrow{*} N_j \rightarrow Y_1 N_k Y_2 \xrightarrow{*} X_1 N_i X_2$
(The left and right production chains can be empty). MODE boldtag(N_j) will be declared as a structure with a field of mode REF boldtag(N_k): Hence, the spelling of N_i passes through at least one STRUCT (preventing circularity) and at least one REF (preventing infinity) between definition and application (RR 7.4.1.).

4 No incestuous unions:
After unraveling, all members of unions will turn out to be structured modes, which cannot be coerced to anything (RR 4.7.)

5 No unions with one member:
We gave nonterminals with one production a separate treatment in order to avoid unions with one member.

6 No double field selectors in structures:
Use of procedure "s" prevents that. It produces a suffix consisting of the smallest number of "i"-s necessary. For a right hand side production "aaabb" for example, the following mode definition is generated (supposing boldtag("aaabb") returns "AAABB" etc):
MODE AAABB = STRUCT(REF A a, REF A ai, REF A aii, REF B b, REF B bi).
See also the next point.

7 No undeclared or redeclared identifiers:
The same arguments as for undefined and redefined modes; only "tag" takes over the role of "boldtag". We have made sure that tag never produces the identifier "lg", "child", "parsetree" or "a68", nor any special identifier generated by the parser generator, nor any identifier whose last letter is an "i" (to avoid generating STRUCT(REF AI ai, REF A a, REF A ai) for a right hand side production "ai a a").

8 No mode errors.
This point was treated more formally, as will be mentioned again in the next section.

9 Array indexing.
All array indices occurring in the output of the algorithm are simple constants. A superficial glance at the algorithm may suggest the opposite, but closer inspection will learn that the algorithm itself replaces all variable indices by constants.

2.4.2. Do produced programs constitute an embedding?

We have formally proved that programs produced by the algorithm do not contain mode errors and satisfy the embedding requirements as defined in 2.2. Constructing this proof was not difficult but the result is rather long and tedious. Instead of reproducing the complete proof we will only mention some points that came forward as a result of this effort.

It will be clear that the following substitutions in the definition should be made:

```
M      -> .boldtag(S); (S is the start symbol of the grammar G)
a68tol -> a68to.tag(S).
```

Equality. The notion of equality as used in point 3 needs a definition, which is less straightforward than it may seem intuitively. Consider the following piece of program:

```
MODE INTCHAR = UNION(CHAR, INT);
INTCHAR IC = "c";
CHAR c = "c";
```

Whether or not ic equals c seems to be a matter of taste or of circumstances. We first define an equality relation on (MODE, value) pairs.

Definition

Let M_1, M_2 be mode declarers and v_1, v_2 be values.
Then $(M_1,v_1) = (M_2,v_2)$ iff
M_1 and M_2 specify equivalent modes,
and
either (M_1 and M_2 are PLAIN modes and $v_1 = v_2$ where "=" denotes the Algol68 equality operator),
or (M_1 and M_2 are of the form REF M_3 and either v_1 :=: v_2 :=: NIL or $(M_3, M_3(v_1)) = (M_3, M_3(v_2))$),
or (M_1 and M_2 are of the form STRUCT(F_1 t_1, ..., F_n t_n) and (F_1, t_1 OF v_1) = (F_1, t_1 OF v_2) and ... and (F_n, t_n OF v_1) = (F_n, t_n OF v_2),
or (M_1 and M_2 are of the form UNION(M_1, ..., M_n) and there is a member M_i in the UNION such that $(M_i, v_1) = (M_i, v_2)$).
In all other cases $(M_1, v_1) = (M_2, v_2)$ is undefined.

(We don't have to worry about arrays and procedures since they don't occur in values delivered by lgtoa68).
Using this definition we can formulate an equality relation on values:
$v_1 = v_2$ iff there is a mode M such that $(M, v_1) = (M, v_2)$.

The relation $\in$. In 2.2. we circumscribed the relation $\in$ between values and modes as "similar to the relation 'accepted by' as used in the Algol68 report". This formulation was deliberately vague; in order to proof that produced programs satisfy the embedding property, we have to restrict this relation in the way implied by the following definition:

Definition

Let M be a mode declarer and v be a value.
Then $v \in M$ iff
Either M is a PLAIN mode and v is accepted by M,
or M is of the form REF M' and v :/=: NIL and $v \in M'$;
or M is of the form STRUCT(F_1 t_1, ..., F_n t_n) and t_1 OF $v \in F_1$, and ... and t_n OF $v \in t_n$;
or M is of the form UNION(M_1, ..., M_n) and there is a member M_i of the union such that $v \in M_i$.
In all other cases $v \in M$ is undefined.

The important restriction lies in the case that M is of the form REF M', where v is not allowed to be NIL. The reason for this restriction is the following. Structured modes produced for right hand side productions contain references in all fields. These references are necessary in order to avoid the production of infinite modes for recursive grammars, but they spoil the embedding property as envisaged: Each reference introduces additional values accepted by the mode which do not correspond to terminal productions. Hence point 4 of the definition (For all m $\in$ M there is a w $\in$ L such that a68to.tag(S)(m) = w) can only be proved under the additional assumption that the value tree for m does not contain any NIL's.

A possible way to mend this rather serious flaw in the embedding strategy was suggested by D. Grune: use PROC M instead of REF M in the fields of structured modes. If we then also extend the definition of equality on (MODE, value) pairs with the case "or M_1 and M_2 are of the form PROC M_3 and (M_3, $M_3(v_1)$) = (M_3, $M_3(v_2)$)" we can indeed replace the relation $\in$ by "accepted by". We did not choose to do so for two reasons. First of all, the modification in the equality definition leads to a rather unusual idea about equality of procedures: ignoring all side effects; and second, it carries us away too far from reality, i.e. the ILP embedding presented in the next chapter.

The only other point worth noticing is, that in order to proof point 5b, (for all m $\in$ .boldtag(S), a68to.tag(S)(m) = w => ltoa68(w) = m), the non ambiguity of the grammar is needed: Suppose PT_1 and PT_2 are two different parsetrees for the same string w, Let .tag(S)toa68(PT_1) = m1, and .tag(S)toa68(PT_2) = m2. Suppose that parser(w) = PT_1. Then, though a68to.tag(S)(m2) = w, ltoa68(w) $\neq$ m2. The non ambiguity of the grammar however guarantees that there is only one parsetree.

2.5. A parser generator

In this section we present the parser generator as it is included in the algol68 equivalent of the algorithm. The same notation is used. Some properties of the produced parser are:

- The parser expects the sequence of symbols to be parsed in a global variable subject. For simplicity we will assume that all terminal symbols of G consist of a single character, so that there is no need for a lexical scanner to produce subject.

- The parser is a simple recursive descent parser.
  With each terminal, nonterminal and (unique) right hand side, one parser procedure is associated. The heading of such a procedure looks like
  PROC parse_s = (INT index)STRUCT(INT postindex, REF PARSETREE parsetree):
  parse_s can succeed or fail. Success is recursively determined as follows:

  - If s is a terminal (and not the empty string), then parse_s succeeds if the next uncovered symbol in subject (pointed to by index) equals s. In that case parse_s covers s: the postindex field of the result equal index plus one. If s equals the empty string, parse_s always succeeds without covering anything.

  - If s is a nonterminal, parse_s succeeds if one of the procedures for its alternatives succeeds. parse_s covers the symbols covered by its successful descendant.

  - If s is a right hand side production, parse_s succeeds if all procedures for its members succeed. Index points to the symbol which is the first one to be covered by the first member of s; each next member begins to look where the previous one left of. Parse_s covers all symbols covered by its descendants.

  If parse_s succeeds, it returns a pointer to the first symbol it did

not cover, and a pointer to the PARSETREE it constructed for the symbols it did cover.
If parse_s fails, it returns the old index and a nil reference.

- The class of languages that can be parsed successfully by a recursive descent parser is not the complete class of context free languages. Moreover, the order in which alternatives appear in the grammar influences the behaviour of the parser.

The parser generator uses one additional procedure, starttest(N_i). It returns the empty string if $N_i \neq S$ and the string
" AND (index = UPB subject)" if $N_i = S$.

```
gen{MODE PARSETREE = STRUCT(STRING node,
                            FLEX[1:0] REF PARSETREE descendants);
    MODE PARSE     = STRUCT(INT postindex,
                            REF PARSETREE parsetree);
    STRING subject;
   }

# For each terminal which is not ε , a parser procedure is           #
# generated which succeeds if subject[index] equals that terminal,   #
# and fails otherwise. The procedure for  ε  always succeeds.        #

'for' i 'to' t
'do'
'if' T_i = ε 'then'
  gen{
     PROC parse.tag(ε) = (INT index) PARSE:
     (index,  HEAP PARSETREE := ("", ()) );
     }
```

```
'else'
  gen{
      PROC parse.tag($T_i$)toa68 = (INT index) PARSE:
      IF index > UPB subject THEN (index, NIL)
         ELSE IF subject[index] = ".$T_i$"
         THEN (index + 1, HEAP PARSETREE := (".$T_i$", ()) )
         ELSE (index, NIL)
      FI FI;
     }
'fi'
'od';

  # For each right hand side alternative consisting of more than one #
  # symbol, a parser procedure is generated which succeeds if it     #
  # finds subsequent pieces of subject which each match one symbol.  #

'for' i 'to' n
'do' 'for' j 'to' $m_i$
'do' 'if' $l_{ij}$ > 1 'and' 'not' 'old' $a_{ij}$ 'then'
gen{
    PROC parse.tag($a_{ij}$) = (INT index)PARSE:
    (INT nextindex := index;
     PARSE p;
     [.$l_{ij}$] REF PARSETREE descendants;
     k $\rightarrow$ $l_{ij}$:
     {p := parse.tag($E_{ijk}$)(nextindex);
      IF parsetree OF p :=: REF PARSETREE (NIL)
         THEN (index, NIL)
         ELSE  nextindex := postindex OF p;
               descendants[k] := parsetree OF p;
     }
     (nextindex, HEAP PARSETREE := (".$a_{ij}$", descendants))
     k $\rightarrow$ $l_{ij}$:
     {FI }
    );
   }
'fi' 'od';
```

```
# For each non terminal, a parser procedure is generated which      #
# tries each alternative in turn and succeeds as soon as an alter- #
# native succeeds. The procedure for S also checks, if the whole    #
# of subject is matched.                                            #
gen{
    PROC parse.tag($N_i$) = (INT index) PARSE:
    (PARSE p;
     j → $m_i$:
     {p := parse.tag($a_{ij}$)(index);
      IF (parsetree OF p :/=: REF PARSETREE(NIL))    .starttest($N_i$)
         THEN
             (postindex OF p, HEAP PARSETREE := (".$N_i$", parsetree OF p))
         ELSE
     }
     (index, NIL)
     j → $m_i$
     {FI }
    );
   }
'od';

# Finally, we generate a procedure parser which provides the correct  #
# interface between the parser generated here and the call by ltoa68 #
# provided that terminal symbols of G consist of one character.       #

gen{
    PROC parser = (STRING w) REF PARSETREE:
    (subject := w;
     parsetree OF parse.tag(S)(1)
    );
   }
```

3. Algol68G-0: The embedding of ILP

3.1. From method to application

We already mentioned in the introduction, that the embedding technique described in the previous chapter was developed for one special purpose: the production of an Algol68 library prelude which is equivalent (in a well defined sense) with the intermediate graphics language ILP.

This chapter is concerned with that special prelude. We will extensively refer to the defining document of ILP [2]; without at least some knowledge of its contents this chapter cannot be understood. An overview of ILP is given in [4].

The Algol68 extension defined by the prelude will be called Algol68G-0. It forms only the basic layer of the high level graphics language Algol68G as envisaged; it provides the interface between ILP and Algol68 without as yet offering any operations on graphical objects.

The G-0 prelude is not identical to the set of declarations produced by the embedding generator fed with the ILP syntax as given in [2]. First of all, we made a number of systematic changes in order to make the prelude more efficient and more user friendly. Secondly, we added a set of operators which check whether certain value restrictions on the graphical modes (i.e. the modes which are declared as part of the prelude) are satisfied. These restrictions cover the larger part of the non context free features of ILP. Thirdly, we added procedures to declare detectors and generators.

The G-0 prelude consists of seven different modules. Figure 2 shows these modules and the way they are related; an arrow pointing from one module to another means that the second uses the first.

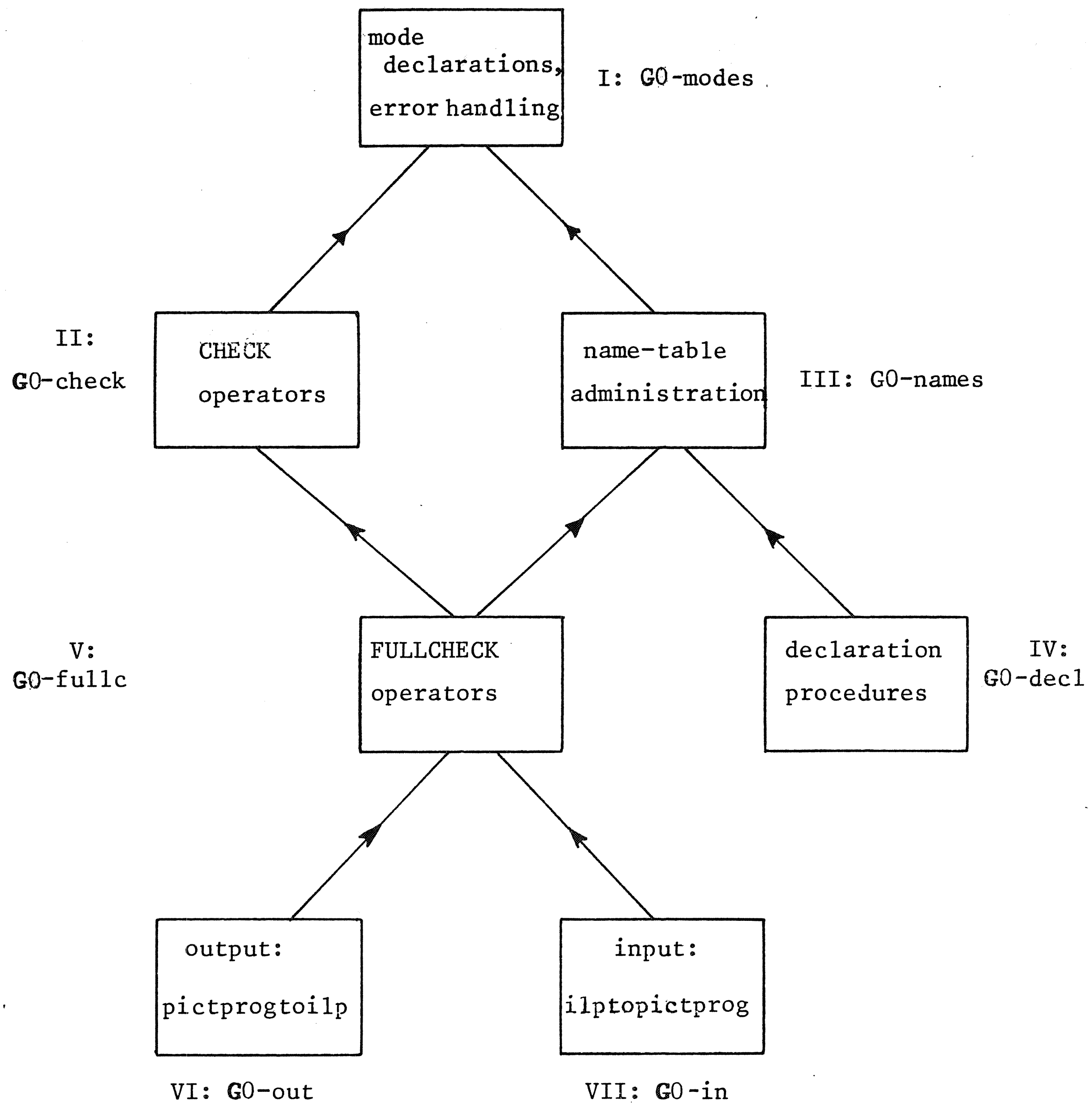

figure 2. The structure of the G-0 prelude

Section 3.2.1 through 3.2.5 contains additional comment on the structure of these modules. We do not list the complete prelude.

Section 3.3. describes Algol68G-0, the language defined by the prelude. It lists all mode declarations, and provides information about check operators and available procedures.

Section 3.4. contains some more general comment and some plans for the future.

3.2. Algol68G-0: The implementation

3.2.1. The mode declarations

In 3.3.1 the mode declarations incorporated in the G-0 prelude are listed, together with the context free grammar on which they are based. This grammar is not identical to the ILP syntax as given in [2], though it does produce exactly the same language, as can be verified easily.

The use of a recursive descent parser required the removal of left recursion from the ILP grammar; furthermore we removed ambiguities and did some "factoring" on rules.

We also made a number of systematic changes to the mode declarations generated for this grammar, in order to make the prelude more efficient and easier to use. As a result of these changes, nearly all mode declarations for single terminals (with the exception of NIL, TYPFAULT, PENFAULT, DOT and UNIT) could be deleted. The changes fall into six classes:

1 The automatically generated names are changed to more appropriate ones at many points.

2 When a structured mode contains at least one field corresponding to a nonterminal, and one or more fields corresponding to terminals, the latter are removed. This does not affect the number of values accepted by the mode and by choosing appropriate tags for the remaining fields, the uniqueness of the mode can always be guaranteed. For example, the construct "WITH attribute DRAW picture" leads to a structured node which has only two fields, of mode REF ATTR (1) and

(1) In this chapter bold words are always written in CAPITALS.

REF PICTURE respectively.

3 In the original embedding, a rule of the form

$x\colon T_1 \mid T_2 \mid \ldots. \mid T_m$

with all T_i terminals, leads to a declaration of a united mode (say X) which accepts m values, one for each alternative. Instead of declaring such a united mode we identify X with the subrange of integers from 1 to m. This means we replace all applied mode indications of X by INT, and impose an additional value restriction on the affected modes. If m=2, we use BOOL instead of INT and then don't even need a value restriction.
For our own and the users convenience we declare some additional constants with mnemonic names, which associate such an integer (or boolean) value with some terminal.
Two example of declarations affected by this change are the declarations for MODE TYPO (p. 55) and MODE NPICT (p. 37)

4 The mode produced for the nonterminal "value" has lists of digits (possibly interleaved with a few special characters like +, -, . and e) as values. We replaced this mode by REAL or INT, (depending on value restrictions), thus cutting out a complete subtree and making a nice set of operators available.
Equally, the mode declarations for "properstring" and for "name" are deleted and all applied indications are replaced by STRING. The affected modes get an additional value restriction (no blanks and satisfying the identifier syntax respectively).

5 Syntax rules of the form "x: a x ¦ x" lead to declarations like

```
MODE X = UNION(AX, A);
MODE AX = STRUCT(REF A  a,
                 REF AX ax);
```

The mode AX has list-like structures as values, but they are not terminated by a NIL-reference. In most cases we replaced these lists by arrays; in the remaining cases we changed to a "normal" list structure, i.e. we omit the union and assume that the list is terminated by NIL.

6 We sometimes removed references in fields of structured modes.
All fields in structured modes originally contain a reference, included to guarantee that no modes can have values of infinite size. As explained in section 2.4., their presence has a drawback: they introduce additional values accepted by the mode which do not correspond to terminal productions. As a consequence we can no longer completely rely on the Algol68 mode mechanism to ensure that all values correspond to valid ILP constructs (that is, valid at least according to the context free syntax), and have to include an explicit check on the absence of fields which have value NIL. For all non recursive structured modes there is an alternative: omitting the reference. This however would greatly increase the size of values, and thus introduce an unacceptable amount of overhead when these values are passed as arguments to procedures. We decided to adopt the following strategy. References are not omitted if they point to values of a mode on which value restrictions are imposed. We can then attach a special meaning to nil-references, they replace incorrect values. We will return to this point in the next section.

3.2.2. <u>Context sensitivity: CHECK and FULLCHECK</u>

ILP is not a context free language, which is another reason why we cannot fully out our plan to use the Algol68 mode mechanism to ensure that any sequence of operations on an ILP program will produce another valid ILP program.

Most of the non context free features of ILP are of the type "the number of columns in an affine matrix must equal the number of rows plus one", and can in Algol68G-0 be expressed as value restrictions on the graphical modes.

The G-0 prelude contains operators to check these restrictions. These operators act as a filter, their operand is a pointer to a graphical object. If the object is well formed, the pointer is returned unchanged; if not, the NIL-pointer is returned.

There are two sets of operators. All operators in the first set are called CHECK, (1) They check exactly one level of value restrictions; non-NIL fields in structures are assumed to be well formed. For example, CHECK on the mode corresponding to "pictures" checks whether all pictures in such a list have the same dimension, but does not check whether the pictures themselves are well formed. There is a CHECK operator for each mode with imposed value restrictions.

If a CHECK operator finds an error, it issues some appropriate message and sets an errorflag. Execution of the Algol68G program is not terminated, but no more ILP output will be produced.

(Error messages are problematic by the way. We would like to associate with such a message the line number in the particular program from which the most recent call to a prelude procedure or operator occurred; but there seems to be no way to retrieve this information. We provide a substitute which may help a user to locate the offensive point in his program: the error handling procedure also prints the value of a global variable which can be set from the user program to an integer value by a call to a procedure "l" or to a string value by a call to a procedure "sl". Its initial value is the empty string.)

In future versions of Algol68G, operators which deliver a value of a mode on which a value restriction is imposed, should always let this value pass through a CHECK filter before it is returned. In that way, all components of other modes will indeed be either NIL or well formed.

Members of the second operator set are called FULLCHECK. These operators have two tasks: they check value restrictions on all levels (using the CHECK operators), and they check those non context free features of ILP which cannot be expressed as value restrictions, namely:
- All names of named pictures should be different,

---

(1) Algol68 allows definition of different operators with the same name (in this case CHECK), as long as these operators can be differentiated according to the modes of their argument.

- All names of attribute packs should be different,
- External references must be valid.

Like the CHECK operators, the FULLCHECK operators return their argument unchanged if it is well formed, and NIL if it is not.

The set of modes with which a FULLCHECK operator is associated can be determined (iteratively) as follows:

- A FULLCHECK operator is defined for all modes which contain external references at the top level;

- A FULLCHECK operator is defined for a united mode if it has at least one member which has an associated CHECK or FULLCHECK operator.

- A FULLCHECK operator is defined on a structured mode if it has at least one field which has an associated CHECK or FULLCHECK operator, and on an array if a CHECK or FULLCHECK operator is associated with its elements.

Both the input and the output procedure use the FULLCHECK set to assure that no incorrect ILP programs are accepted or produced.

3.2.3. <u>Names and declarations</u>

Detectors (other than the common detector), symbols, curves and templates must be declared. Procedures "decldetectors", "declsymb", "declcurves" and "decltemplates" are provided for this purpose. For curves and templates additional information like the number and type of parameters must be specified as well. The four declaration procedures all contain a call to a boolean procedure, named "detavailable", "symavailable", "curveavailable" and "tempavailable" respectively. These procedures return TRUE if the declaration is acceptable and FALSE if it is not, in the last case the declaration procedure issues an error message. The body of these procedures depends on the (local) ILP implementation. (In the G-0 prelude running at this moment all "available" procedures return FALSE).

Names of named pictures and attribute packs need not be declared separately, FULLCHECK takes care of that. All names are gathered into one nametable. Associated with each name is its type (named picture, attribute pack, symbol, curve, template or detector) and in the cases of template and curve a descriptor for number and types of parameters. Names may occur more than once in the table, provided that the associated objects have different types; the ILP syntax is such that it can be determined immediately whether a name refers to a named picture, an attribute pack, a symbol, a curve, a template or a detector.

All names occurring in an input ILP program, are automatically declared by the input procedure.

3.2.4. The output procedure

In the original embedding a separate output procedure is associated with each mode declaration. However, this extra layer of procedures introduces a substantial overhead caused by argument transmission; we performed some inline expansion of such procedure calls to increase efficiency. Only the outermost procedure, "pictprogtoilp", is available to the user, all other procedures are declared within its body. First "pictprogtoilp" checks the errorflag; if that is set it immediately returns. Otherwise it performs a FULLCHECK on its argument, only if this FULLCHECK does not return the nil reference it produces the ILP version. The output is sent to a file, specified by the user; see 3.3.2.. An attempt is made to provide a reasonable lay-out.

3.2.5. The input procedure

Again, only one (the outermost) input procedure is available to the user, called "ilptopictprog". This procedure translates ILP programs to Algol68 objects in one pass; there are no separate phases as in the original embedding. ILP is LL(1); hence only one symbol at a time needs to be considered. A procedure "nextsymb" gets the next lexical unit from the input file, which is specified by the user; see 3.3.2.. The parser is a stupid one: If it finds an error, it issues one message and then terminates. No error recovery is provided. We have a reason to write such a stupid parser, as will be explained in section 3.4.1. If the parser completes translation of the ILP program, a FULLCHECK is performed on the object produced (this check can very well be seen as a second pass of the translator). Only if this FULLCHECK does not return the NIL-reference, execution of the Algol68G program continues.

3.3. Algol68G-0: The language extension

In this section the Algol68G-0 extension is defined. Some material has been covered more extensively in other parts of this report, but is repeated here to make this section self contained.

The G-0 definition is split into two parts:

- Specification of the graphical modes and associated operators. The only operators provided by the G-0 extension are operators which check whether values of the graphical modes are well formed, i.e. whether they correspond to valid ILP constructs.

- Specification of the (few) available procedures.

3.3.1. Mode declarations

The kernel of Algol68G consists of a set of graphical modes which are closely related to the ILP syntax. In this section the syntax and all associated mode declarations are listed.

The syntax rules are given in BNF. Non-terminals consist of lower case letters and underlines. The non-terminal that is defined in a rule is separated by a colon (:). Alternatives are separated by a vertical bar (¦). The end of a rule is marked with the symbol $. A terminal is either a special single character from the following list:

```
( ) { } , . ; [ ]
```

or it is a delimiter denoted in CAPITALS. There are no syntax rules for the nonterminals "value", "name" and "proper_string". They produce a denotation for a number, an identifier (first character a letter, the others letters or digits), and a string over some (unspecified) alphabet. "value" is mapped to REAL or INTEGER, "name" and "properstring" are mapped to STRING.

Each mode declaration is listed next to the corresponding syntax rule. A declaration can be followed by some additional entries, according to the format

```
MODE X = .....;

       Remark:
       CHECK:
       FULLCHECK
       mnemonics:
```

- A remark entry is present only when it might not be clear to which ILP construct a declaration refers, or when it is not formed according to the rules laid down in the previous sections.

- Presence of a CHECK entry followed by one or more predicates, means that the prelude contains an operator declaration with the heading

  ```
  OP CHECK = (REF X x)REF X:
  ```

  This operator tests whether its operand (at least if it is non-nil) satisfies the predicates listed in the entry. If that is the case, the operand is returned unchanged, but if not, the nil-reference is returned. One predicate occurs so frequently that we have introduced an abbreviation: (*) means that all fields/elements/members of X marked with an asterisk should be non-nil.

- Presence of a FULLCHECK entry means that the prelude contains an operator declaration with the heading

```
OP FULLCHECK = (REF X x)REF X:
```

If we consider a value x of mode X as a tree with root r, then CHECK tests properties of and relationships between immediate descendants of r; a CHECK operator never contains a call to another CHECK operator. FULLCHECK tests whether value restrictions are satisfied at all levels in the tree, a FULLCHECK operator always contains at least one reference to another FULLCHECK or CHECK operator. A FULLCHECK operator tests all predicates tested by all CHECK operators defined for submodes of X, and in addition some relationships between different values of one mode: whether all names are unique, and whether named pictures and attribute packs don't contain recursive references.
If p is a non-nil reference to an object of mode PICTPROG, and FULLCHECK p = p, then p corresponds to a syntactically correct ILP program.

- A mnemonics entry contains some additional declarations associated with integer and boolean fields in structures, of which each value corresponds to one ILP terminal.

```
picture_program:        pictstruct ¦
                        pictstruct picture_program $



pictstruct:             named_picture ¦
                        attribute_pack $

named_picture:          rootsub dimension  name picture . $

rootsub:                PICT ¦
                        SUBPICT $

dimension:              DIMLESS ¦
                        dim $

dim:                    ( value  ) ¦
                        empty $




attribute_pack:         ATTR dimension  name attribute  . $





picture:                name ¦
                        picture_element ¦
                        { pictures } ¦
                        subspace_pict ¦
                        withdraw_node $
```

```
MODE PICTPROG = STRUCT (REF PICTSTRUCT pictstruct,  (*)
                        REF PICTPROG   next        );

   - CHECK: (*).
   - FULLCHECK.

MODE PICTSTRUCT = UNION (NPICT, ATTRPACK);

   - FULLCHECK.

MODE NPICT = STRUCT (BOOL root,
                     INT dim,
                     STRING pname,
                     REF PICTURE pict    (*)
                    );

   - CHECK:
       dim >= 0,
     & pname should be an identifier,
     & the dimension of pict equals dim,
     & (*).
   - FULLCHECK:
       no other npict with the same pname.
   - mnemonics:
       BOOL troot = TRUE,
            tsub  = FALSE;
       INT  dimless = 0;

MODE ATTRPACK = STRUCT (INT dim,
                        STRING aname,
                        REF ATTR attr   (*)
                       );

   - CHECK:
       dim >= 0,
     & aname should be an identifier,
     & the dimension of attr equals dim.
   - FULLCHECK.
       no other attrpack with the same aname,

MODE PICTURE = UNION (REF NPICT,    (*)
                      PICTEL,
                      PLIST,
                      SUBSPP,
                      WDNODE    );

   - Remark:
     Rather than mapping the alternative "pname" in the original
     syntax to STRING and adding a value restriction "if a value
     of mode PICTURE is acceptable to STRING, it must be the
     name of some named picture", we map it to REF NPICT:
     Algol68 now takes care of the value restriction!

   - CHECK (*).
   - FULLCHECK.
```

```
pictures:                picture  ¦
                         picture pictures $



withdraw_node:           WITH attribute DRAW picture $



picture_element:         coordinate_type ¦
                         text ¦
                         generator ¦
                         NIL $



coordinate_type:         type attribute_matches ( coordinates ) $



type:                    POINT ¦
                         LINE ¦
                         CONTOUR $

coordinates:             coordinate ¦
                         coordinate , coordinates $
```

```
MODE PLIST = STRUCT (REF PICTURE pict,    (*)
                     REF PLIST   next);

   - CHECK:
       all pictures which do not have dimension 0 are of the same
       dimension,
     & (*).
   - FULLCHECK.

MODE WDNODE = STRUCT (REF ATTR      attr,    (*)
                      REF PICTURE   pict     (*)
                     );

   - CHECK:
       attr and pict are of the same dimension, unless one of them has
       dimension 0,
     & (*).
   - FULLCHECK.

MODE PICTEL = UNION (PLC,
                     TEXT,
                     GENERATOR,
                     NULL);

   - FULLCHECK.

MODE NULL = STRUCT(BOOL null);

   - mnemonic:
       NULL null;
   - remark:
       NULL is the mode invented for NIL.
       It is not necessary to assign a value to the null-field of null;
       this field is never inspected by the prelude.

MODE PLC = STRUCT (INT type,
                   REF MATCHES matches,     (*)
                   REF COORDS coords        (*)
                  );

   - CHECK:
       0 <= plc <= 3,
     & (*).
   - FULLCHECK.
   - mnemonics:
       INT tpoint   = 1,
           tline    = 2,
           tcontour = 3;

MODE COORDS = FLEX [1:0] REF COORD;      (*)

   - CHECK:
       all elements in the array are of the same dimension,
     & the array contains at least one element,
     & (*).
   - FULLCHECK.
```

```
coordinate:             attribute_matches coordinate_value ¦
                        attribute_matches ( coordinate_values ) $


coordinate_values:      coordinate_value ¦
                        coordinate_value , coordinate_values $


coordinate_value:       dimensional_value ¦
                        PP ¦
                        EP $


dimensional_value:      [ values ] $

matrix_value:           [ dimensional_values ] $

   - Remark:
       There are no declarations corresponding to the next two rules.

dimensional_values:     dimensional_value ¦
                        dimensional_value , dimensional_values $

values:                 value ¦
                        value , values$

subspace_pict:          SUBSPACE dim new_axes picture $

new_axes:               position ( shift axes ) $

shift:                  dimensional_value $

position:               CURRENT ¦
                        ORIGIN $

axes:                   , dimensional_values ¦
                        empty $
```

```
MODE COORD = STRUCT (REF MATCHES matches,     (*)
                     REF COVALS covals        (*)
                    );
   - CHECK: (*)
   - FULLCHECK.

MODE COVALS = FLEX [1:0] COVAL;

   - CHECK:
       all elements in the array which are accepted by dimval, are
       of the same dimension,
     & the array contains at least one element.

MODE COVAL = UNION (DIMVAL, POS);

MODE POS = STRUCT (BOOL pos);

   - mnemonics:
       POS pp = (POS scr; pos OF scr := TRUE ; scr),
           ep = (POS scr; pos OF scr := FALSE; scr);

MODE DIMVAL = FLEX [1:0] REAL;

MODE MATVAL = FLEX [1:0,1:0] REAL;

      Remark:
      A row and not a column of a matrix value corresponds to
      a dimensional value. This choice is made because it allows one
      to write a whole matrix as a display of dimensional values:
      If m = ((1,2,3),(4,5,6)) then m[1,2] = 2 and not 4.



MODE SUBSPP = STRUCT (INT dsub, dsur,
                      BOOL position,
                      DIMVAL shift,
                      FLEX [1:0] DIMVAL axes,
                      REF PICTURE pict        (*)
                     );

   - Remark:
       This structure has an extra field: the integer dsub, denoting
       the dimension of the surrounding space.
   - CHECK:
       dsub <= dsur,
     & the dimension of pict equals dsub,
     & the number of elements in axes equals either 0 or dsub,
     & all elements in axes have dimension dsur,
     & shift has dimension dsur,
     & (*).
   - FULLCHECK,
   - mnemonics:
       BOOL current = TRUE,
            origin  = FALSE;
```

```
generator:               symbol |
                         curve |
                         template $


symbol:                  SYMBOL names $

names:                   name |
                         name , names $


curve:                   CURVE type attribute_matches
                                     ( curve_generators ) $






curve_generators:        curve_generator |
                         curve_generator , curve_generators $



curve_generator:         attribute_matches curve_determinator |
                         attribute_matches ( curve_determinators ) $



curve_determinators:     curve_determinator |
                         curve_determinator , curve_determinators $



curve_determinator:      name |
                         name ( interval curve_parameters ) $

curve_parameters:        curve_parameter |
                         curve_parameter , curve_parameters $
```

```
MODE GENERATOR = UNION (SYMBOL,
                        CURVE,
                        TEMPLATE);

   - FULLCHECK.

MODE SYMBOL = STRUCT (FLEX [1:0] STRING snames);

   - CHECK:
       All elements of symbols satisfy the identifier syntax.
   - FULLCHECK:
       All elements of symbols are declared as symbol names.

MODE CURVE = STRUCT (INT type,
                     REF MATCHES matches,     (*)
                     REF CURGENS curgens     (*)
                    );

   - CHECK:
       1 <= type <= 3,
     & (*).
   - FULLCHECK.
   - mnemonics: see PLC.

MODE CURGENS = FLEX [1:0] REF CURGEN;     (*)

   - CHECK:
       the array contains at least one element,
     & (*).
   - FULLCHECK.

MODE CURGEN = STRUCT (REF MATCHES matches,    (*)
                      REF CURDETS curdets     (*)
                     );

   - CHECK: (*)
   - FULLCHECK.

MODE CURDETS = FLEX [1:0] REF CURDET;

  - CHECK:
      the array contains at least one element,
    & (*).
  - FULLCHECK.

MODE CURDET = STRUCT (STRING cname,
                      REF INTERVAL interval,      (*)
                      FLEX [1:0] CURPAR parameters
                     );

   - CHECK:
       cname satisfies the identifier syntax,
     & (*),
   - FULLCHECK:
       cname is declared as a curve determinator,
     & iv and params are as specified by the associated descriptor;
       (see PROC declcurves in the next subsection).
```

```
curve_parameter:        value |
                        dimensional_value $

interval:               UNIT , |
                        ( value , value ) , |
                        empty $





template:               TEMPLATE ( template_generators ) $

template_generators:    template_generator |
                        template_generator , template_generators $


template_generator:     name |
                        name ( template_parameters ) $

template_parameters:    template_parameter |
                        template_parameter , template_parameters $
```

```
MODE CURPAR = UNION (REAL, DIMVAL);


MODE INTERVAL = UNION (UNIT,
                       []REAL,
                       QVOID
                      );
```

- CHECK:
  If an interval i   [] REAL, then LWB i = 1 and UPB i = 2.

```
MODE UNIT = STRUCT (BOOL unit);
```

- mnemonic:
  UNIT unit;

```
MODE QVOID = STRUCT (BOOL qempty);
```

- Remark:
  QVOID is the mode invented for ε.
  We could have used VOID instead; but not all available Algol68 compilers allow VOID as a member in a union. The CDC compiler, which we used for implementation of the prelude, for example does not.
- mnemonic:
  QVOID qempty;

```
MODE TEMPLATE = FLEX [1:0] REF TEMPGEN;
```

- CHECK:
  the array contains at least one element,
  & (*).
- FULLCHECK.

```
MODE TEMPGEN = STRUCT (STRING tname,
                       FLEX [1:0] TEMPPAR parameters
                      );
```

- CHECK:
  tname satisfies the identifier syntax,
  elements of parameters which are accepted by REF NPICT or REF ATTRPACK are non-NIL.
- FULLCHECK:
  tname is declared as a template name,
  & parameters are as specified by the associated descriptor (see the entry for PROC decltemplates in the next section).

```
template_parameter:      value ¦
                         dimensional_value ¦
                         name $


text:                    TEXT attribute_matches ( strings )


strings:                 string ¦
                         string , strings $


string:                  attribute_matches proper_string ¦
                         attribute_matches ( proper_strings ) $


proper_strings:          proper_string ¦
                         proper_string , proper_strings $
```

```
MODE TEMPPAR = UNION (REAL,
                      DIMVAL,
                      REF NPICT,    (*)
                      REF ATTRPACK, (*)
                      STRING
                     );
```

- Remark:
  See the remark with the declaration for PICTURE concerning the members REF NPICT and REF ATTRPACK.
  The same treatment of the alternative "dname" is not possible; unlike pnames (which may occur in just one NPICT) and anames (which may occur in just one ATTRPACK), a dname (i.e. a reference to some detector) can occur in more than one value of the mode DETECT.

```
MODE TEXT = STRUCT (REF MATCHES matches,     (*)
                    REF MSTRINGS strings     (*)
                   );
```

- CHECK: (*)
- FULLCHECK.

```
MODE MSTRINGS = FLEX [1:0] REF MSTRING;       (*)
```

- CHECK:
  the array contains at least one element,
  & (*).
- FULLCHECK.

```
MODE MSTRING = STRUCT (REF MATCHES matches,    (*)
                       REF PSTRINGS pstrings   (*)
                      );
```

- CHECK: (*)
- FULLCHECK.

```
MODE PSTRINGS = STRUCT (FLEX [1:0] STRING pstring)
```

- CHECK:
  the character blank does not occur in any element of the array,
  & the array contains at least one element.

```
MODE MATCHES = FLEX [1:0] REF MATCH;     (*)
```

- CHECK:
  No two matches have the same match field,
  & (*).
- FULLCHECK.

```
attribute_matches:       deny attribute_match attribute_matches |
                         empty $

deny:                    ~ |
                         NOT |
                         empty $



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

attribute:               prefixed_attribute |
                         basic_attribute $

prefixed_attribute:      prefix  basic_attribute $



prefix:                  ABS |
                         REL

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



attributes:              attribute |
                         attribute ; attributes $
```

```
MODE MATCH = STRUCT (BOOL deny,
                     INT match);

   - Remark:
     In objects of mode MATCH the alternative spellings "~" and
     "NOT" cannot be distinguished.
     deny = TRUE corresponds to the alternative "~" or "NOT",
     deny = FALSE corresponds to the alternative "empty".
   - CHECK:
     1 <= match <= 6.
   - mnemonics:
     HEAP MATCH tf := (TRUE, 1),
                dt := (TRUE, 2),
                st := (TRUE, 3),
                pn := (TRUE, 4),
                cm := (TRUE, 5),
                vs := (TRUE, 6);

MODE ATTR = UNION (PREFATTR, BASATRR);

   - FULLCHECK.

MODE PREFATTR = STRUCT (BOOL pref,
                        REF BASATTR attr   (*)
                       );

   - CHECK: (*)
   - FULLCHECK.
   - mnemonics:
     BOOL tabs = TRUE,
          trel = FALSE;

MODE BASATTR = UNION (ATTRCLASS,
                      REF ATTRPACK,    (*)
                      ALIST,
                      NULL);

   - Remark:
     The same observation which led to a member REF NPICT in the
     declaration for PICTURE, led to a member REF ATTRPACK here.
   - CHECK: (*)
   - FULLCHECK.

MODE ALIST = STRUCT (REF ATTR  attr,    (*)
                     REF ALIST next
                    );

   - CHECK:
     All attributes which don't have dimension 0 have the
     same dimension,
    & (*).


   - FULLCHECK.
```

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


transformation:         rotate |
                        transcale |
                        matrix |
                        projection |
                        port $

rotate:                 ROTATE value AROUND invariant $

invariant:              ( dimensional_values ) $




transcale:              tras dimensional_value $


tras:                   TRANSLATE |
                        SCALE $

matrix:                 matrix_type matrix_value $




matrix_type:            MATRIX |
                        AFFINE |
                        HOMMATRIX $
```

```
MODE ATTRCLASS = UNION (TRANSFORM,
                        DETECT,
                        STYLE,
                        CONTROL,
                        PEN,
                        CMODE,
                        VIS
                       );

   - FULLCHECK.

MODE TRANSFORM = UNION (ROT,
                        TRANSCALE,
                        MATRIX,
                        PROJECT,
                        PORT    );

MODE ROT = STRUCT (REAL angle,
                   FLEX [1:0] DIMVAL axes
                  );

   - Remark:
       Angle is supposed to be in radians and not in degrees!
   - CHECK:
       All elements in axes have the same dimension,
     & the number of elements in axes is one less than that dimension.

MODE TRANSCALE = STRUCT (BOOL tras,
                         DIMVAL dim);

   - mnemonics:
     BOOL ttrans = TRUE,
          tscale = FALSE;

MODE MATRIX = STRUCT (INT type,
                      MATVAL matval);

   - CHECK:
       (1 <= type <= 3).
     & if type = 1 or type = 3, matval is square,
       if type = 2, matval should have one more rows than
       columns, i.e. UPB m = (2 UPB m) + 1.
   - mnemonics:
       INT tmatrix = 1,
           taffine = 2,
           thommat = 3;
```

```
projection:             PROJECT par dimensional_value
                                ON or dimensional_value $

par:                    PARALLEL ¦
                        empty $

or:                     ORIGIN ¦
                        empty $


port:                   window viewp $





window:                 WINDOW ( dimensional_value ,
                                           dimensional_value ) $


viewp:                  ; viewport ¦
                        empty


viewport:               VIEWPORT ( dimensional_value ,
                                              dimensional_value ) $


style:                  linestyle ¦
                        pointstyle ¦
                        typographic $


linestyle:              period ¦
                        map ¦
                        thick $


period:                 PERIOD ( period_decription ) $

period_description:     dash ¦
                        dash , gap ¦
                        dash , gap , dash $
```

```
MODE PROJECT = STRUCT (BOOL par,
                       DIMVAL eye,
                       BOOL or,
                       DIMVAL space);

   - Remark:
       par = FALSE corresponds to the alternative empty;
       or  = FALSE corresponds to the alternative empty.
   - CHECK:
       The dimension of eye equals the dimension of space;

MODE PORT = STRUCT (REF WINDOW window,    (*)
                    REF VIEWP viewport    (*)
                   );

   - CHECK:
       Window and viewport must have the same dimension,
     & (*).
   - FULLCHECK.

MODE WINDOW = STRUCT (DIMVAL  w1, w2);

   - CHECK:
       w1 and w2 must have the same dimension.

MODE VIEWP = UNION (QVOID, VIEWPORT);

   - FULLCHECK.

MODE VIEWPORT = STRUCT (DIMVAL v1, v2);

   - CHECK:
       v1 and v2 must have the same dimension.

MODE STYLE = UNION (LINEST,
                    POINTST,
                    TYPOST);

   - FULLCHECK.

MODE LINEST = UNION (PERIOD,
                     MAP,
                     THICK);

   - FULLCHECK.

MODE PERIOD = STRUCT (REF DASH1 dash1,   (*)
                      REF GAP gap,       (*)
                      REF DASH2 dash2    (*)
                     );

   - CHECK:
       if gap is of mode VOID, then so is dash2.
     & the sum of fields of mode INTEGER is <= 100,
     & (*).
   - FULLCHECK.
```

dash:	DOT ¦ value $
gap:	value $
map:	MAP (value reset) $
reset:	RESETCOORDINATE ¦ CONTINUE ¦ RESETLINE $
thick:	THICK (value)
pen:	PENFAULT ¦ contrast ¦ intens ¦ colour $
contrast:	CONTRAST (value , value) $
intens:	INTENS (value) $

```
MODE DASH1 = UNION (DOT, INT);

   - CHECK:
       If d ∈ INTEGER, d >= 0.

MODE DASH2 = UNION(DOT, INT, QVOID);

   - CHECK:
       If d ∈ INTEGER, d >= 0.

MODE DOT = STRUCT (BOOL dot);

   - mnemonic:
       DOT dot;

MODE GAP = UNION (INT, QVOID);

   - CHECK:
       If g ∈ INTEGER, g >= 0.

MODE MAP = STRUCT (REAL value,
                   INT  reset);

   - CHECK:
       value > 0,
     & (1 <= reset <= 3).
   - mnemonics:
       INT tresetco   = 1,
           tcontinue  = 2,
           tresetline = 3;

MODE THICK = STRUCT (REAL thick);

   - CHECK:
       thick >= 0.

MODE PEN = UNION (PENFAULT,
                  CONTRAST,
                  INTENS,
                  COLOUR   );

   - FULLCHECK.

MODE PENFAULT = STRUCT(BOOL penfault);

MODE CONTRAST = STRUCT (REAL c1, c2);

   - CHECK:
       0 <= c1 <= c2 <= 100.

MODE INTENS = STRUCT (REAL i);

   - CHECK:
       0 <= i <= 100.
```

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


typographic:            TYPFAULT ¦
                        typo ( value ) $


typo:                   FONT ¦
                        SIZE ¦
                        ITALIC ¦
                        BOLD $

pointstyle:             DOT ¦
                        POINTSTYLE typographic ¦
                        POINTSTYLE marker $


marker:                 "any non blank character " $


control:                MACHINEDEPENDENTCONTROL proper_string $


coordinate_mode:        FIXED ¦
                        FREE $
```

```
MODE COLOUR = STRUCT (REAL yellow,
                           blue,
                           red    );

   - CHECK:
       yellow >= 0 & blue >= 0 & red >= 0.

MODE TYPOST = UNION (TYPFAULT, TYP);

   - FULLCHECK.

MODE TYPFAULT = STRUCT(BOOL typfault);

MODE TYP = STRUCT (INT typtyp,
                   REAL val  );

   - CHECK:
       1 <= typtyp <= 4.
   - mnemonics:
       INT tfont   = 1,
           tsize   = 2,
           titalic = 3,
           tbold   = 4;

MODE POINTST = UNION (DOT,
                      PTYPO,
                      MARKER );

   - FULLCHECK.

MODE PTYPO = STRUCT (REF TYPOST ptypo  (*) );

   - CHECK: (*)
   - FULLCHECK.

MODE MARKER = STRUCT (CHAR marker);

   - CHECK:
       marker ≠ blank;

MODE CONTROL = STRUCT (STRING control);

   - CHECK:
       control does not contain the character blank.

MODE CMODE = STRUCT (BOOL mode);

   - mnemonics:
       BOOL fix  = TRUE,
            free = FALSE;
```

```
visibility:               VISIBLE ¦
                          INVISIBLE $



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

detector:                 name ¦
                          empty $
```

```
MODE VIS = STRUCT (BOOL visible);

   - mnemonics:
       BOOL tvisible   = TRUE,
            tinvisible = FALSE;

MODE DETECT = STRUCT (INT type,
                      STRING dname, pstring);

   - Remark:
       There is a small deviation here: we in fact add a field for
       the UNDETECT case. A value restriction is added stating that
       for type = undetect, pstring = "".
   - CHECK:
       (1 <= dtype <= 3),
     & dname is either the empty string or satisfies the identifier syntax,
     & pstring should not contain a blank,
     & if dtype = 3 then pstring should be the empty string.
   - FULLCHECK:
       dname is either the empty string (common detector) or is
       declared as a detector.
   - mnemonics:
       INT tdetect   = 1,
           tsetdel   = 2,
           tundetect = 3;
```

3.3.2. G-0 procedures

Procedures to declare externals

Declaration of externals, i.e. symbols, curves, templates and detectors is obligatory. Testing whether all externals and detectors are declared indeed is left to the FULLCHECK operators.

```
PROC declsymbols = ([] STRING sn) VOID:
```

Declsymbols declares all strings in its argument as symbols.
Diagnostics:
"sname not an identifier",
"sname redeclared",
"unavailable external reference".
(i.e. the symbol is known not to be in the available ILP library).

```
PROC declcurves = ([] STRUCT (STRING cname,
                              BOOL   iv,
                              [] INT params) cs) VOID:
```

Declcurves declares each element in its argument as a curve determinator. The argument cs must be interpreted as follows:

- cname OF cs[i] is the name of the i-th curve determinator declared;

- iv OF cs[i] = TRUE denotes that the i-th curve is a parameter curve; iv OF cs[i] = FALSE denotes that it is not.

- params OF cs[i] denotes number and dimension of the parameters of the i-th curve determinator:
  There are UPB params OF cs[i] parameters. If iv OF cs[i] = TRUE, (i.e. the i-th determinator belongs to a parameter curve), there must be at least one parameter, but if it is not, parameters are not necessarily absent. This is rather confusing; two different

meanings of the word "parameter" are involved, see 3.5.3.2.1 of [2] for an explanation.
The dimension of the j-th parameter is (params OF cs[i])[j].
Curve parameters are either dimensional values (params[j] >= 2) or real numbers (params[j] = 1).

Diagnostics:
"cname not an identifier",
"cname redeclared",
"parameter description index out of range" (i.e. < 0).
"parameter curve without parameters not allowed",
"unavailable external reference".

An example:
The call

```
declcurves((("circle", FALSE, (1,2)),
           ("parabola", TRUE, (1,1,1))
          ))
```

declares "circle" as a non parameter curve (i.e. without interval specification), with one parameter of dimension one and one of dimension two (probably radius and centre), and "parabola" as a parameter curve with three real numbers as parameters.
After the declarations

```
REAL a = 1, b = -1, c = 5, r = 2;
DIMVAL m = (2,2);

CURDET par := ("parabola", unit, (a,b,c)),
       cir := ("circle", qempty, (r, m));
```

cur and par are well formed values of the mode CURDET.

```
PROC decltemplates = ([] STRUCT (STRING tname,
                                 [] INT  params) ts) VOID:
```

Decltemplates declares each element in its argument as a template generator. The argument ts must be interpreted as follows:

- tname OF ts[i] is the name of the i-th template generator declared.

- params OF ts[i] denotes number and types of parameters of the i-th template generator. A template parameter can be a reference to a named picture (type denoted by -2), a reference to an attribute pack (type denoted by -1), the name of a detector (type denoted by 0), a real number (type denoted by 1) or a dimensional value of dimension n (type denoted by n).
  Mnemonics for these types are declared as well:
  ```
  INT npicttype    = -2,
      attrpacktype = -1,
      dnametype    =  0;
  ```

Diagnostics:
   "tname not an identifier",
   "tname redeclared",
   "parameter description index out of range" (i,e. < -2).
   "unavailable external reference".

An example:
The call
```
decltemplates((("square", (1, npicttype)),
               ("wdnode", (attrpacktype, npicttype)),
               ("enable", (dnametype, npicttype))
             ))
```

declares "square" as a template expecting a number and a reference to a named picture as parameters, "wdnode" as a template expecting references to an attribute pack and a named picture as parameters, and "enable" as a template expecting a detector and a reference to a named picture as parameters.

Output

```
PROC pictprogtoilp = (REF PICTPROG pictprog,
                      UNION (REF FILE, FILDES) file) VOID:
```

where

```
MODE FILDES = STRUCT (STRING idf,
                      INT p, l, c);
```

Pictprogtoilp checks whether its first argument "pictprog" corresponds to a syntactically correct ILP program and if so, produces the ILP equivalent on the file denoted by its second argument "file".

Pictprogtoilp writes to a file named "outfile". In case "file" is of the mode FILE, pictprogtoilp assumes that it refers to a file already opened for writing: "outfile := file". In case "file" is of mode FILDES, pictprogtoilp opens "outfile" for writing by means of the call

```
"establish(outfile, idf OF file, stand out channel,
           p OF file, l OF file, c OF file     )"
```

(i.e. outfile is established with external name idf, with p pages, l lines per page and c characters per line).

Two integer variables are associated with pictprogtoilp:
INT prc denotes the number of decimals put out for real numbers and is initialized to 2; INT maxindent is a lay out parameter: it denotes the maximum indentation allowed on the output file. It is initialised to 40, but changed to ENTIER ((c OF file) / 2) whenever pictprogtoilp establishes an output file.

There are many diagnostics, all related to violations of the value restrictions on PICTPROG as listed in the previous section.

Input

```
PROC ilptopictprog = (UNION (REF FILE, FILDES) file) REF PICTPROG:
```

Ilptopictprog expects an ILP program on the file denoted by its argument. It reads from REF FILE "infile". In case "file" is of mode REF FILE, ilptopictprog assumes that it refers to a file opened for reading: "infile := file". In case "file" is of mode FILDES (see for the declaration the previous entry output), ilptopictprog opens infile for reading by means of the call

```
"establish(infile, idf OF file, stand in channel,
           p OF file, l OF file, c OF file   )".
```

If the program on infile is not syntactically correct, ilptopictprog will issue a message and terminate execution of the Algol68G program. No error recovery is provided; see 3.4.1.

l and sl

```
PROC l = (INT    i) VOID   and
PROC sl = (STRING s) VOID
```

assign their argument to a global variable "label". Whenever an error message is issued by the G-O prelude, the current value of "label" is printed as well; hence "s" and "sl" can be used to locate offensive calls. We are not very enthusiastic about this facility.

3.4. Discussion

The design of the G-0 prelude as presented in this report is based on four a priori decisions. In chronological order, these decisions were:

1 To design an intermediate graphics language (ILP) and use that as a uniform interface between all modules of a graphics system. As a consequence of this decision, a high level graphics language is defined in terms of operations on ILP programs, and implemented as an extension of an existing language. This decision is by far the most fundamental; it cannot be changed without changing the complete set up of the graphics project.

2 To use Algol68 as a host language,

3 To implement the interface as a library prelude, centered around a set of declarations for graphical modes.

4 To construct the G-0 prelude in the way described in this report, stressing some issues (equivalence with ILP, safety) and disregarding others (efficiency).

In this concluding section we want to give some comment on these decisions. At this moment such comment is necessarily perfunctory, real conclusions cannot be drawn until considerable experience has been gained with a working graphics system.

We will discuss the four points in reverse order, starting, so to speak, close to the G-0 prelude and then gradually moving further away from it.

3.4.1. Aspects of the G-0 prelude

We have designed the G-0 prelude with two major requirements in mind: There should be some well established relationship between the graphical mode declarations and ILP constructs; and the prelude should be safe in the sense that it accepts and produces only syntactically correct ILP programs.

There are however two other requirements to be met. Algol68G programs should function in an interactive environment. They should not be too difficult to write, and they should not run too slowly. We have consciously neglected these requirements in this first design stage -- we have the firm belief that optimising a well structured program is always easier than well structuring a poorly designed but fast one.

As for the first of these two stepchildren, we must agree that some of the mode declarations look awkward. For example, we have hesitated some time before we included the rather baroque declaration for PERIOD.

This awkwardness can easily be hidden from the user by declaring some additional procedures and/or operators. Instead of declaring PERIOD as a structure with three integer fields, using negative values to denote DOT and "not specified", or requiring that all three values are always specified, we can also declare an additional procedure

```
   PROC period = (INT d1, g, d2) PERIOD: ( ..... );
```

Such a procedure facilitates construction of a value of mode PERIOD, without disturbing the Algol68G/ILP correspondence.

Efficiency is an other and harder problem. There are several sources of inefficiency in the prelude in its current form. One of them is that many value restriction checks are likely done twice. We intend to use the CHECK operators when values are assembled into a structure or an array, but we also have PICTPROGTOILP perform an entire FULLCHECK on its argument. Leaving out the first CHECK has the disadvantage that errors are no longer detected at the earliest possible moment. Leaving out the FULLCHECK in the I/O procedures means that we can no longer guarantee that only syntactically correct ILP programs are accepted and produced.

A possible solution is to include an extra boolean in each value which is set after a successful check. Such a field should then be protected against change by the particular program.

Two more fundamental sources of inefficiency lie in the treatment of the ILP interface. First, at this moment the output procedure produces symbolic ILP code, and the input procedure accepts symbolic ILP code. Each time a program is read or written, a lot of string manipulation is performed -- not one of the cheapest operations. Second, there is no reason why not all other modules in the graphics system (the ILP interpreter, the picture editor) should be designed equally safe as the Algol68G module -- with the nasty consequence that each time an ILP construct passes a boundary between different modules, a lot of checking on both sides is done, as no module has the slightest faith in any of the others.

Especially in an interactive environment this causes an unacceptable amount of overhead. Short circuiting different modules is the only reasonable solution. This can best be done by sending files around not with symbolic ILP code but with the underlying binary representation, (this solves the first problem) and by defining all modules such that they guarantee correctness of their output but assume correctness of their input. The other way round is equally possible but to us less appealing. (This solves the second problem.) For Algol68G this means rewriting all "put" statements in "pictprogtoilp", rewriting all "get" statements in "ilptopictprog" and deleting the call to FULLCHECK in "ilptopictprog".

It will be clear now why we decided to write a stupid parser which immediately gives up when it detects an error: We plan to rewrite the Algol68G-0 prelude in the way just explained, and we do not expect the parser to have to handle error prone, hand written ILP programs.

Somewhere in the graphics system there might be a module to translate (possibly hand written) ILP programs to some binary equivalent, but this is by no means necessary. It is very well possible that symbolic ILP does not appear anywhere in the system - reducing it to a pure descriptive tool.

3.4.2. Implementation technique: Library prelude vs. preprocessor

Taking the use of Algol68 as a host language for granted, two reasonable implementation techniques present themselves:

1 Define an Algol68 extension and write a preprocessor (preferably, but not necessarily in Algol68) which translates programs in the extended language to equivalent (according to the definition of the extension) Algol68 programs.

2 Write a library prelude.

We chose the second alternative and are satisfied with that choice.

Concerning the theoretical part of the work, it facilitated establishing a formal relationship between Algol68G-0 and ILP. In the case of a language extension translated to Algol68 by a preprocessor, one must consider both the relationship between the extended language and ILP and that between the extended language and Algol68: The semantics of the language accepted by the preprocessor have to be defined in terms of equivalent Algol68 constructs, and it has to be shown that the preprocessor does indeed produce these constructs. In the library prelude case, this relationship needs not to be considered, because there is no such thing as an intermediate Algol68 program. The semantics of the extension defined by the library prelude coincide with the semantics of Algol68. We could concentrate completely on the relationship between Algol68G and ILP.

A second advantage is, that writing a library prelude is comparatively easy. It took us some time to work out the method reported here, but once that was clear the implementation took only about four weeks.

A third advantage is that library preludes can be nested, allowing a modular design of the extension. We have restricted ourselves to the definition of a core system; but we can easily build new layers on top of this one. Such a layered design is harder to realize with a preprocessor.

There are disadvantages as well. An implementor has hardly any freedom to choose a suitable form for an extension. A preprocessor can accept new keywords, allowing constructions like
WITH a1, a2, a3 DRAW p1, WITH a4 DRAW p2, p3 HTIW, p4 HTIW;
It could turn out to be difficult to define procedures and operators such that calls to them have a reasonable format, without too many parentheses and/or casts (look at the call to "declcurves" given in 3.3.2 for example).

Another problem was mentioned already in 3.3.2.: the inaccessibility of the source text obstructs clear error messages.

3.4.3. <u>The use of Algol68</u>

From a theoretical point of view using Algol68 is perfect; few other languages would have allowed us to define such a neat relationship between ILP and an extension with a function similar to that of Algol68G.

We have not been completely happy though about programming in Algol68. The language is not easy to learn; many restrictions can only be understood if one is aware of implementational issues. Also, intensive use of the mode mechanism makes a programmer soon long for an abstract data type facility. In principle simple tasks like copying a PICTPROG value (i.e. making a copy of a complete tree) or comparing two values (in the sense defined in 2.4.2.) require huge procedures; several times we had to write duplicate, triplicate or even quadruplicate operators to perform the same task on objects of different modes (extending a flexible array is a nice example).

It would be very useful to embed ILP in another language as well and then to compare the results.

3.4.4. The two level strategy

$$\begin{tikzcd}
HLGL \arrow[d, shift right=2, swap, "\textcircled{3}"] \\
ILP \arrow[u, shift right=2, swap, "\textcircled{4}"] \arrow[d, shift right=2, swap, "\textcircled{1}"] \\
DM \arrow[u, shift right=2, swap, "\textcircled{2}"]
\end{tikzcd}$$

figure 3

The mathematical centre graphics system is designed according to the scheme shown in figure 3. At the lower level, the ILP interpreter (1) should be able to transform an ILP program to an image which can be made visible on a drawing machine, and an input processor (2) should be able to construct a new ILP program or modify an existing one as a reaction to input from a (graphical) terminal. (See [6] for some suggestions.) The language processor at the higher level should be able to accept, transform and produce ILP programs.

In principle, we do not want the two levels to interfere with each other: Algol68G programs should handle objects which at least have the level of pictures and attributes; i.e. it should essentially handle tree structures. But this restricts the applications of Algol68G to those areas where not much knowledge is needed about the graphical structure of a picture.

For example, we expect Algol68G to be well suited to a task like converting chemical formulas in Wisswesser line notation [5] to ILP pictures representing the structural formula, and even to the reverse task. Writing a general hidden line algorithm in Algol68G will be problematic though, as such an algorithm needs information about the position of certain elements in the final image. Algol68G either has to include procedures which perform these calculations, which we consider as completely unacceptable because it would amount to duplicating a large part of the ILP interpreter into Algol68, or it has to have a communication channel with the interpreter, asking it to calculate parts of an image without sending the result to a drawing machine.

Implementing both tasks in Algol68G will be an excellent way to test many features of the graphics system.

4. Literature

[1]. P.J.W. ten Hagen, "Grafische Programmeertalen", Mathematical Centre Syllabus 25, p. 23-40, 1976.

[2]. T. Hagen, P.J.W. ten Hagen, P. Klint & H. Noot, "ILP. Intermediate Language for Pictures", Report IW 68/77, Mathematical Centre, 1977.

[3]. A. van Wijngaarden et. al. "Revised Report on the Algorithmic Language Algol68", Mathematical Centre Tract 50, 1976.

[4]. T. Hagen, P.J.W. ten Hagen, P. Klint & H. Noot, "The Intermediate Language for Pictures", Information Processing 77, (ed. B Gilchrist), p. 173-178, North Holland Publishing Company, 1977.

[5]. E.J. Smith, "W.J. Wiswesser's Line Formula Chemical Notation," McGraw Hill, New York, 1968.

[6]. P. Klint & H.J. Sint, "A Framework for the Integration of Graphics and Pattern Recognition", Report IW 96/78, Mathematical Centre, 1978.

```
'BEGIN'

#*********************************************************************#
#**                                                                 **#
#** APPENDIX 1.                                                     **#
#**                                                                 **#
#** THIS IS AN EXAMPLE OF AN EMBEDDING GENERATED BY THE ALGOL68     **#
#** VERSION OF THE EMBEDDING GENERATOR.                             **#
#** THE INPUT GRAMMAR WAS                                           **#
#**              G = [ [ S, X, Y, Z],                               **#
#**                    [ A, B, C, EMPTY],                           **#
#**                    [ S -> B B X ! A A Y ! C C C ! Z,            **#
#**                      X -> C X ! EMPTY,                          **#
#**                      Y -> B B X,                                **#
#**                      Z -> EMPTY ]                               **#
#** WHERE "EMPTY" DENOTES THE EMPTY STRING.                         **#
#**                                                                 **#
#** THE TAGS AND BOLDTAGS ARE MADE ACCORDING TO THE FOLLOWING RULES: **#
#** - TAG RETURNS ITS ARGUMENT IF THAT HAPPENS TO BE AN IDENTIFIER  **#
#**   NOT USED FOR OTHER PURPOSES; OTHERWISE A TAG FROM THE SERIES  **#
#**   "S11", "S12", "S13",..... IS ASSOCIATED WITH IT.              **#
#**   FOR EXAMPLE, TAG("A") = "A", TAG("EMPTY") = ("EMPTY"),        **#
#**   TAG("B B X") = "S11".                                         **#
#** - BOLDTAG RETURNS ITS ARGUMENT SURROUNDED BY SINGLE QUOTES IF IT **#
#**   IS AN IDENTIFIER AND DOES NOT CORRESPOND TO A RESERVED WORD,  **#
#**   OTHERWISE A BOLD TAG FROM THE SERIES "'B11'", "'B12'", "'B13'" **#
#**   ...... IS ASSOCIATED WITH IT.                                 **#
#**   (WHY DIDN'T WE START WITH 'B1'?. WE DIDN'T START WITH 'B1'    **#
#**   BECAUSE THE CDC COMPILER STUBBORNLY REFUSED TO ACCEPT 'B9'.)  **#
#**   FOR EXAMPLE, BOLDTAG("A") = "'A'", BOLDTAG("EMPTY") = "'B11'", **#
#**   BOLDTAG("B B X") = "'B12'".                                   **#
#** - S RETURNS A SUFFIX WHICH CONSISTS OF THE SMALLEST POSSIBLE    **#
#**   NUMBER OF "I"-S; USUSALLY 0 BUT SOMETIMES 1 OR 2; SEE FOR     **#
#**   EXAMPLE MODE 'B12' GENERATED FOR "B B X".                     **#
#**                                                                 **#
#** THIS COMMENT AND SOME LAY OUT SYMBOLS WERE ADDED TO THE OUTPUT  **#
#** BY HAND.                                                        **#
#**                                                                 **#
#*********************************************************************#

'MODE' 'ONE'='BOOL';

'BOOL' ONLY = 'TRUE';
'MODE' 'A'='STRUCT'('ONE' A);

'MODE' 'B'='STRUCT'('ONE' B);

'MODE' 'C'='STRUCT'('ONE' C);

'MODE' 'B11'='STRUCT'('ONE' EMPTY);

'MODE' 'B12'='STRUCT'('REF' 'B' B,'REF' 'B' BI,'REF' 'X' X);

'MODE' 'B13'='STRUCT'('REF' 'A' A,'REF' 'A' AI,'REF' 'Y' Y);
```

```
'MODE' 'B14'='STRUCT'('REF' 'C' C,'REF' 'C' CI,'REF' 'C' CII);

'MODE' 'S'='UNION'('B12','B13','B14','Z');

'MODE' 'B15'='STRUCT'('REF' 'C' C,'REF' 'X' X);

'MODE' 'X'='UNION'('B15','B11');

'MODE' 'Y'='B12';

'MODE' 'Z'='B11';




'PROC' A68TOA =('A' A)'STRING':
"A ";

'PROC' A68TOB =('B' B)'STRING':
"B ";

'PROC' A68TOC =('C' C)'STRING':
"C ";

'PROC' A68TOEMPTY =('B11' EMPTY)'STRING':
"";

'PROC' A68TOS11 =('B12' S11)'STRING':
A68TOB(B  'OF' S11) +
A68TOB(BI 'OF' S11) +
A68TOX(X  'OF' S11);

'PROC' A68TOS12 =('B13' S12)'STRING':
A68TOA(A  'OF' S12) +
A68TOA(AI 'OF' S12) +
A68TOY(Y  'OF' S12);

'PROC' A68TOS13 =('B14' S13)'STRING':
A68TOC(C   'OF' S13) +
A68TOC(CI  'OF' S13) +
A68TOC(CII 'OF' S13);

'PROC' A68TOS =('S' S)'STRING':
'CASE' S 'IN'
       ('B12' S11): A68TOS11(S11),
       ('B13' S12): A68TOS12(S12),
       ('B14' S13): A68TOS13(S13),
       ('Z'   Z  ): A68TOZ(Z)
'ESAC';

'PROC' A68TOS14 =('B15' S14)'STRING':
A68TOC(C 'OF' S14) +
A68TOX(X 'OF' S14);
```

```
'PROC' A68TOX =('X' X)'STRING':
'CASE' X 'IN'
       ('B15' S14  ): A68TOS14(S14),
       ('B11' EMPTY): A68TOEMPTY(EMPTY)
'ESAC';

'PROC' A68TOY =('Y' Y)'STRING':
A68TOS11(Y);

'PROC' A68TOZ =('Z' Z)'STRING':
A68TOEMPTY(Z);




'PROC' ATOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'A':
('HEAP' 'A' T;
 A 'OF' T := ONLY;
 T);

'PROC' BTOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'B':
('HEAP' 'B' T;
 B 'OF' T := ONLY;
 T);

'PROC' CTOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'C':
('HEAP' 'C' T;
 C 'OF' T := ONLY;
 T);

'PROC' EMPTYTOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'B11':
('HEAP' 'B11' T;
 EMPTY 'OF' T := ONLY;
 T);

'PROC' S11TOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'B12':
'HEAP' 'B12' := (BTOA68((DESCENDANTS 'OF' PARSETREE)[1]),
                 BTOA68((DESCENDANTS 'OF' PARSETREE)[2]),
                 XTOA68((DESCENDANTS 'OF' PARSETREE)[3]));

'PROC' S12TOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'B13':
'HEAP' 'B13' := (ATOA68((DESCENDANTS 'OF' PARSETREE)[1]),
                 ATOA68((DESCENDANTS 'OF' PARSETREE)[2]),
                 YTOA68((DESCENDANTS 'OF' PARSETREE)[3]));

'PROC' S13TOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'B14':
'HEAP' 'B14' := (CTOA68((DESCENDANTS 'OF' PARSETREE)[1]),
                 CTOA68((DESCENDANTS 'OF' PARSETREE)[2]),
                 CTOA68((DESCENDANTS 'OF' PARSETREE)[3]));

'PROC' STOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'S':
('REF' 'PARSETREE' CHILD = (DESCENDANTS 'OF' PARSETREE)[1];
 'HEAP' 'S' := 'IF' NODE 'OF' CHILD = "B B X"
               'THEN' S11TOA68(CHILD)
```

```
                'ELSE' 'IF' NODE 'OF' CHILD = "A A Y"
                       'THEN' S12TOA68(CHILD)
                       'ELSE' 'IF' NODE 'OF' CHILD = "C C C"
                              'THEN' S13TOA68(CHILD)
                              'ELSE' ZTOA68(CHILD)
                              'FI'
                       'FI'
                'FI'
);

'PROC' S14TOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'B15':
'HEAP' 'B15' := (CTOA68((DESCENDANTS 'OF' PARSETREE)[1]),
                 XTOA68((DESCENDANTS 'OF' PARSETREE)[2]));

'PROC' XTOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'X':
('REF' 'PARSETREE' CHILD = (DESCENDANTS 'OF' PARSETREE)[1];
 'HEAP' 'X' := 'IF' NODE 'OF' CHILD = "C X"
                'THEN' S14TOA68(CHILD)
                'ELSE' EMPTYTOA68(CHILD)
                'FI'
);

'PROC' YTOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'Y':
S11TOA68((DESCENDANTS 'OF' PARSETREE)[1]);

'PROC' ZTOA68 =('REF' 'PARSETREE' PARSETREE)'REF' 'Z':
EMPTYTOA68((DESCENDANTS 'OF' PARSETREE)[1]);

'PROC' LGTOA68 =([]'STRING' S)'REF' 'S':
('REF' 'PARSETREE' PARSETREE = PARSER(S);
 'IF' PARSETREE 'IS' 'REF' 'PARSETREE'('NIL')
 'THEN' 'NIL'
 'ELSE' STOA68(PARSETREE)
 'FI' );
```