J. M. Lafuente
D. Gries

Language Facilities for Programming User-Computer Dialogues

Abstract: Extensions to PASCAL that provide for programming man-computer dialogues are proposed. An interactive application program is viewed as a sequence of *frames*, representing stages of dialogue activity, and separate computational steps. First, extensions are presented to allow the description of the items of information contained in each frame. Second, PASCAL is extended to allow the inclusion of *behavior rules* for a frame to specify the interactive dialogue. The behavior rules are specified nonprocedurally. Previously, programming such dialogues has required the specification of all possible interactions and their effects in a procedural fashion.

1. Introduction

Typical applications of interactive user-computer dialogue systems include the collection of business data (key data entry), question-answering sessions typified by computer-aided instruction (CAI) systems, and interactive data base systems, such as may be used in banks or airline reservation systems. These systems can be characterized as follows:

1. The system is in a state (often called a *frame* in CAI systems [e.g., 1, 2]) in which text is displayed to the user, and the user answers questions, pushes buttons, or types in values. The system monitors the user input for validity and consistency and asks the user to supply additional values if necessary. This process may cause changes in the format and content of the text being displayed, but, basically, the system is in one state or frame.
2. After values have been satisfactorily keyed in, the system performs some computations based on these values. It then selects another (or the same) frame and returns to step 1 to resume the dialogue in the new frame.

Previously, such systems have been implemented in assembly language or in FORTRAN or other high level languages. Some work has been done to make languages more amenable to programming dialogue systems. For example, some have considered attention-handling [3, 4] constructs such as

ON LIGHTPEN DETECT THEN GO TO LABEL1; and
IF KEY 5 THEN CALL KPROG5;

while others [5, 6] have suggested state diagrams for describing and restricting interactions. Most of the techniques used in the design and implementation of such dialogues have been adequately summarized by Martin [7].

However, the design and implementation of such dialogues is still a long, arduous task, partly because the programmer must describe everything—how text is to be displayed, how user inputs are to be monitored, etc.—in a procedural fashion. For example, if there are restrictions on the combinations of two input values required from the user, then the test for these restrictions must be programmed as a sequence of procedural statements.

In this paper, we propose extensions to the programming language PASCAL [8, 9] that are designed to ease the description of interactive dialogues. First, we provide for the description of the layout of information to be displayed for a frame, in a device-independent manner. This means providing for a description of the logical units of information—the *items*—that may appear on the display screen. Essentially, an item is a PASCAL variable with some additional characteristics, such as the position on the screen, the text, the format of values that the item can have, whether or not the value can be modified by the user, and so forth. Our notation also provides for grouping items into logical units called *subframes*, which can be used in more than one frame.

```
                    BANK OF NEW YORK

                      NEW ACCOUNT

Enter information. Hit ENTER when done.
NAME:
SEL. SERVICE NO.:               SEX:   *MALE
                                       *FEMALE
NO. CHILDREN:
SALARY:                         STATUS: *SINGLE
SPOUSE'S SALARY:                        *MARRIED
```

Figure 1 A typical frame.

Second, we extend PASCAL to allow the inclusion of *behavior rules* for a frame, which are, essentially, the specifications of the interactive dialogue. A behavior rule is either a Boolean expression that describes a constraint on items and their values (e.g., whether or not the user is to select one or more options from the item), or it is a statement that expresses the conditions under which an error should be reported, a value should be changed by the system, the text being displayed should be changed, the dialogue is to terminate, etc. The important point is that these behavior rules are specified nonprocedurally. Thus, the programmer need not be concerned with the order in which the rules are executed. Moreover, the compiler can ensure that the rules are consistent and can disregard rules that are superfluous.

We restrict our attention to facilities for implementing *alphanumeric* user-computer dialogues (as distinguished from dialogues involving the manipulation of graphical entities, such as lines, points, etc.). Our intent is not to give a complete, rigorous, formal definition but to outline briefly ideas which we believe are interesting and useful for programming such dialogues. More work needs to be done if a full language is to be designed and implemented [10]. We do give some hints on implementation where appropriate, and more information can be found in [11].

PASCAL has been chosen as a base notation for expressing our ideas because of its simplicity and clarity of definition, its block structure, and its flexible data type constructs. As will be seen, our extensions blend quite well with the PASCAL concepts and syntax. We assume the reader is familiar with PASCAL.

We do assume several simple and hopefully obvious extensions to PASCAL, such as the use of the break character "_" in identifiers. Its scope and block structure rules are also assumed to hold for frames and subframes (which are defined later).

We make use of the operators **cor** and **cand** (conditional **or** and conditional **and**), which can be used in place of **or** and **and** to "short-circuit" evaluation of a ⟨*condition*⟩ or Boolean expression:

a **cor** *b* ≡ *if a then true else b*
a **cand** *b* ≡ *if a then b else false*

The reader must realize that we have left much to his imagination, and we rely heavily on his knowledge of current systems. For example, we assume a device-independent, two-dimensional description of a display screen, with rows numbered 0 through 40 (say) and columns numbered 0 through 100 (say), each element of which can contain a character. The interface between this grid and a real display must be specified in some "job control language" that we do not describe. The fact that a light pen is triggered when pointing at a certain place on the screen is described in the language as "selecting" the option being displayed at that point. Similarly, pushing a button is referred to as selecting an option; how the interface is realized is beyond the scope of this paper.

2. Short example

Before proceeding with a description of the language extensions, we present and discuss a small example to illustrate some of the language features and their use.

Figure 1 shows a frame as the user might see it on a display screen. He is to key in the obvious information requested—name, selective service number if the user is male, number of children, salary, and spouse's salary. The user must also select either *MALE or *FEMALE and either *SINGLE or *MARRIED. When finished, he is to depress the ENTER button on the keyboard.

Shown on this screen are *display* items, each consisting only of text, *key-in* items, each consisting of text and a field for a value to be entered by the user, and *menu* items, each consisting of text together with a set of options from which the user makes a selection. The two display items are declared as follows:

var *title*: **display text** 'BANK OF NEW YORK';
    heading: **display text** 'NEW ACCOUNT' **at** (0,16),
        'Enter information. Hit ENTER when done.' **at** (2,0)

The tuple following **at** indicates the row and column numbers where the text is to be placed relative to the origin of the item [(0,0) is the default]. The position of the item itself on the screen is not established by this declaration.

There are five key-in items, which are declared below. For each, we give the type of value to be keyed in, the position of the field for the value (relative to the item origin), the text, and a format under which the value is to be read in. In all of these cases, the relative position of the text is assumed to be (0,0).

var *name*: **key-in** *char* **at** (0,6) **text** 'NAME:' **format** A20;
    sel_ser: **key-in** *integer* **at** (0,18)
        text 'SEL. SERVICE NO.:' **format** I9;
    nchild: **key-in** *integer* **at** (0,14)
        text 'NO. CHILDREN:' **format** I10;

salary: **key-in** *real* **at** (0,8) **text** 'SALARY:'
format F15;
salary2: **key-in** *real* **at** (0,17)
text 'SPOUSE''S SALARY:' **format** F15

Finally, we define the menu items *sex* and *status*:

type *options1* = **set of** (MALE, FEMALE);
var *sex*: **menu** *options1* **at** (0,6) **text** 'SEX:';
type *options2* = **set of** (SINGLE, MARRIED);
var *status*: **menu** *options2* **at** (0,8) **text** 'STATUS:'

Note that in a menu item we specify a *set* of options, which are essentially string constants. The first option is displayed at the indicated position with respect to the item origin, with an asterisk preceding it; the remaining options in successive rows immediately below.

Suppose that these variables have been declared, either locally within a frame or globally. We now add to the definition of the frame the **contains** clause, which describes where each of these items is to appear on the screen. (The top leftmost position is row 0, column 0.)

contains *title* **at** (0,17); *heading* **at** (2,3);
name **at** (5,3); *sel_ser* **at** (6,3);
sex **at** (6,31); *nchild* **at** (8,3);
salary **at** (9,3); *salary2* **at** (10,3);
status **at** (9,31)

We now wish to specify the following constraints and relations among the items defined above:

1. (Requirement) At least one option must be selected from *sex*; a value must be entered for *salary*.
2. (Limit) Only one option can be selected from *status* and *sex*.
3. (Requirement) *nchild* must have a nonnegative value (which is disregarded unless the user is married).
4. (Requirement) If the user is male, then a value must be entered for *sel_ser*.
5. (Exclusion) If the user is female, then a value should not be entered for *sel_ser*.
6. (Requirement) *salary2* is required if the user is married and *salary* is less than $10 000.
7. (Binding) The item *salary2* is to appear on the screen only if the user is married.
8. (Terminate) Process the given information when the ENTER key is depressed.

To express these constraints and relations, we write behavior rules for the frame as follows:

require *sex, salary*;
with *sex*: **allow only** 1 **option**;
with *status*: **allow only** 1 **option**;
require *nchild* $\geq$ 0 **if** MARRIED **in** *status*;
require *sel_ser* **if** MALE **in** *sex*;
exclude *sel_ser* **if** FEMALE **in** *sex*;
require *salary2* **if** MARRIED **in** *status* **cand**
salary < 10000;
let *salary2.d* **be** MARRIED **in** *status*;
terminate if ENT_KEY

The *.d* part of the notation *salary2.d* refers to the *display/no-display* property of item *salary2*, i.e., whether or not it is to be displayed to the user. The last rule indicates that the dialogue is to terminate when the user depresses the ENTER key, provided that there are no errors; if a rule is violated, however, an error message is given to the user and the dialogue is not terminated.

This small example is intended to give an idea of what we are proposing. We now go into more detail: Sections 3, 4, 5, and 6 discuss items, subframes, frames, and behavior rules, respectively.

3. Items

Like a variable of a program, an item may contain a value of a specific type, and this value may be changed during the execution of a program. However, unlike a variable, an item has attributes that determine where and how it is to appear on a two-dimensional display. An item *declaration* describes the type of value that the item can have, an initial value (if desired), and the position of that value relative to the origin of the item. It may also describe additional text to be displayed along with the value. Finally, an item may be associated with an input or output file; in this case the item can be considered as a "window" through which the file can be viewed by the user.

We describe four kinds of items in this section: *display*, *key-in*, *menu*, and *attention*. A fifth, the *list* item, is similar to the menu item; it specifies a list of values, instead of a set of values, and is useful when the order of selection of options is important. The list item is not needed in our examples; hence it has been left out. A sixth, the *table* item, although quite useful, is fairly complicated and is not needed for a general view of our ideas; its description has been relegated to Appendix A.

The overall syntactic structure of an ⟨*item type*⟩ and an ⟨*item declaration*⟩ is given below. As usual, the brackets "[ ]" denote an optional syntactic entity. The reader should realize that not all parts of an ⟨*item type*⟩ may be present for a particular item kind. To keep our description simple, we give a grammar that "covers" a more exact grammar for the language (which is given in Appendix B). When discussing each item kind, it will become clear which parts are allowed and which are not.

⟨*item type*⟩ ::= ⟨*item kind*⟩ ⟨*type*⟩ [⟨*position*⟩]
[**text** ⟨*string*⟩ ⟨*position*⟩
{, ⟨*string*⟩ ⟨*position*⟩}]
[**initial** ⟨*expression*⟩]
[**file** ⟨*id*⟩]
⟨*item kind*⟩ ::= **display** | **key-in** | **menu** | **attention**

$\langle$*position*$\rangle$ ::= **at** ($\langle$*row number*$\rangle$, $\langle$*column number*$\rangle$)
$\langle$*type*$\rangle$ ::= a PASCAL data type
$\langle$*item declaration*$\rangle$ ::= $\langle$*id*$\rangle$: $\langle$*item type*$\rangle$

We consider the **file** $\langle$*id*$\rangle$ part separately, after the various item kinds have been discussed.

Before proceeding with a discussion of the specific item kinds, let us make some general observations. First, a $\langle$*position*$\rangle$ gives the relative row and column numbers where the value or text string is to be placed. An item declaration does not indicate where the item is to actually appear on the display; this is specified for each frame by means of a **contains** clause to be defined later. If $\langle$*position*$\rangle$ is omitted, (0,0) is assumed.

Secondly, the **initial** $\langle$*expression*$\rangle$ gives an initial value for the item. The type of the $\langle$*expression*$\rangle$ is assumed to match the $\langle$*type*$\rangle$ of the value of the item.

Thirdly, any item has three properties that affect its appearance and behavior, and these properties may change as the dialogue progresses:

1. The *selected/unselected* property indicates whether or not the item contains a value. We say that the *status* of an item is *selected* or *unselected*. An item is unselected until it receives a value by an assignment in the program, by means of the **initial** clause in its declaration, or by user's input. Provision should also be made for an item to be *deselected* (set to unselected status) by the user (perhaps by pointing again to it with a light pen).
2. The *enabled/disabled* property indicates whether or not the user is permitted to act on the item—enter a value, etc.
3. The *display/no-display* property indicates whether or not the item is to appear on the display device. Thus an item may appear or disappear from view, depending on actions by the user or the program.

The programmer must have the means for testing and changing these properties. To do this, we use the PASCAL notation for referencing fields of a record. Thus, for an item x,

$x.s$	is a Boolean value that describes the selected/unselected property of x, i.e., $x.s$ = *true* if x is selected, *false* otherwise.
$x.d$	is *true* if x is to be displayed, *false* otherwise.
$x.e$	is *true* if the user is allowed to act on item x, *false* otherwise.

Thus, the assignment $x.d := y.s$ can be interpreted as "if y is selected, then display x; otherwise, do not display x."

We are now ready to discuss the individual kinds of items.

• *Display item*
A *display* item is used to present information consisting of text and a value to the user. The program may change the value, but the user may not. Thus, the enabled/disabled property $x.e$ of a display item x is always *false*. The example of Section 2 contained two display items whose declarations contained no $\langle$*type*$\rangle$. Thus the items were used only to present information to the user, and no values were associated with them. Another declaration of a display item with both text and a value is

var *x*: **display** *integer* **at** (0,16)
    text 'current_number:' **at** (0,0).
    'maximum: 9' **at** (1,0) **initial** 0

Whenever and wherever item x is displayed, it will look as follows (except that its value, initially 0, may have been changed by the program):

current number: 0
maximum: 9

• *Key-in item*
A *key-in* item x (say) is similar to a display item except that the user is allowed (usually expected) to enter a value for it, as long as its enabled/disabled property is *true*. He can do this by positioning the cursor on the display screen at the place where the value for x belongs (or, alternatively, by aiming the light pen there) and then typing in a value. Upon typing in such a value, x automatically assumes status *selected*, i.e., $x.s$ is set to *true*. Presumably, the user would also have means for deselecting the item—for erasing its value completely—thus setting $x.s$ to *false*. In this case, the value of the key-in item is undefined. The programmer can also do this simply by writing $x.s :=$ *false*.

Consider the item declaration

var *y*: **key-in** *integer* **at** (0,16)
    text 'current number:' **at** (0,0)

When first displayed, it will look like "current number:". After the user keys in the value 6 (say), the item value becomes 6, $y.s$ is set to *true*, and the screen is changed to "current number: 6".

• *Menu item*
The *menu* item is used to describe a set of options from which the user may make zero or more selections. In the $\langle$*item type*$\rangle$ specification (see the beginning of Section 3), the $\langle$*type*$\rangle$ must be a PASCAL set type. The value of a menu item is always a subset of the set of values defined by the $\langle$*type*$\rangle$. Consider, for example, the following declaration:

type *color* = **set of** (*red, white, blue*);
var *z*: **menu** *color* **at** (1,2)
    text 'Which color do you like?' **at** (0,0)

In PASCAL, *color* is a set type which denotes the powerset of the set of values {*red, white, blue*}. The value of *z* can be any subset of the base type {*red, white, blue*}. Facilities exist in PASCAL for adding a member of {*red, white, blue*} to z and for testing whether z contains a value from {*red, white, blue*} (for example, *red* **in** z yields *true* or *false* depending on whether *red* is in z or not). These facilities are available to the programmer in designing and coding the dialogue.

For a menu item z, $z.s$ corresponds to whether the value of z is nonempty ($z.s = true$) or is the empty set ($z.s = false$). When the above item z is displayed, it will look like

Which color do you like?
  *red
  *white
  *blue

Thus, the options are displayed beginning in the same column, in successive rows, with the first option beginning at (1,2) (with an asterisk preceding it). If the user now selects "*red" and "*blue," the display will be changed to reflect this fact (perhaps by brightening the selected options). Then $z.s = true$ and the value of z is the set {*red, blue*}. Similarly, by aiming the light pen at an already selected color, he could delete it from z.

We summarize. The value of a menu item z is a subset of the set of options defined by the ⟨*type*⟩ in the declaration of z, and $z.s \equiv card(z) > 0$ (the cardinality of z is greater than 0). All options of the specified ⟨*type*⟩ are displayed in a standard format, with the ones currently in z indicated in some manner. The user may select an unselected option (add it to z) or deselect a selected option (delete it from z). The programmer may reference the value of z (change z, test whether an option is in z, etc.) as he would normally use a variable of type *set*.

- *Attention item*

An *attention* item is declared in the same fashion as a menu item (except that the item kind is **attention**). The ⟨*type*⟩ must be a scalar or subrange type. The difference between the two kinds of items lies in how their corresponding values are treated. The value of an attention item is always a scalar from the set of values defined by ⟨*type*⟩ (or undefined, if no option has been selected). The item is always restored to unselected status whenever the system becomes ready to accept the next user action.

The reason for attention items is to allow the use of special buttons or keys or to make it simple for the programmer to specify an immediate system response when a phrase on the display screen is pointed at with a light pen. Any number of such keys, buttons, or phrases may be specified as options of an attention item. After the user pushes a button or selects a phrase, the system may record the input and then deselect the option (that is, set the status of the item to unselected). The user may then choose another option.

Consider, for example, an attention item declared as follows:

```
type choices = (PROCEED, RESET, TERMINATE);
var x: attention choices at (1,2)
      text 'What would you like to do next?'
```

When the above item is displayed to the user, it will appear as follows:

What would you like to do next?
  *PROCEED
  *RESET
  *TERMINATE

The programmer may wish to specify a system response for each option as follows. When the user selects *PROCEED, the current frame dialogue is to terminate, information entered by the user is to be recorded, and the next frame is to be presented. If *RESET is selected, all key-in items in the current frame are to be "reset" (i.e., their status set to unselected) and the user is to type in new values. Finally, if *TERMINATE is chosen, the dialogue is to terminate. In any case, $x.s$ is set to *true* after the user action, until the system is ready to accept the next user action, at which time $x.s = false$ and the value of x is again undefined.

- *Associating a file with an item*

In many data-directed interactive applications it is necessary to read information from a file, present the information to the user (usually in tabular form), accept changes from the user, and then rewrite the file with the updated information. A table item, described in Appendix A, is part of the mechanism for doing this, and a more detailed discussion appears there. However, the association of a file with an item is interesting enough to be explained here.

If a file attribute **file** f appears in the declaration of a key-in item x, then f must be an identifier denoting a PASCAL input or output sequential file. We say that item x is *bound* to the file f. The value of x is always the value of the buffer variable $f\uparrow$ whose value can be used and changed by file operators. Thus, item x becomes a "window" through which file f can be viewed. In the case of an input file, the programmer can read the components of the file using the PASCAL operators *reset* and *get*, whereas an output file can be written using the operators *rewrite* and *put*.

A simple example will illustrate these ideas. Suppose *oldf* is a file of real numbers to be inspected. The user looks at each value, changes it if necessary, and then pushes a key, which causes the value to be appended to an output file *newf*. We declare two items *k1* and *a1* and a procedure *read_file* as follows:

```
BANK OF NEW YORK  JULY 4, 1976
                  1:30 P.M.
```

Figure 2 A subframe.

```
type oldf = file of real;
     newf = file of real;
var kl: key-in real at (2,15)
        text 'Change value if necessary.' at (0,0),
             'Depress key 1 when done.' at (1,0),
             'Next value is:' at (2,0)
        input file oldf output file newf
        format F10.2;
    al: attention (KEY12);
    .
    .
    .
procedure read_file;
begin if not eof(f) then
        begin put(newf); get(oldf)
        end
end
```

In this example, *kl* is a key-in item bound to files *oldf* and *newf*, and *al* is an attention item containing only the option KEY12, which denotes, say, hardware key number 12. Procedure *read_file* is invoked each time the user depresses this key. (This is specified by the programmer with a *terminate* behavior rule, described in Section 6.) This procedure writes the current value of *kl* (which is also the value of *oldf*↑ and *newf*↑) into *newf* and reads the next value from *oldf*, which is then displayed to the user automatically by virtue of the attribute **input file** *oldf*.

4. Subframes

The subframe notation is used to group together a set of items (and/or other subframes) as a logical unit. In addition to items declared locally, the subframe may contain items and other subframes declared globally. It must specify where (on the display screen) each of the included items and subframes is to be placed. This, of course, is relative to the row and column where the subframe being defined is to be placed. It may also contain behavior rules (see Section 6) which describe restrictions and interrelations on the items of the subframe.

A subframe type is thus similar to a PASCAL record type in that it allows the programmer to view a collection of objects as a single value. The syntax is:

⟨*subframe type*⟩ ::= **subframe** ⟨PASCAL *declaration*⟩
[⟨*item inclusion part*⟩]
[⟨*behavior rules*⟩] **end**

In the ⟨PASCAL *declaration*⟩ the programmer can declare items (and other subframes) which are local to the subframe. The ⟨*item inclusion part*⟩ lists all items and subframes (including local ones) which are considered to be part of the subframe, together with their position with respect to the subframe:

⟨*item inclusion part*⟩ ::= **contains** ⟨*id*⟩ **at** ⟨*position*⟩
{; ⟨*id*⟩ **at** ⟨*position*⟩}

As an example, suppose we want a subframe to contain the information in Fig. 2, where the items *date* and *time* have already been declared globally. The subframe *y* would be given by

```
var y: subframe
        banktext: display text 'BANK OF NEW YORK';
        contains banktext at (0,2);
                 date at (0,20);
                 time at (1,20)
     end
```

5. Frames

A *frame* describes a set of dialogues between the system and the user. It is similar in concept and definition to a PASCAL procedure without parameters, except that it also describes how information is to be displayed to the user and how the user is to interact with the program.

A frame definition may contain declarations of local variables, items, subframes, functions, and procedures to be used in the frame. In addition, it may contain

1. A procedure *initialize*, which is automatically invoked (if declared) upon entry to the frame in order to perform necessary initialization.
2. A **contains** clause, as described above in the discussion of subframes; this specifies the items to be displayed together with their position and other attributes (format, intensity, text of the error message if an illegal value is supplied, etc.).
3. A set of *behavior rules* (described in Section 6), which describes the user interaction with the program (for this frame), restrictions on this interaction, and conditions for termination of the frame dialogue.
4. A procedure *terminate*, which is automatically invoked (if declared) upon termination of the dialogue, just before the frame itself is terminated.

The "execution" of a frame thus proceeds as follows. First, procedure *initialize* is executed. Then the items are displayed to the user and a dialogue takes place. Upon its termination, procedure *terminate* is executed (if it exists) and the frame itself terminates. (Control is returned to just after the point of invocation.)

Let us illustrate this with a simple example in which a global item is used in two frames in different ways. Suppose, for example, that the following items were declared globally:

```
var accno: key-in char at (0,16)
           text 'ACCOUNT NUMBER:';
   balance: key-in real at (0,9)
           text 'BALANCE:' format F10.2
```

In one frame, say *F1*, the item *accno* is displayed to the user and he must specify a value, which the system then uses to search a file for information concerning his bank account:

```
frame F1;
  contains accno at (3,3)
           intensity high format I6
           format msg 'Invalid value – respecify';
  procedure initialize;
  begin accno.s := false
  end
endframe
```

When this frame is presented to the user, item *accno* will appear at position (3,3). The **intensity** attribute specifies that the item is to be brightened (high intensity) in this frame. The **format msg** attribute specifies an error message to be given to the user if the value entered for *accno* is not a six-digit number.

In another frame, *F2*, the same item is to be displayed to the user but at a different position and with different attributes. The purpose of this frame is to allow the user to request a bank "operation" (for example, to show his account balance). Thus we declare the local item *op_type* and include the global item *accno*:

```
frame F2;
  var op_type: attention (KEY1, KEY2, KEY3);
  contains accno at (5,5); op_type;
  rules let accno.e be false;
       terminate if op_type.s
  end
  procedure terminate;
  begin case op_type of
    KEY1: begin balance := get_bal;
             balance.d := true
           end
      .
      .
      .
  end
endframe
```

The **let** rule prevents the user from changing the account number (the item is disabled). When the user selects the option KEY1, the procedure *terminate* is automatically invoked; the function *get_bal* obtains the account balance, which can then be displayed to the user in another (or the same) frame.

We see from this example that different attributes and behavior rules that affect an item can be specified in each frame where the item is to be used. Consequently, whereas the basic characteristics (item kind, data type, initial value, etc.) of an item do not change, the behavior and appearance of an item can vary from one frame to another.

As a convenience for the programmer, we relax the syntax rules and allow frame attributes to be specified directly in the declaration of local items or subframes (thus eliminating the need for a **contains** clause for local items). For example, instead of writing

```
frame F1;
  var x: key-in char at (0,12) text 'ENTER NAME:';
  contains x at (1,2) format A15;
  .
  .
  .
endframe
```

one can, equivalently, write

```
frame F1;
  var x: key-in char at (1,14) text 'ENTER NAME:'
        at (1,2) format A15;
  .
  .
  .
endframe
```

- *A PASCAL program*

Finally, we need to extend the definition of a PASCAL program to include a ⟨*frame definition part*⟩, which consists of one or more frame definitions. We can now think of a program as a description of a class of *application dialogues* and all the processing steps required to support any such dialogue. The processing is specified by procedure declarations and by the ⟨*statement part*⟩ of the program. The ⟨*statement part*⟩ contains statements to be initially executed and statements to invoke frames and to determine the order of presentation of frames.

6. Behavior rules

As the user acts on items of a frame, the appearance of the frame changes depending on the behavior rules that have been specified for the frame. For example, entering data for an item may cause a message to be displayed, pressing a key may result in tutorial information being given, and pointing to an item option with a light pen may cause part of another item to be brightened or dimmed or to disappear altogether. With complicated relationships, a simple user action can cause the appearance of the frame to change quite radically, so much so that the user no longer sees it as the same frame. Nevertheless, the basic notion is that it *is* the same frame, at least as far as the programmer perceives it. The user, however, does not need to think in terms of frames.

Three types of behavior rules can be used to guide the interactions with the user during a frame dialogue. The *binding rule* is used to specify the conditions under which the value (and status properties) of a variable should be changed by the system; the *requirement rule* is used to specify the conditions that must be *true* before the frame dialogue may terminate; and the *terminate rule* is used to specify when termination is to occur.

- *Binding rule*

The binding rule has the general form

let ⟨*variable*⟩ **be** ⟨*expression*⟩ [**if** ⟨*condition*⟩]

The conventional rules about matching types of ⟨*expression*⟩ and ⟨*variable*⟩ in an assignment apply here also. After each response by the user, all **let** rules are processed (in an undetermined order and perhaps several times each) as follows: If a ⟨*condition*⟩ of a **let** rule is *true*, then the ⟨*expression*⟩ is evaluated and assigned to the ⟨*variable*⟩.

After processing all the **let** rules, for any rule with its ⟨*condition*⟩ *true*, the ⟨*variable*⟩ will have the value of the ⟨*expression*⟩.

We realize that with these semantics, **let** rules may be difficult to implement efficiently. Different ordering of processing of the rules can produce different results, the processing may not terminate, and there may be inconsistencies that cannot be resolved. For example, consider

let $x.s$ **be not** $x.s$ **if not** $x.s$; **let** $x.s$ **be not** $x.s$ **if** $x.s$ or
let a **be** $b + 1$; **let** b **be** $a + 1$

Proposals have been made for languages in which the order of execution is not explicitly given [12–14]. In particular, Foster and Elcock [14] describe a compiler for a generalized language consisting of such "assertions." We believe that the use of **let** behavior rules can be restricted so as to allow efficient processing and safe checks for termination, without harming flexibility too much. One simple restriction is that it be possible to order the rules so that the ⟨*variable*⟩ of **let** rule i does not appear in rules 1 through $i - 1$. We adhere to this restriction in this paper.

- *Requirement rule*

The rule

require ⟨*condition*⟩

is used to indicate that ⟨*condition*⟩ must be *true* before the frame dialogue may terminate. Should termination be signaled when ⟨*condition*⟩ is *false*, a message is displayed to the user, who must then respecify entries so that ⟨*condition*⟩ is *true*.

- *Terminate rule*

The rule

terminate if ⟨*condition*⟩

indicates that when ⟨*condition*⟩ is *true*, the current frame interactions are to terminate and control is to return to the point in the program where the current frame was invoked. After performing computations, the program may invoke another (or the same) frame or terminate execution.

As examples of these basic types of rules, we rewrite some of the behavior rules of the example presented in Section 2. Remember, for an item x, $x.s$ refers to the selected/unselected status, and $x.d$ to its display/no-display status.

require *sex.s*;	{a *sex* option must be selected}
require *salary.s*;	{a *salary* must be given}
require *card*(*sex*) = 1;	{exactly one of MALE and FEMALE must be chosen}

require *card*(*status*) = 1;
require not (MARRIED **in** *status*) **cor** *nchild* ≥ 0;
require not MALE **in** *sex* **cor** *sel_ser.s*;
require not FEMALE **in** *sex* **cor not** *sel_ser.s*;
let *salary2.d* **be** (MARRIED **in** *status*);
terminate if ENT_KEY

- *Some syntactic sugar*

Although the three types of rules above are powerful enough, some syntactic sugar can increase the ease of writing behavior rules. One abbreviation is simply to use the item name x instead of $x.s$. For example, "**require** *salary*" makes perhaps more sense than "**require** *salary.s*," which means that the item *salary* must be selected. To illustrate other syntactic extensions, we give below a set of formats for new rules and show immediately below how each could be written in the skeleton syntax given above.

1. **require** ⟨*condition1*⟩ **if** ⟨*condition2*⟩
   ≡ **require not** ⟨*condition2*⟩ **cor** ⟨*condition1*⟩
2. (Exclusion rule) **exclude** *v1*, *v2* **if** ⟨*condition*⟩
   ≡ **require not** (*v1.s* **or** *v2.s*) **if** ⟨*condition*⟩
3. (Limit rule) **limit is** 1 **of** *sex*
   ≡ **require** *card*(*sex*) ≤ 1
4. **require at least** 1 **of** *sex*
   ≡ **require** *card*(*sex*) ≥ 1
5. (Option limit rule) **with** ⟨*id*⟩: **allow only** 1 **options**
   ≡ **let** *card*(⟨*id*⟩) ≤ 1
6. (Implication rule) **with** ⟨*id*⟩: ⟨*condition*⟩ **implies** *red*, *blue*
   ≡ **let** ⟨*id*⟩ **be** ⟨*id*⟩ + [*red*] **if** ⟨*condition*⟩;
   let ⟨*id*⟩ **be** ⟨*id*⟩ + [*blue*] **if** ⟨*condition*⟩
7. (Option exclusion rule) **with** ⟨*id*⟩: ⟨*condition*⟩ **excludes** *blue*
   ≡ **let** ⟨*id*⟩ **be** ⟨*id*⟩ − [*blue*] **if** ⟨*condition*⟩

Another convenient extension is the *enter* behavior rule, which allows a procedure to be invoked directly without terminating the frame dialogue:

enter ⟨*id*⟩ **if** ⟨*condition*⟩
≡ invoke procedure ⟨*id*⟩ if ⟨*condition*⟩ is *true*;
Upon return from the procedure, resume the frame dialogue.

As can be seen, the language allows the programmer to say the same thing in many ways. The language designer must search for a minimum set of rules with maximum flexibility (if such a thing is possible). It is probably better to have just a few ways of describing behavior rules rather than too many.

7. A larger example

As an example of a dialogue program, suppose that a financial institution wishes to computerize some of its operations. We show below part of the application program.

The program is structured as a sequence of frames *F1*, *F2*, · · ·. Frame *F1*, shown in Fig. 3, allows the user to select one of the bank functions and, accordingly, to proceed to another frame to perform the desired function (New Account, Deposit, or Withdrawal). Frames *F2* and *F3* (Figs. 4 and 5) allow the user to enter the required information in order to open a new account and are typical of applications requiring the collection of a large amount of interrelated information, with very little processing.

- *Program*

```
program bank;
  var title: display text 'BANK OF NEW YORK';
  R: subframe
       return: attention (RETURN);
       rules terminate if return.s end
     end;
  a1: attention (OPEN ACCOUNT, DEPOSIT, WITHDRAWAL)
      at (1,8) text 'Select transaction desired:';
  d1: display text 'NEW ACCOUNT';
  heading: subframe
             inf: display at (4,3)
                    text 'Enter information. Hit ENTER
                    when done.';
             contains d1 at (2,19);
           end;
  name: key-in char at (0,6) text 'NAME:';
  status: menu set of (MARRIED, SINGLE) at (0,9)
          text 'STATUS:';
  accno: display at (0,24)
         text 'The new account No. is:';
frame F1; · · · endframe;
frame F2; · · · endframe;
           · · ·
begin
  start: F1;
  case a1 of
     NEW ACCOUNT: F2; F3; F4;
     DEPOSIT: F5;
     WITHDRAWAL: F6;
  end;
  go to start
end.
```

```
                   BANK OF NEW YORK

Select transaction desired:
       *OPEN ACCOUNT
       *DEPOSIT
       *WITHDRAWAL
```

Figure 3 Frame *F1*.

```
                   BANK OF NEW YORK

                     NEW ACCOUNT

 Enter information. Hit ENTER when done.
 NAME:
 PERMANENT ADDRESS:
 MAILING ADDRESS - Same as above?   *YES
    ENTER ADDRESS:                  *NO

 STATUS:  *MARRIED
          *SINGLE
```

Figure 4 Frame *F2*.

First, we declare global variables, which can be referenced in any of the frames. Subframe *R* will be used in frame *F4* (Fig. 6) to return control to frame *F1*; selecting the RETURN option would terminate the dialogue in frame *F4*, whereby the program would then cause frame *F1* to be presented.

Subframe *heading* is used to display text at the top of the screen in frames *F2* and *F3*. It consists of *d1*, which is also used in frame *F4*, and display item *inf*.

- *Definition of frames*

Frame *F1* is shown in Fig. 3. It is defined as follows:

```
frame F1;
  contains title at (0,17) intensity high;
           a1 at (2,2);
  rules terminate if a1.s
  end
endframe
```

This frame contains two global items: display item *title* and attention item *a1*, which is used to select the desired function to be performed. The **case** statement in the program is used to present the next frame, depending on the option selected.

Frame *F2* is shown in Fig. 4.

```
                BANK OF NEW YORK

                  NEW ACCOUNT

Enter information. Hit ENTER when done.
NAME:
SEL. SERVICE NO.:               SEX:   *MALE
                                       *FEMALE
NO. CHILDREN:
SALARY:
SPOUSE'S SALARY:
```

Figure 5 Frame *F3*.

```
                BANK OF NEW YORK

                  NEW ACCOUNT



NAME:

The new account No. is:

                                        *RETURN
```

Figure 6 Frame *F4*.

frame *F2*;
  var *addr1*: **key-in** *char* **at** (0,19)
      text 'PERMANENT ADDRESS:';
    d3: **display text** 'MAILING ADDRESS –';
    m1: **menu set of** (YES, NO) **at** (0,16)
      text 'Same as above?';
    addr2: **key-in** *char* **at** (0,15) **text**
      'ENTER ADDRESS:';
  contains *title* **at** (0,17); *heading*; *name* **at** (5,3);
      addr1 **at** (6,3); *d3* **at** (7,3); *m1* **at** (7,21);
      addr2 **at** (8,6); *status* **at** (10,3);
  rules require *name*;
    require at least 1 **of** *addr1*, *addr2*;
    require *addr1* **if** YES **in** *m1*;
    require *addr2* **if** NO **in** *m1*;
    with *m1*: **allow only** 1 **options**;
    exclude *addr2* **if** YES **in** *m1*;
    with *status*: **allow only** 1 **options**;
    terminate if ENT_KEY
  end;
  procedure *initialize*;
  begin *reset_frame*; *ml* := [YES]
  end
endframe

The global items *title*, *name*, and *status* and the subframe *heading* are included in this frame. One of the two addresses must be supplied, depending on whether YES or NO is selected. If neither option is in *m1*, then it is valid to enter either address or both. Actually, the rule **require** *addr1* is superfluous because it can be inferred from the other rules.

Upon entering the frame and before the frame is presented to the user, the procedure *reset_frame* causes the status of all items in *F2* to be set to unselected. Then, the item *m1* is initialized to [YES]. We have not defined the procedure *reset_frame* here; this could be a "built-in" procedure provided by the implementation.

Frame *F3* is shown in Fig. 5.

frame *F3*:
  var *sel_ser*: **key-in** *integer* **at** (0,18)
      text 'SEL. SERVICE NO.:' **format** I9;
    sex: **menu set of** (MALE, FEMALE) **at** (0,6) **text** 'SEX:';
    nchild: **key-in** *integer* **at** (0,14)
      text 'NO. CHILDREN:' **format** I10;
    salary: **key-in** *real* **at** (0,8) **text** 'SALARY:'
      format F20;
    salary2: **key-in** *real* **at** (0,17)
      text 'SPOUSE''S SALARY:' **format** F15;
  contains *title* **at** (0,17); *heading*; *name* **at** (5,3);
      sel_ser **at** (6,3); *sex* **at** (6,31); *nchild* **at** (8,3);
      salary **at** (9,3); *salary2* **at** (10,3);
  rules with *sex*: **allow only** 1 **options**;
    require *sex*, *salary*;
    require *nchild* $\geq$ 0 **if** MARRIED **in** *status*;
    let *name.e* **be** *false*;
    require *sel_ser* **if** MALE **in** *sex*;
    exclude *sel_ser* **if** FEMALE **in** *sex*;
    let *salary2.d* **be** MARRIED **in** *status*;
    require *salary2* **if** MARRIED **in** *status* **cand**
      salary < 10000;
    terminate if ENT_KEY
  end;
  procedure *initialize*;
  begin *reset_item*(*sel_ser*, *sex*, *nchild*, *salary*, *salary2*)
  end;
  procedure *terminate*;
  begin *accno* := *new_acc*; *file_acc*
  end
endframe

This frame is very similar to the example shown earlier in Fig. 1. As in frame *F2*, the items *title* and *heading* are included here. The items *status* and *accno* are referenced here but will not be displayed to the user. The item *name* contains the information supplied by the user in the previous frame, but it cannot be changed here. Initially, the procedure *reset_item* (not defined here) sets the status of all key-in and menu items (except *name*) to unselected.

The relations among the items are expressed by the behavior rules. For example, selective service number is only required from male applicants. To determine loan eligibility, the bank wants to know the spouse's salary if the applicant's salary is less than \$10 000. And *salary2* makes no sense if the applicant is not married; therefore, *salary2* only appears on the screen if MARRIED was selected in frame *F2*. When all these relations are satisfied, the user can depress the ENTER key, which causes a new account to be produced and filed by the local procedures *new_acc* and *file_acc* (not defined here).

Frame *F4* is shown in Fig. 6.

frame *F4*;
  contains *title* **at** (0,17); *dl* **at** (2,19); *name* **at** (7,3);
    accno **at** (9,3); *R* **at** (12,41);
  rules let *name.e* **be** *false*
  end
endframe

This frame is used to display the new account number to the user after all specified information has been collected in frames *F2* and *F3*. The item *name*, which appears in frames *F2* and *F3*, also appears here but at a different location. Selecting the RETURN option causes a return to the program, according to the **terminate** rule specified in subframe *R*; frame *F1* will then be displayed again.

8. Implementation problems

The efficient implementation of behavior rules may be a point of contention about the applicability of this work. In this section, therefore, we briefly discuss implementation problems and give hints at solutions. A more detailed explanation of these problems, and a discussion of techniques and algorithms to solve them, appear in the first author's thesis [11].

To begin with, a set of behavior rules may be inconsistent, and to check for this possibility the compiler should contain a compile-time analysis procedure. For example, suppose that the following rules are given to describe the frame behavior:

1. **require** *sel_ser* **if** MALE **in** *sex*;
2. **with** *sex*: **allow only** 1 **of** MALE, FEMALE;
3. **let** *sex* **be** *sex* + [FEMALE] **if** *sel_ser.s*

The problem is that selecting the MALE option leads to an inconsistency. Another situation that must be detected is when a termination condition can never occur or would lead to an inconsistency.

These kinds of problems can be analyzed with a graph model of interactions. A graph representation of the above rules is illustrated in Fig. 7, where the nodes of the graph represent items and options, and the edges represent constraints and dependencies. Associated with each node x is a *possible status* property *ps*, as follows:

If $node(x).ps = 1$, then x must be selected;
If $node(x).ps = 0$, then x cannot be selected;
If $node(x).ps = 2$, then x can be in either status.

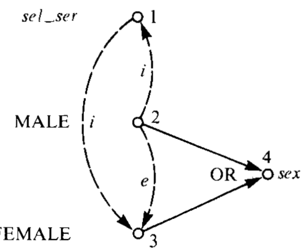

Figure 7 Graph representation of behavior rules.

Shown in the graph are implication links (labeled *i*) and exclusion links (labeled *e*), which express relations between the possible status of nodes. The graph is constructed as the rules are parsed. For example, the first rule above produces the link $2 \xrightarrow{i} 1$. A rule such as **require** x does not produce a link; it simply causes the possible status of node x to be set to 1. The specification of

require x; **exclude** x

would result in the inconsistency $node(x).ps = 1$ and $node(x).ps = 0$.

It is not always necessary to test conditions after every user action. Here, the graph model also helps, since it is possible to determine which user inputs have an effect on each system response (such as the display of a message or the termination of the dialogue). Furthermore, certain combinations of user's inputs are irrelevant in the sense that, although they may occur, they cause no change in the observable state of the system. A run-time procedure that takes advantage of these observations to implement behavior rules efficiently is contained in [11].

9. Summary

We have presented extensions to PASCAL to provide a language for programming alphanumeric user-computer dialogues. In this language, interactive behavior rules can be defined nonprocedurally by an application programmer, leaving the task of checking for inconsistencies and setting up the control logic to the system. This type of language, therefore, can reduce much complicated coding of interactive applications.

The extensions were designed to be natural and to blend well with the concepts and syntax of PASCAL. We feel that, in general, block-structured languages with flexible data-type definition facilities are good candidates for the embedding of interactive behavior specifications.

We saw, for example, how items were defined with a simple extension of PASCAL types, how subframes were simple extensions of PASCAL records, and how frames were defined in a way similar to PASCAL procedures. We

```
                  WEEKLY PAYROLL

 EMPLOYEE                    HOURS
   NAME                      WORKED     PAY

F.W. SMITH                    20
J.M. ROBERTS                  36
A.L. ANDERSON                 40
```

Figure 8 A table item.

also saw how the scope rules for variables in blocks and procedures extended very nicely to items in frames and subframes.

It is, of course, possible to extend other languages to allow the specification of interactive dialogues, but the extensions may not be as elegant. A possible way to extend COBOL is discussed in [11].

In the description of our language, we deliberately omitted some language features for the sake of simplicity. For example, in a real implementation, the language may be extended to contain abbreviations, more item and frame attributes, and more functions to manipulate table items. Also, a graphics protocol could be designed to lay out the information on the frames interactively.

Appendix A: Table item

High level programming languages normally contain data constructs such as arrays, structures, and records, which are built up from the basic data types (integer, real, character, etc.). Similarly, our language contains a structured type of item called a *table* item, which is made up of the basic items that we have considered so far. The table item allows the programmer to define a two-dimensional structure of values, with text, position, and other characteristics—the number of rows to be displayed, whether the user is to select rows or enter values in a column, etc.

A table item may be viewed as a set of *columns*; each column is defined very much like a key-in or display item (depending on whether or not the user is permitted to enter values into the column), having attributes such as position, format, initial value, etc. The programmer can also specify that the rows of the table are to behave as a menu or as an attention item. Behavior rules can then be specified to describe selection constraints and relations among the rows or columns of the table.

An example of a table item is shown in Fig. 8. Let us assume that the values contained in the first column can be obtained from a global table *emp_tab*, which was used by an authorized person to specify the employees' names. We then include this column in our new table and add two more columns. When this table is displayed, a clerk is to enter the hours worked each week, and the program will then compute the pay. The table can be defined as follows:

var *pay_tab*: **table** (20) **text** 'WEEKLY PAYROLL' **at** (0,9)
    columns
      hrs: **key-in** *integer* **at** (5,18)
          text 'HOURS' **at** (−3,−2), 'WORKED'
          initial 0 **format** I2;
      pay: **display** *real* **at** (5,25)
          text 'PAY' **at** (−2,1) **initial** 0 **format** F6.2;
      contains *emp_tab.name* **at** (5,2)
    end
    rows (10) **attention**

This defines a table item *pay_tab* having a maximum of 20 rows; of these, only 10 rows can be seen by the user. The rows are to behave as an attention item (having 10 options). To specify the behavior of this table, we give behavior rules for the containing frame:

let *pay_tab.name.e* **be** *false*;
require *hrs* $\leq$ 40;
let *pay* **be** *hrs* $*$ *rate*

The first rule prevents the user from changing the values in the *name* column. The pay is computed automatically by multiplying the hours worked (which cannot exceed 40) by the global variable *rate*.

Thus, a table item may be used for displaying or updating information, depending on the ⟨*item kind*⟩ specified for the columns (and their status properties). If used for updating information, then, at any time during the presentation of the containing frame, one of the rows may be designated as the *current row*. Each value in the current row behaves as a display or key-in item, according to the ⟨*item kind*⟩ specified for the corresponding column. Only the current row can be acted on by the user.

The programmer can manipulate the information in the table item, change the status properties of the columns, select rows of the table, or designate another row as the current row. For this we need some notation.

Notation If T is a table item, the rows are denoted by $T[1]$, $T[2]$, $\cdots$, and, in particular, *T.current_row* is the current row. The variable *T.current_index* is the *current-row index*, i.e., *T.current_row* is equivalent to T[*T.current_index*]. To reference elements in each row, we use PASCAL record notation; thus,

$T[i].x$ = the value at row i, column named x.

Now, the table item T itself has a value, which is the value that results from treating the rows as a menu or attention item. For example, if **menu** is specified for the rows of T and rows 1 and 3 are the only selected rows, then the value of T is the set {1, 3}, *T.s* and 1 **in** T are *true*.

To illustrate, suppose that the user can enter the "hours worked" in any of the rows of the item *pay_tab* defined above. But, prior to entering the information, he must indicate which row he wishes to act on by pointing to it with the light pen. This can be accomplished by adding the behavior rule

let *pay_tab.current_index* **be** *pay_tab*

which specifies that the value of the table item (the selected row) is to be assigned to the variable *pay_tab.current_index*.

Alternatively, we could define key number 1 as the "next key," so that pressing key 1 would compute the pay for the current row and place the cursor at the next row. This can be programmed as follows:

```
var next_key: attention (KEY1);
        ⋮
enter pl if next_key.s;
        ⋮
procedure pl;
begin pay_tab.current_row.pay :=
      pay_tab.current_row.hrs * rate;
      pay_tab.current_index :=
      pay_tab.current_index + 1
end
```

More flexibility may be needed to manipulate rows of a table item. For example, the following built-in procedures would be helpful:

nrows(T)	gives the number of rows defined so far,
delete(T, i)	deletes row i from the table,
insert(T, i)	inserts a row (of undefined values) after row i.

Moreover, in order to increase the usefulness of table items, the language should provide a mechanism for associating external files with a table item, and it should provide operators for transferring data. We explore these ideas briefly in the next section.

Associating a file with a table item

We are particularly interested in sequential files of type *record*. We call these *structured sequential files* and define them as follows:

⟨*structured sequential file*⟩ ::= **file of** ⟨*record type*⟩

Exactly how the components are allocated in external storage is implementation-dependent.

By specifying the attribute **file** (input or output) in the declaration of a table item, the programmer establishes an association between the table item T and records of the specified file. In this case, we say that the table item is *bound* to the file.

What we wish to accomplish is, in fact, a generalization of the binding of files to key-in or display items, which was discussed earlier. In this case, a table item T bound to a file f would be regarded as a work area into which one could read records from f (if f is an input file) or write records to f (if f is an output file). Superimposed on this work area is a window through which the user can view the area. The size of this window, say w, is specified in the row specification part of the declaration of T, and represents the number of rows that the user sees. Let p be the current number of rows and s the size of the table (maximum number of rows). The programmer can manipulate values in rows $T[1]$, $T[2]$, $\cdots$, $T[p]$, while the user only has access to w or p rows, whichever is smaller.

The implementation may provide facilities for scrolling, for reading (writing) records from (to) a file, and for associating the file information with the table item. The operators for reading and writing would be generalizations of the PASCAL operators *get*(f) and *put*(f). We omit the details and refer the reader to [11] for more information.

Appendix B: Syntax

We present here a summary of our syntactic extensions to PASCAL.

⟨*frame definition part*⟩ ::= ⟨*frame definition*⟩ {; ⟨*frame definition*⟩};

⟨*frame definition*⟩ ::= **frame** ⟨*id*⟩: ⟨*frame body*⟩ **endframe**

⟨*frame body*⟩ ::= ⟨*label declaration part*⟩
⟨*constant definition part*⟩
⟨*type definition part*⟩
⟨*variable declaration part*⟩
⟨*item inclusion part*⟩
⟨*behavior rules*⟩
⟨*procedure declaration part*⟩

⟨*item type*⟩ ::= ⟨*item kind*⟩ [⟨*type*⟩ [**at** (⟨*row*⟩, ⟨*col*⟩)]] ⟨*attribute specification*⟩ | ⟨*table item type*⟩

⟨*item kind*⟩ ::= **display** | **key-in** | **menu** | **attention**

⟨*attribute specification*⟩ ::= [⟨*text specification*⟩] {⟨*item attribute*⟩}

⟨*text specification*⟩ ::= **text** ⟨*string*⟩ [**at** (⟨*row*⟩, ⟨*col*⟩)] {, ⟨*string*⟩ [**at** (⟨*row*⟩, ⟨*col*⟩)]}

⟨*type*⟩ ::= ⟨PASCAL *type*⟩ | ⟨*subrange set*⟩ | ⟨*value set*⟩

⟨PASCAL *type*⟩ ::= a PASCAL standard type, scalar type, or set type

⟨*subrange set*⟩ ::= ⟨*constant*⟩ .. ⟨*constant*⟩ {, ⟨*constant*⟩ .. ⟨*constant*⟩}

⟨*value set*⟩ ::= (⟨*constant*⟩ {, ⟨*constant*⟩})

⟨*table item type*⟩ ::= **table** (⟨*size*⟩)
⟨*attribute specification*⟩
⟨*column specification part*⟩
⟨*row specification part*⟩

⟨*column specification part*⟩ ::= **columns** {⟨*id*⟩: ⟨*item type*⟩;}
[⟨*item inclusion part*⟩] **end**

⟨*row specification part*⟩ ::= **rows** (⟨*size*⟩) [⟨*row kind*⟩]

⟨*row kind*⟩ ::= **menu** | **attention**

⟨*subframe type*⟩ ::= **subframe** {⟨*id*⟩: ⟨*type1*⟩;}
[⟨*item inclusion part*⟩]
[⟨*behavior rules*⟩] **end**

⟨*type1*⟩ ::= ⟨*subframe type*⟩ | ⟨*item type*⟩

⟨*item inclusion part*⟩ ::= **contains** ⟨*item specification*⟩
{; ⟨*item specification*⟩}

⟨*item specification*⟩ ::= ⟨*id*⟩ {, ⟨*id*⟩ } {⟨*frame attribute*⟩}

⟨*frame attribute*⟩ ::= **at** (⟨*row*⟩, ⟨*col*⟩)
| **intensity high**
| **format** ⟨*format specification*⟩
| **format msg** ⟨*string*⟩

⟨*behavior rules*⟩ ::= **rules** ⟨*rule*⟩ {; ⟨*rule*⟩} **end**

⟨*rule*⟩ ::= ⟨*requirement rule*⟩ | ⟨*binding rule*⟩
| ⟨*terminate rule*⟩ | ⟨*enter rule*⟩

⟨*requirement rule*⟩ ::= **require** ⟨*condition1*⟩ [**if** ⟨*condition*⟩]

⟨*condition1*⟩ ::= ⟨*condition*⟩ | ⟨*name part*⟩

⟨*name part*⟩ ::= ⟨*item name*⟩ {, ⟨*item name*⟩}

⟨*binding rule*⟩ ::= **let** ⟨*variable*⟩ **be** ⟨*expression*⟩
[**if** ⟨*condition*⟩]

⟨*terminate rule*⟩ ::= **terminate if** ⟨*condition*⟩

⟨*enter rule*⟩ ::= **enter** ⟨*id*⟩ **if** ⟨*condition*⟩

⟨*condition*⟩ ::= ⟨*basic condition*⟩
| ⟨*condition*⟩ **cor** ⟨*basic condition*⟩

⟨*basic condition*⟩ ::= ⟨*logical expr*⟩ | ⟨*basic condition*⟩
cand ⟨*logical expr*⟩

⟨*logical expr*⟩ ::= a logical expression
| (⟨*condition*⟩) | ENT_KEY

References and note

1. S. L. Feingold, "PLANIT—A Flexible Language Designed for Computer-Human Interaction," *AFIPS Conf. Proc.* **31**, 545 (1967).
2. "COURSEWRITER III Student Text," Form no. GC20-1706, IBM Corporation, White Plains, NY, 1969.
3. C. I. Johnson, "Principles of Interactive Systems," *IBM Syst. J.* **7**, 147 (1968).
4. I. W. Cotton, "Languages for Graphic Attention-Handling," *Advanced Computer Graphics*, R. D. Parslo and R. E. Green, eds., Plenum Press, New York, 1971, p. 1049.
5. W. M. Newman, "A System for Interactive Graphical Programming," *AFIPS Conf. Proc.* **32**, 47 (1968).
6. T. R. Stack and S. T. Walker, "AIDS—Advanced Interactive Display System," *AFIPS Conf. Proc.* **38**, 113 (1971).
7. J. M. Martin, "Design of Man-Computer Dialogues," Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.
8. K. Jensen and N. Wirth, "PASCAL User Manual and Report," *Lecture Notes in Computer Science* **18** (second edition), Springer-Verlag, New York, 1975.
9. N. Wirth, "The Programming Language PASCAL," *Acta Informatica* **1**, 35 (1971).
10. The language facilities and techniques described in this paper are experimental; an implementation is not planned by the IBM Corporation.
11. J. M. Lafuente, "The Specification of Data-Directed Interactive User-Computer Dialogues," Ph.D. Thesis, Cornell University, Ithaca, NY, 1977.
12. E. D. Homer, "An Algorithm for Selecting and Sequencing Statements as a Basis for a Problem-Oriented Programming System," *Proceedings of the 21st ACM National Conference*, New York, NY, 1966, p. 305.
13. L. G. Tesler and H. J. Enea, "A Language Design for Concurrent Processes," *AFIPS Conf. Proc.* **32**, 403 (1968).
14. J. M. Foster and E. W. Elcock, "ABSYS1: An Incremental Compiler for Assertions: An Introduction," *Machine Intelligence* **4**, 423 (1969).

Received May 25, 1977; revised September 15, 1977

J. M. Lafuente is located at the IBM System Products Division laboratory, P.O. Box 390, Poughkeepsie, New York 12602; Prof. Gries is located at the Department of Computer Science, Cornell University, Ithaca, New York 14850.