To: Distribution

From: J. Falksen

Date: January 18, 1977

Subject: Report Generator Language

INTRODUCTION

The Report Generator Language (RGL) is a lanugage to describe reports which are to be created. The result of the compilation is a report command. Executing it then causes the desired reports to be created.

The report command can have parameters if needed.

Please send comments or suggestions to:
Falksenj.Multics
on System M, or to
J. Falksen, HIS, K-28
P O Box 6000
Phoenix AZ 85005

Overview

Report Generator Language is a programming language designed to aid a person in producing formatted reports. A single report command can produce one or more reports. The destination and even existance of any of the reports may be varied according to conditions.

A report is basically described in 3 parts; 1) what is the input like, 2) what is the output like, 3) what phases of processing are necessary.

Input can have fixed or variable fields; under certain circumstances fields may be optional.

Output is broken into pages of user defined length. Reports can have headers and footers, that is, data which appears before and after the pages of formatted data. Pages can have headers and footers, that is, data which appears at the top and bottom of each page.

The data can be broken into blocks via control breaks. One or more fields can be designated as control fields, the order of specification being the priority of the break. A control break occurs when the data in a control field differ from the data in the same control field from a previous record. When a break on a certain level occurs, this forces a break at all lower levels.

Each break level can have a header and footer, that is, data which appears before and after each block.

Each record of input can produce one or more lines of data in one or more reports.

The processing to be done on the data is described in one or more phases. The final phase ends with a PRINT request. All other phases will end with a HOLD or HOLD/SORT request. A phase represents one pass thru the data.

The simplest form of report will just do a PRINT. A more complex form might pass thru the data once, accumulating totals of field X, Y, and Z. Then a second pass thru the data would print the record, including X, Y, and Z, and indicating what percentage each of them is of the total.

Or the data may only need to be sorted and then printed.

Here is an example of a very simple, unconditional report. It sends its output to user_output and uses all default field sizes.

```
DECLARE 1 INPUT ,
        2 country_name CHARACTER DELIMITED ",",
        2 capital CHARACTER DELIMITED ",",
        2 time_zone CHARACTER (4);

DEFINE  1 REPORT cct,
        2 DETAIL d1,
          3 LINE,
            4 country_name,
            4 capital,
            4 time_zone;

PRINT d1;

END;
```

Now this turns out to be a little too simple; you don't like the way it looks. The fields are varying in length and so the report was not columnar. You then can be more specific about the format.

```
DECLARE 1 INPUT ,
        2 country_name CHARACTER DELIMITED ",",
        2 capital CHARACTER DELIMITED ",",
        2 time_zone CHARACTER (4);

DEFINE  1 REPORT cct PAGELENGTH 66
                MINLINE 6
                MAXLINE 60
                ON FILE "w_capitals",
        2 DETAIL d1,
          3 LINE,
            4 country_name            COLUMN 1,
            4 capital                 COLUMN 20,
            4 time_zone               COLUMN 40;

PRINT d1;

END;
```

The next example adds sorting and a grand total. This is to print a report of all the countries which are aligned with the

United States, indicating their military strength.

```
-------------------------------------------------------------------
  DECLARE 1 INPUT RECORD 280 FILE ">udd>WORLD>wpmf",
          2 country CHARACTER (20) POSITION 200,
          2 cdr_in_chief CHARACTER (24),
          2 mil_no_men CHARACTER (6),
          2 aligned CHARACTER (20) POSITION 12;

  DEFINE  1 REPORT good_guys ON SWITCH "military_",
          2 REPORTHEAD,
            3 LINE 24,
              4 "ALIGNED MANPOWER" CENTER,
            3 LINE +2,
              4 (%SUBSTRING(%MMDDYY,5,2)
                 || %MONTH
                 || %SUBSTRING(%%MMDDYY,3,2)) CENTER,
              3 LINE +5,
          2 DETAIL d1,
            3 LINE,
              4 cdr_in_chief,
              4 mil_no_men
                  LET (total := total + mil_no_men;),
          2 REPORTFOOT,
            3 LINE +2
              4 "Total number of men:",
              4 total  COLUMN 30 PICTURE "zzz,zzz,zz9";

  IF (aligned EQ "USA")
  HOLD country,cdr_in_chief, mil_no_men
  SORT country, cdr_in_chief;

  PRINT d1;

  END;
-------------------------------------------------------------------
```

Sometimes information is to be printed only when it changes. This is called control break processing.

This report gives the largest cities for all the countries in the specified input file. Only cities with population greater than 1,000,000 will be listed. Also, only the first 25 for any country will be listed. The name of the country will be printed when the country begins and at the beginning of a new page. The sort includes the capital indicator in such a way that the capital always sorts first, regardless of the population. The first line for a country will contain the capital name.

```
-----------------------------------------------------------------
  DECLARE 1 PARAMETER,
          2 cfile CHARACTER (*);

  DECLARE 1 INPUT FILE ">udd>WORLD>cities" STREAM,
          2 capital CHARACTER (1), /* "*" if capital */
                                   /* " " if not capital */
          2 country CHARACTER DELIMITED ",",
          2 city CHARACTER DELIMITED ",",
          2 altitute CHARACTER DELIMITED ",",
          2 population CHARACTER DELIMITED ",";

  DECLARE city_ct FIXED;
  DECLARE newpage BOOLEAN;

  DEFINE 1 REPORT key_cities ON FILE cfile
                   BREAK (country),
          2 PAGEHEAD,
            3 LINE 1,
              4 "" LET (newpage:=TRUE;),
          2 DETAIL d1,
            3 LINE +3 IF (%LEVEL(country)
                           OR (newpage AND (city_ct LT 25))),
              4 country COLUMN 1
                      LET (newpage:=FALSE;),
              4 "CAPITAL:  " IF (%LEVEL(1)) COLUMN 25
                      LET (city_ct := 0;),
              4 city IF (%LEVEL(1)),
              4 population COLUMN 60 IF (%LEVEL(1)),
            3 LINE IF (NOT %LEVEL(1) AND (city_ct LT 25)),
              4 city COLUMN 10 LET (city_ct:=city_ct+1;),
              4 population COLUMN 60 PICTURE "zzz,zzz,zz9";

  IF (population GT 1000000)
  HOLD city
  SORT country,capital DESCENDING,population DESCENDING;

  PRINT d1;

  END;
-----------------------------------------------------------------
```

Here is an example of two reports which are produced in parallel but which end up in one file serially. The first is a table of contents with roman numeral page numbering; the second is the salary listing by department, with arabic pagenumbers, and showing the average salary of the people working for each manager.

```
-----------------------------------------------------------------
  DECLARE 1 PARAMETER,
          2 xxxxx CHARACTER (*);        /* output name */

  DECLARE 1 INPUT RECORD 390 FILE ">udd>EMPREC>master>emp1",
          2 salary CHARACTER (9) POSITION 230,
          2 name CHARACTER (28) POSITION (1),
          2 mgr CHARACTER (28) POSITION 350,
          2 dept CHAR(6);

  DECLARE count FIXED;
  DECLARE saltot FLOAT;

  DECLARE 1 REPORT body ON FILE xxxxx NUMBER 2
                  BREAK (dept, mgr) MAXLINE 58,
          2 DETAILHEAD dept,
            3 LINE 0,         /* start a new page */
          2 DETAILHEAD mgr,
            3 LINE +3,
              4 ("***** " || dept || " *****"),
              4 mgr column 30,
          2 DETAIL db,
            3 LINE
              4 name          LET(count:=count+1;),
              4 salary        PICTURE "zz,zz9.00"
                  LET(saltot:=saltot+salary;),
          2 DETAILFOOT mgr,
            3 LINE +2,
              4 "Average salay of these ",
              4 count,
              4 " employees is ",
              4 saltot PIC "zz,zz9.99"
                  LET (saltot:=saltot/count;)
              4 "." LET(count:=0;saltot:=0;),
          2 PAGEFOOT,
            3 LINE 60,
              4 %PAGENUMBER() CENTER;

  DEFINE 1 REPORT toc ON FILE xxxxx NUMBER 1
                  BREAK(dept,mgr) MAXLINE 58,
          2 REPORTHEAD,
            3 LINE 10,
              4 "CONTENTS" CENTER,
            3 LINE +6,
              4 "Page" RIGHT,
            3 LINE +1,
          2 DETAILHEAD dept,
            3 LINE +2 MAXLINE 55,
              4 "DEPARTMENT",
              4 dept,
            3 LINE,
          2 DETAILHEAD mgr,
            3 LINE,
              4 %REPEAT(" .",32) CENTER,
              4 mgr RIGHT,
              4 %PAGENUMBER(body)  RIGHT,
          2 DETAIL dt,
            3 LINE +0,        /* don't want any detail data */
          2 PAGEFOOT,
            3 LINE 60,
              4 %MMDDYY RIGHT,
              4 %ROMAN(%PAGENUMBER()) CENTER;

  HOLD salary
  SORT dept, mgr, name ;
  PRINT db;
  PRINT dt;
  END;
-----------------------------------------------------------------
```

Following there are parts which are written in BNF. This BNF is very simple, there are few "features". There are two operators:

::=	means "is defined as"
!	means "end of rule", anything following is commentary

So a rule will have the general form

```
        <name> ::= <things>  ! comment
```

An "OR" condition is indicated by multiple definitions of the same rule name.

```
  <lik>  ::= <integer>  ! must be 0 - 9
  <lik>  ::= *  !
```

This says that the rule "lik" is defined as either an integer in the range 0 thru 9, or the literal "*".

Rule names are enclosed within "<" and ">"; they are constructed with a few conventions. These conventions will aid the reader by giving him some information about the rule being referenced without looking at the rule definition. The forms are these:

```
        <xxx>  <xxx...>  <xxx , ...>  <[xxx]>  <[xxx...]>
```

<xxx>	means that the rule describes an xxx
<xxx...>	means that the rule describes a list of xxx's
<xxx , ...>	means that the rule describes a list of xxx's separated by commas
<[xxx]>	means that the rule describes an optional xxx
<[xxx...]>	means that the rule describes an optional list of xxx's

Words in capital letters represent keywords. The grammar will never show alternative forms of keywords. These will be shown as a separate list of abbreviations.

Below only the <xxx> forms will be defined.

Keywords are reserved.

The input to RGL is a Multics segment. It will be a segment whose name is zzz.rgl. If an object segment is kept from the compilation, it will be a report command or I/O appendage named zzz.

To use the kept object as a command you would say:

```
          zzz arg ...
```

To use the kept object as an I/O appendage you would say:

```
          io_call attach STR report_ zzz arg ...
          io_call open STR appropriate_output_mode
```

In order to isolate report generation from all knowledge of I/O switches. The "report_" I/O switch will be the only interface between reports and switches. The report command will act as an "I/O appendage" to the report_ switch. All report processing will be done in the appendage; all knowledge of stream and switch mechanisms will be in report_. report_ will implement open, put_chars, write_record, and close. Opening modes will be stream_output or sequential_output.

The I/O appendage is similar to the GSP of the grahpics system.

The input segment has this general form:

```
------------------------------------------------------------------
  <input>              ::= <dcl...>
                           <report...>
                           <exec...>
                           END ;  !
------------------------------------------------------------------
```

Data Declarations

The first thing in the source is the declaration section. All names must be declared before use.

There are 3 kinds of declarations:

```
                     parameter, input, data
```

Parameter Declaration

If the report command needs parameters, this form of declaration must be used. There can be positional and/or keyword parameters.

```
--------------------------------------------------------------------
  <dcl>                ::= DECLARE 1 PARAMETER <parm...> ;  !

  <parm>               ::= , 2 <identifier> <parm_spec...>  !

  <parm_spec>          ::= CHARACTER ( * )  !
  <parm_spec>          ::= CHARACTER ( <number> )  !
  <parm_spec>          ::= BOOLEAN  !

  <parm_spec>          ::= DEFAULT <quoted_str>  !;
  <parm_spec>          ::= KEY ( <keyword , ...> )  !

  <keyword>            ::= <quoted_str>  !
--------------------------------------------------------------------
```

The first 3 <parm_spec>'s are the data type. One must be specified.

CHARACTER (*) means that the length will be that of the parameter given to the report command.

CHARACTER (<number>) means that the length will be that specified. The parameter given to the report command will be truncated or padded as needed.

BOOLEAN means that the keyword is either absent or present. It has the value TRUE or FALSE.

DEFAULT is optional. If it is specified for a positional parameter, then that parameter is optional. Otherwise, the report command will give an "EXPECTED ARGUMENT MISSING" message if the parameter is not supplied. This cannot apply to a BOOLEAN parameter.

KEY is optional. If it is specified, then the parameter is a keyword parameter. All keywords must begin with a "-". When multiple keywords are specified, they represent alternate ways of supplying the keyword. This must be supplied if the parameter is BOOLEAN. If the parameter is not BOOLEAN, then the parameter following the keyword is the value of this parameter.

Examples:

```
  DECLARE 1 PARAMETER,
            2 in_file  CHARACTER(*),
            2 out_file CHARACTER(*),
            2 brief    BOOLEAN KEY ("-brief","-bf");

  DECLARE 1 PARAMETER,
            2 stream   CHARACTER(*) KEY ("-os")
                    DEFAULT "user_output";
```

When calling the report command, the positional parameters need

not be first. The first non-keyword is the first positional parameter, etc.
These parameter lists are equivalent:

```
          from_s to_s -bf
          from_s -bf to_s
          -bf from_s to_s
```

Input Declaration

The input to be used must always be specified. It can be from a file or I/O switch.

```
------------------------------------------------------------------
  <dcl>                ::= DECLARE 1 INPUT <[input_ctl...]>
                                           <ifld...> ; !
  <input_ctl>          ::= RECORD <number>  !;
  <input_ctl>          ::= RECORD !;
  <input_ctl>          ::= STREAM  !

  <input_ctl>          ::= FILE <cexp>  !;
  <input_ctl>          ::= ATTACH <cexp>  !

  <ifld>               ::= , 2 <identifier> <idcl_spec>  !
  <idcl_spec>          ::= CHARACTER ( <number> ) POSITION <number>  !;
  <idcl_spec>          ::= CHARACTER ( <number> ) OPTIONAL  !
  <idcl_spec>          ::= CHARACTER DELIMITED <quoted_str>  !;
  <idcl_spec>          ::= CHARACTER DELIMITED <quoted_str> OPTIONAL  !
  <idcl_spec>          ::= FIXED  !;
  <idcl_spec>          ::= FIXED ( <number> )  !
  <ifld>               ::= , 2 FILL ( <number> )  !
  <ifld>               ::= , 2 FILL ( <number> , <number> )  !
------------------------------------------------------------------
```

RECORD

The input is made up of fixed length records which have a specified byte-length.

STREAM

The input is an ASCII file with records separated by NL characters. This is the default.

FILE

The source will be gotten by directly referencing the segment <cexp>, a relative pathname.

ATTACH

The source will be gotten by attaching a switch. <cexp> is the attach description which is used.

If neither FILE nor ATTACH is given, then the report cannot be run as a command. When the report is run as an I/O appendage, the FILE/ATTACH specification is ignored.

CHARACTER

CHARACTER (<number>) means that the specified number of bytes will be taken as a character string.

POSITION <number> means the field will begin at a specified byte location in the record. If not specified, the default is to begin at the byte following the previously declared field.

OPTIONAL means that the field may not exist. If the record ends before this field, it is not an error. This obviously can apply to STREAM input only.

CHARACTER DELIMITED

DELIMITED means that the field is a string delimited by a character string. The delimiter can be more than one character long.

OPTIONAL means that the field may not exist. This can happen one of two ways. Either the record runs out before the field is reached, or a delimiter is encountered which is the delimiter of a later field.

FIXED

FIXED means that the data is binary data. Can only be used with RECORD. This field will occupy 4 bytes of the record.

FIXED (<number>) means that the data is not a word long. <number> is the precision of the field. It will occupy <number>+1 bits starting ar the current bit.

FILL

FILL (<n1>) means that <n1> bytes of record are to be skipped.

FILL (<n1>,<n2>) means that <n1> bytes and <n2> bits are to be skipped.

Data Declarations

Any data which is used within the report command must be declared. Various data types are available.

```
-----------------------------------------------------------------
  <dcl>               ::= DECLARE <identifier> <dcl_spec> ;  !

  <dcl_spec>          ::= FIXED  !;
  <dcl_spec>          ::= FLOAT  !;
  <dcl_spec>          ::= PICTURE <quoted_str>  !
  <dcl_spec>          ::= EDIT <quoted_str>  !
  <dcl_spec>          ::= CHARACTER ( <number> )  !
  <dcl_spec>          ::= CHARACTER ( <number> ) VARYING  !
  <dcl_spec>          ::= BOOLEAN  !
  <dcl_spec>          ::= RETURNS ( <dcl_spec> )  !;
  <dcl_spec>          ::= TABLE ( <init> )  !;
  <dcl_spec>          ::= TABLE ( <init> ) VARYING  !
-----------------------------------------------------------------
```

FIXED

This is like PL/I fixed bin.

FLOAT

This is like PL/I float dec.

PICTURE

This is a character string, with conversion specified when assigning to it. Like PL/I pictured data.

EDIT

This is a character string, with conversion specified when assigning to it. This is somewhat like an ioa_ control string, with these controls:

```
   ^nc   insert n characters from the sending field
   ^nx   insert n spaces
   ^^    insert a ^ character
   ^nf   move forward n characters in sending field
   ^nb   move backward n characters in sending field
   ^b    move to beginning of sending field
```

For example: you could edit a phone number with

```
   EDIT "(^3c)^3c-^4c"
```

CHARACTER

This is like PL/I character.

CHARACTER VARYING

This is like PL/I character varying.

BOOLEAN

This is a binary-valued element. It can contain TRUE and FALSE. It is similar to PL/I bit(1) aligned.

RETURNS

This specifies a function. The attribute can be any listed proir to this point. This is like PL/I

```
       entry options(variable) returns(xxx).
```

TABLE

This specifies an element-to-element transformation which is to be done. All elements of the table are of a fixed length, this being the maximum of the values specified.

TABLE VARYING

This specifies an element-to-element transformation which is to be done. Each element has whatever length it is initialized to. There are 4 kinds of transformations which can be done.

```
-------------------------------------------------------------------
  <init>                 ::= <initaa...>  !
  <initaa>               ::= <number> -> <number>  !
-------------------------------------------------------------------
```

An arithmetic to arithmetic transformation can be requested. For example:

```
  TABLE ( 6->1  4 -> 2  1->3 )
```

```
-------------------------------------------------------------------
  <init>                 ::= <inital...>  !
  <inital>               ::= <number> -> <quoted_str>  !
-------------------------------------------------------------------
```

An arithmetic to character transformation can be requested. For example:

```
  TABLE ( 1->"first" 2->"second" )
```

```
-------------------------------------------------------------------
  <init>                 ::= <initla...>  !
  <initla>               ::= <quoted_str> -> <number>  !
-------------------------------------------------------------------
```

A character to arithmetic transformation can be requested. For example:

```
  TABLE ( "JAN"->1 "FEB"->2 "MAR"->3 )
```

```
----------------------------------------------------------------

  <init>                ::= <initll...>  !
  <initll>              ::= <quoted_str> -> <quoted_str>  !
----------------------------------------------------------------
```

A character to character transformation can be requested. For example:

```
  TABLE ( "CPL"->"Corporal" "PVT"->"Private") VARYING
```

Report Definition

There must be at least one report defined. A report definition looks very much like a PL/I structure, with various line groups specified at level 2.

A report can be made up of up to 7 kinds of line groups. The line groups are ordered as they are to appear, i.e. headings before details.

```
----------------------------------------------------------------
  <report>              ::= DEFINE 1 REPORT <identifier>
                        <[report_ctl...]>
                        <[heading...]>
                        <detail...>
                        <[footing...]>
                        ;  !
  <report_ctl>          ::= PAGEWIDTH <number>  !;
  <report_ctl>          ::= PAGELENGTH <number>  !;
  <report_ctl>          ::= MINLINE <number>  !;
  <report_ctl>          ::= MAXLINE <number>  !;
  <report_ctl>          ::= BREAK ( <identifier , ...> )  !;
  <report_ctl>          ::= ON <output_sel>  !;
  <report_ctl>          ::= ON ( <output_sel_OR> )  !

  <output_sel_OR>       ::= <output_sel>  !;
  <output_sel_OR>       ::= <output_sel_OR>  IF ( <expr> )
                                              OR <output_sel>  !
  <output_sel>          ::= FILE <cexp>  !;
  <output_sel>          ::= FILE <cexp> NUMBER <number>  !;
  <output_sel>          ::= SWITCH <cexp>  !
----------------------------------------------------------------
```

PAGEWIDTH

This specifies the maximum print position which can be used in the report. The default is 65.

PAGELENGTH

This specifies the number of lines on a page. The default is 66.

MINLINE

This specifies the first line on a page to be used. The default is 0, however the PAGEHEAD can cause other action.

MAXLINE

This specifies the last line on a page which can be used. The default is PAGELENGTH, however the DETAIL and PAGEFOOT can cause other action.

BREAK

This specifies that control breaks are desired. If this is specified, then DETAILHEAD and DETAILFOOT can be used. The fields specified are the control fields. They are listed in decreasing order. %LEVEL(1) refers to the first field, %LEVEL(2) the second, etc. The default is no breaks.

ON

This specifies where the report is to go. It can be written to an I/O switch or placed into a segment (MSF). The default is SWITCH "user_output". If the destination is a FILE, then it is assumed that it is for a printer. If the destination is a STREAM, then it is assumed that it is for a terminal.

Alternatives may be specified for the destination. All but the last of the alternatives will contain an IF clause. These tests will be made in the order specified and the first one which is true will be taken. If all are false then the last (non-conditional) is taken.

Example:
```
  ON SWITCH in_file || "_str_"
  ON ( FILE "<<monday>a" IF (%DAY EQ "Monday")
    OR SWITCH "discard_output_")
  ON FILE ">udd>m>WORLD>" || infile
```

Sometimes it is necessary to produce more than one part of a report simultaneously, but to want them to end up in one file serially. A report preceeded by a table of contents is one such application. To accomodate this, the FILE may specify the NUMBER option. The numbers specified determine the order they finally are put into the named file. The filename must be precisely identical in its definition in order for the match to be made.

Example:
```
  DEFINE 1 REPORT body ON FILE "monthly" NUMBER 2 ... ;
  DEFINE 1 REPORT toc  ON FILE "monthly" NUMBER 1 ... ;
```

Headings

Headings are placed before the detail information. There are 3 possible kinds.

```
------------------------------------------------------------------
  <heading>            ::= , 2 REPORTHEAD <lines...>  !
  <heading>            ::= , 2 PAGEHEAD <lines...>  !
  <heading>            ::= , 2 DETAILHEAD <identifier>
                                <[detail_ctl...]> <lines...>  !
------------------------------------------------------------------
```

REPORTHEAD

This is a group of lines which precede the report. It is processed only once, the first time a detail for the report is printed. If REPORTHEAD appears before PAGEHEAD, then it will be like a cover page.

PAGEHEAD

This is a group of lines which are to appear at the top of each page. The first LINE of this group should contain an absolute line number.

DETAILHEAD

This is a group of lines which are to appear whenever a control break occurs on the specified <identifier>. DETAILHEAD cannot appear unless the BREAK <report_ctl> has been specified. This will not occur on the %LEVEL(0) break. Each level of break can have its own DETAILHEAD.

There can be multiple occurances of each kind of heading. This means that interleaving of parts can occur.
For example:

```
  2 PAGEHEAD,
    3 LINE 7,
      4 %MMDDYY COLUMN 70,
  2 DETAILHEAD alpha,
    3 LINE 8,
      4 "GROUP",
      4 alpha,
 2 PAGEHEAD,
    3 LINE +1,
      4 %PAGENUMBER() COLUMN 70 PICTURE "ZZ,ZZ9",
```

Details

The detail group is the main body of most reports.

```
-----------------------------------------------------------------
  <detail>              ::= , 2 DETAIL <identifier> <[detail_ctl...]>
                                      <lines...>  !
  <detail_ctl>          ::= IF ( <expr> )  !
  <detail_ctl>          ::= MAXLINE <number>  !;
  <detail_ctl>          ::= FIT  !
-----------------------------------------------------------------
```

The name of the detail is for reference by the PRINT statement.

IF

This optional specification causes <expr> to be evaluated. If it is TRUE then the detail will be processed.

MAXLINE

This optional specification indicates the last line of a page on which the group can start. If the current line is greater than this, then a new page is started.

FIT

This optional specification is used when the result of the group can vary in size, but the whole group must stay together on the page. When this is specified, first %FIT is set to TRUE and the group is tentatively processed. If it will fit, then it becomes part of the page. If it will not fit, then a new page is started, %FIT is set to FALSE and the group is processed again. If the group does not fit this time, it is just split when the bottom of the page is reached (as if the FIT option were not given).

Footings

Footings are placed after the detail information. There are 3 possible kinds.

```
-----------------------------------------------------------------
  <footing>             ::= , 2 DETAILFOOT <identifier>
                                 <[detail_ctl...]> <lines...>  !
  <footing>             ::= , 2 PAGEFOOT <lines...>  !
  <footing>             ::= , 2 REPORTFOOT <lines...>  !
-----------------------------------------------------------------
```

DETAILFOOT

This is a group of lines which are to appear whenever a control break occurs at the specified <identifier>. DETAILFOOT cannot occur unless the BREAK <report_ctl> has been specified. This will not occur after the apparent break when the first record is obtained.

PAGEFOOT

This is a group of lines which are to appear at the bottom of each page. The first line of this group should contain an absolute line number.

REPORTFOOT

This is a group of lines which follow the report. It is processed only once, at the end (if any details have been printed in the report). If the REPORTFOOT appears after the PAGEFOOT, then it will occupy a separate page.

Lines

Each line of a report is completely specified as to content and position.

```
------------------------------------------------------------------
  <line>              ::= , 3 LINE <[line_ctl]> <[field...]>  !
  <line_ctl>          ::=   <number> IF ( <expr> )  !
  <line_ctl>          ::= + <number> IF ( <expr> )  !
  <line_ctl>          ::=   <number>  !;
  <line_ctl>          ::= + <number>  !;
  <line_ctl>          ::=            IF ( <expr> )  !
------------------------------------------------------------------
```

The first thing after the LINE can be a line number. This can either be a relative number to give single-, double-, etc.-spacing, or it can be an absolute number. The first line of a page is 1. Line 0 means end-of-page, i.e. the current page is finished but the next one is not yet started. The default is +1.

IF

This optional specification causes <expr> to be evaluated. If it is TRUE then the line will be processed. If it is FALSE, then no processing is done and the possible side-effects of this processing do not occur.

Fields

A line is made up of zero or more fields to be printed. There are many ways of placing data in these fields.

```
-----------------------------------------------------------------
  <field>               ::= , 4   <field_value>
                                            <[field_ctl...]>   !
  <field>               ::= , 4 ( <field_value_con> )
                                            <[field_ctl...]>   !

  <field_value_con>     ::= <field_value>  !;
  <field_value_con>     ::= <field_value_con>
                                  CONCATENATE <field_value>  !

  <field_value>         ::= IF ( <expr> ) <identifier>  !
  <field_value>         ::= IF ( <expr> ) <transform>  !
  <field_value>         ::=               <identifier>  !;
  <field_value>         ::=               <transform>  !;
-----------------------------------------------------------------
```

A field can be made up of a literal, a variable, a transformation, or a concatenation of any or all of these. Any or all of these may be conditional. A conditional field_value causes <expr> to be evaluated. If the result is true, then the associated reference is included. A field is of a specific size; even if no data is placed in it, it still occupies the location.

```
-----------------------------------------------------------------
  <field_ctl>           ::= PICTURE <quoted_str>  !
  <field_ctl>           ::= EDIT <quoted_str>  !
  <field_ctl>           ::= CHARACTER ( <number> )  !

  <field_ctl>           ::= LET ( <assign...> )  !;
  <field_ctl>           ::= COLUMN <number>  !;

  <field_ctl>           ::= JUSTIFY LEFT  !;
  <field_ctl>           ::= CENTER  !;
  <field_ctl>           ::= JUSTIFY RIGHT  !;
  <field_ctl>           ::= FILL  !;
  <field_ctl>           ::= FILL ( <number> , <number> )  !;
  <field_ctl>           ::= ALIGN <quoted_str> <number>
                                  ! string is 1 character
  <field_ctl>           ::= FOLD  !
-----------------------------------------------------------------
```

The first 3 <field_ctl>'s are data type. They are mutually exclusive. They have the same meaning as in <dcl_spec>. The next 2 <field_ctl>'s are general attributes. Any or all can be applied. The last 7 <field_ctl>'s are specific to CHARACTER. They are optional, but are mutually exclusive.

A line is set to spaces before any field processing is done.

IF

This optional specification caused <expr> to be evaluated. If it is true, then the field will be processed.

LET

This optional specification causes one or more assignments to be done. This is done before any <field_values>'s are referenced; the field to be printed can be modified just before printing.

COLUMN

This optional specification indicates the print position where the field is to begin. The default is to begin 1 position to the right of the previous field (i.e. to skip a column between fields). Column numbers must be in increasing order.

LEFT, CENTER, RIGHT fields can overlap. The are filled in the order specified.

LEFT

This optional CHARACTER specification says that the data is to be placed in the field against the left end. Leading and trailing spaces are removed before determining the data length. Unused positions to the right are not modified.

CENTER

This optional CHARACTER specification says that the data is to be placed centered in the field. Leading and trailing spaces are removed before determining the data length. Unused positions to the right and left are not modified.

RIGHT

This optional CHARACTER specification says that the data is to be placed in the field against the right end. Leading and trailing spaces are removed before determining the data length. Unused positions to the left are not modified.

FILL

This optional CHARACTER specification says that the data is to be placed in the field a word at a time. If the whole string will not fit in the size specified, then additional lines will be used, containing only the FILL fields from the current line. The words will be placed beginning in position 1 of the field on each line. Words are delimited by spaces. Trailing spaces are removed before determining the data length.

FILL(n1,n2)

This form of the FILL says that the first line of the data is begun in position n1 of the field and any successive ones begin in position n2. Needless to say, both n1 and n2 must be within the field size. If a word is longer than will fit in a single line, it will be truncated to fit.

ALIGN "x" n

This optional CHARACTER specification says that the data is to be scanned from the left for the character "x". The data is then placed in the field so that this character is in position n of the field. If "x" is not found, the data is just assigned to the field.

FOLD

This optional CHARACTER specification says that if the data is longer than the field length, then the first _n_ are printed on the first line, the second _n_ on the second line, etc. Trailing spaces are removed before determining the data length.

```
-----------------------------------------------------------------
  <exec>                 ::= IF ( <expr> ) <stmt>  !;
  <exec>                 ::= <stmt>  !

  <stmt>                 ::= <assign>  !;
  <stmt>                 ::= PRINT <identifier> ;  !;
  <stmt>                 ::= <end_phase>  !
-----------------------------------------------------------------
```

The PRINT statement references a detail. Any conditional processing is specified at the detail, line, or field level. Associated <heading>'s and <footing>'s will be processed when appropriate.

The set of executable statements may be broken into phases by an <end_phase>. At the <end_phase> point two things will happen, in this order.
 1) The data specified will be saved on a temporary file, and processing will go back to retrieve more. This continues until no more data can be retrieved.
 2) Processing then continues at the <end_phase> point, it now being treated as a retrieve for anything which follows.

If a SORT is specified, then the data is re-arranged before subsequent processing.

```
-------------------------------------------------------------------
  <end_phase>            ::= HOLD <identifier , ...> ;  !;
  <end_phase>            ::= SORT <sortkey , ...> ;  !;
  <end_phase>            ::= HOLD <identifier , ...>
                             SORT <sortkey , ...> ;  !;

  <end_phase>            ::= SORT <sortkey , ...> NO DUPLICATE ;  !;
  <end_phase>            ::= HOLD <identifier , ...>
                             SORT <sortkey , ...> NO DUPLICATE ;  !

  <sortkey>              ::= <identifier>  !;
  <sortkey>              ::= <identifier> ASCENDING  !;
  <sortkey>              ::= <identifier> DESCENDING  !
-------------------------------------------------------------------
```

The HOLD specifies which items are to be saved for later use. This can be either input items or local variables. The SORT option says to rearrange them before this later use. The HOLD ... SORT ... form is used when not all fields kept are to be sorted. The hold list need only contain any fields not on the sort list. A sortkey can optionally specify ASCENDING or DESCENDING. The default is ASCENDING. The type of comparison done will depend on the data type of the field. The NO_DUPLICATE option means that only one record will be kept if more than one exist with identical sort keys. The last one is the one kept.

The ASSIGN statement places a value into a variable. The variable can be one which was retrieved from the database. This action does not change the database, it merely changes the field into which the data was retrieved.

```
-------------------------------------------------------------------
  <assign>               ::= <identifier> := <expr> ;  !;
  <assign>               ::= <identifier> := <transform> ;  !

  <transform>            ::= TRANSFORM ( <expr> , <identifier> )!;

-------------------------------------------------------------------
```

There can be different data types involved in the assignment. In this case, the appropriate conversion is done.

The TRANSFORM must have matching data types, i.e. if you are transforming an arithmetic data item, you must reference a table which has arithmetic values.

Expressions

The expressions should be essentially what is allowed in PL/I. The same operator precedence is followed.

Boolean Expressions

A BOOLEAN expression is one which can have the value of either TRUE or FALSE.

```
-----------------------------------------------------------------
  <expr>                  ::= <expr> OR <bterm>  !;
  <expr>                  ::= <bterm>  !

  <bterm>                 ::= <bterm> AND <bfact>  !;
  <bterm>                 ::= <bfact>  !

  <bfact>                 ::=     <bref>  !;
  <bfact>                 ::= NOT <bref>  !;
  <bfact>                 ::= <relation>  !;
  <bfact>                 ::= <membership>  !
-----------------------------------------------------------------
```

BOOLEAN items can be AND'ed and OR'ed. The NOT can be taken and parentheses can be used. (Note that parentheses around <expr> is described under <afact> later.)

If an <cexp> is used as a <bfact> it will be converted. If the value of the expression is 0, "0", "FALSE", "false", "F", or "f" the result is FALSE, otherwise it is TRUE.

Two general types of complex operations are available: relation, and membership.

Comparison relation

Either arithmetic or character expressions can be compared.

```
-------------------------------------------------------------------
  <relation>          ::= <cexp> <compare> <cexp>  !
  <compare>           ::= EQ  !;
  <compare>           ::= NE  !;
  <compare>           ::= LE  !;
  <compare>           ::= GE  !;
  <compare>           ::= LT  !;
  <compare>           ::= GT  !
-------------------------------------------------------------------
```

If the first expression compares correctly with the second one, the result is TRUE; otherwise, it is FALSE. The indicated compares are:

EQ	equal	NE	not equal
LT	less than	LE	less than or equal
GT	greater than	GE	greater than or equal

String Matching relation

A match can be looked for in a string sense. This means that no regard is given to surrounding characters.

```
-------------------------------------------------------------------
  <relation>          ::= <cexp> <str_rel> <cexp>  !
  <str_rel>           ::=     BEGINS  !;
  <str_rel>           ::= NOT BEGIN  !;
  <str_rel>           ::=     ENDS  !;
  <str_rel>           ::= NOT END  !;
  <str_rel>           ::=     CONTAINS  !;
  <str_rel>           ::= NOT CONTAIN  !
-------------------------------------------------------------------
```

The string can be checked for under certain conditions. Given these fields with the indicated contents:

abcd	"fundamentals of geometry"
def	"builtin functions"
ghi	"only for fun"

then this is what would happen:

abcd BEGINS "fun"	- TRUE
abcd NOT BEGIN "fun"	- FALSE
def NOT BEGIN "fun"	- TRUE
def ENDS "fun"	- FALSE
def NOT END "fun"	- TRUE
ghi ENDS "fun"	- TRUE
def CONTAINS "fun"	- FALSE
abcd CONTAINS "fun"	- TRUE

Word Matching relation

A match can be looked for in a word sense. This means that each word in the search string must occur surrounded by delimiters. Delimiters are non-alphanumeric characters, beginning of string, and end of string.

When you have

```
        string BEGINS WORD match
```

match contains one or more words. Words are separated by a single space.

```
-----------------------------------------------------------------
  <relation>          ::= <cexp> <word_rel> <cexp>  !
  <word_rel>          ::=     BEGINS WORD  !;
  <word_rel>          ::= NOT BEGIN  WORD  !;
  <word_rel>          ::=     ENDS WORD  !;
  <word_rel>          ::= NOT END  WORD  !;
  <word_rel>          ::=     CONTAINS WORD  !;
  <word_rel>          ::= NOT CONTAIN  WORD  !
-----------------------------------------------------------------
```

Given these fields with the indicated contents:

```
          abcd      "fundamentals of geometry"
          def       "builtin functions"
          ghi       "only for fun"
then this is what would happen:
  abcd BEGINS WORD "fun"       - FALSE
  abcd NOT BEGIN WORD "fun"    - TRUE
  def NOT BEGIN WORD "fun"     - TRUE
  def ENDS WORD "fun"          - FALSE
  def NOT END WORD "fun"       - TRUE
  ghi ENDS WORD "fun"          - TRUE
  def CONTAINS WORD "fun"      - TRUE
  abcd CONTAINS WORD "fun"     - FALSE
```

Membership

It can be tested whether or not an item is one of a set of things. The set of things can be gotten from a SELECT statement if it is within a WHILE clause.

```
-----------------------------------------------------------------
  <membership>        ::= <cexp>     IN ( <set> )  !;
  <membership>        ::= <cexp> NOT IN ( <set> )  !

  <set>               ::= <number , ...>  !;
  <set>               ::= <quoted_str , ...>  !
-----------------------------------------------------------------
```

It is easier to say

```
          state IN ("MN", "AZ", "MA", "LA")
```

than to say the equivalent

```
             state = "MN"
          OR state = "AZ"
          OR state = "MA"
          OR state = "LA"
```

When an item is being tested, the data type must match. Character vs. character, or arithmetic vs. arithmetic.

BOOLEAN items can be AND'ed and OR'ed. The NOT can be taken and parentheses can be used. (Note that parentheses around <expr> is described under <afact> later.)

If an <cexp> is used as a <bfact> it will be converted. If the value of the expression is 0, "0", "FALSE", "false", "F", or "f" the result is FALSE, otherwise it is TRUE.

Two general types of complex operations are available: relation, and membership.

```
-----------------------------------------------------------------
  <bref>              ::= <cexp>  !;
  <bref>              ::= TRUE  !;
  <bref>              ::= FALSE  !;
  <bref>              ::= <bbuiltin>  !

  <bbuiltin>          ::= %LEVEL ( <number> )  !;
  <bbuiltin>          ::= %LEVEL ( <identifier> )  !;
  <bbuiltin>          ::= %ABSENT ( <identifier> )  !;
  <bbuiltin>          ::= %PRESENT ( <identifier> )  !;
  <bbuiltin>          ::= %FIT  !
-----------------------------------------------------------------
```

%LEVEL(<number>)

This BOOLEAN function is TRUE if a break at level <number> has occured. <number> cannot be greater than the number of BREAK fields declared in the report in which it occurs. %LEVEL(0) is TRUE at end-of-data in each phase of processing.

%LEVEL(<identifier>)

This BOOLEAN function is true if a break has occurred on <identifier>. <identifier> must be declared as a BREAK field in the report where it occurs. If you have specified

```
   BREAK (a,b)
```

then these two are equivalent

```
   %LEVEL(2)
   %LEVEL(b)
```

%ABSENT(<identifier>)

This BOOLEAN function is TRUE if the named variable is blank.

%PRESENT(<identifier>)

This BOOLEAN function is TRUE if the named variable is non-blank.

%FIT

This BOOLEAN function is TRUE if the group in which it occurs fit on the page the first time it was tried.

Character Expressions

```
-----------------------------------------------------------------
  <cexp>                ::= <cexp> CONCATENATE <cref>  !;
  <cexp>                ::= <cref>  !
-----------------------------------------------------------------
```

The only operator defined for character strings is concatenation.

```
--------------------------------------------------------------------
  <cref>              ::= <aexp>  !;
  <cref>              ::= <quoted_str>  !;
  <cref>              ::= <cbuiltin>  !

  <cbuiltin>          ::= %SUBSTR ( <cexp> , <aexp> , <aexp> )  !;
  <cbuiltin>          ::= %SUBSTR ( <cexp> , <aexp> )  !;
  <cbuiltin>          ::= %ROMAN ( <aexp> )  !;
  <cbuiltin>          ::= %MMDDYY  !;
  <cbuiltin>          ::= %YYDDD  !;
  <cbuiltin>          ::= %MONTH  !;
  <cbuiltin>          ::= %DAY  !;
  <cbuiltin>          ::= %HHMMSS  !;
  <cbuiltin>          ::= %REPEAT ( <cexp> , <aexp> )  !
--------------------------------------------------------------------
```

%SUBSTRING

This is just like the SUBSTR in PL/I.

%ROMAN(<aexp>)

This CHARACTER function returns <aexp> converted to roman numerals. The result is VARYING.

Date/Time

These functions are all refering to date and time. This is the date and time when the report command was called.

%MMDDYY

This CHARACTER function returns the date in the form "mm/dd/yy"

%YYDDD

This CHARACTER function returns the date in the 5-character Julian form.

%MONTH

This CHARACTER function returns the month. The result is VARYING and has an initial capital letter.

%DAY

This CHARACTER function returns the name of the day of the week. The result is VARYING and has an initial capital letter.

%HHMMSS

This CHARACTER function returns the time in the form "hh:mm:ss"

%REPEAT(<cexp>,<aexp>)

This CHARACTER function returns <cexp> repeated <aexp> times.

Arithmetic Expressions

In arithmetic expressions, there can be an intermixing of FIXED and FLOAT. When this occurs, the FIXED is converted to FLOAT before the operation is done.

```
-----------------------------------------------------------------
  <aexp>             ::= <aexp> + <aterm>  !;
  <aexp>             ::= <aexp> - <aterm>  !;
  <aexp>             ::= <aterm>  !

  <aterm>            ::= <aterm> * <afact>  !;
  <aterm>            ::= <aterm> / <afact>  !;
  <aterm>            ::= <afact>  !

  <afact>            ::= <aref>  !;
  <afact>            ::= - <aref>  !;
  <afact>            ::= + <aref>  !;
  <afact>            ::= - ( <aexp> )  !;
  <afact>            ::= + ( <aexp> )  !;
  <afact>            ::=  ( <expr> )  !
-----------------------------------------------------------------
```

This describes an ordinary arithmetic expression. It is really shown here only for completeness.

```
-----------------------------------------------------------------
  <aref>             ::= <number>  !;
  <aref>             ::= <identifier>  !;
  <aref>             ::= <identifier> ( <parameter , ...> )  !;
  <aref>             ::= <abuiltin>  !

  <abuiltin>         ::= %PAGENUMBER ( <identifier> )  !;
  <abuiltin>         ::= %PAGENUMBER ( )  !
-----------------------------------------------------------------
```

If <identifer> references a character field, it is valid as long as the value in the field can be converted to FIXED or FLOAT.

%PAGENUMBER(<report_ref>)

This FIXED function returns the value of the current page of the specified report.

%PAGENUMBER()

This FIXED function returns the value of the current page of the current report.

<u>Miscellany</u>

These are a few things which have not been defined anywhere else.

Comments are allowed in the source. These are like PL/I.

```
  /* comment */
  <parameter>         ::= <expr>  !
```

Keywords

The keywords in the BNF also have short forms:

AND	&
ASCENDING	ASC
BOOLEAN	BOOL
CHARACTER	CHAR
COLUMN	COL
CONCATENATE	||
DECLARE	DCL
DEFAULT	DEF
DESCENDING	DESC
EQ	=
GE	>=
GT	>
JUSTIFY	JUST
LE	<=
LT	<
MAXLINE	MAXL
MINLINE	MINL
NE	^=
NOT	^
OR	|
PAGELENGTH	PGL
PAGEWIDTH	PGW
PARAMETER	PARM
PICTURE	PIC
SUBSTRING	SUBSTR
TRANSFORM	TRAN
VARYING	VAR

External Interfaces

Each object module, xx, will have various entry points to fulfill a range of functions.

xx$xx
This is the command interface. It will not exist if neither FILE nor ATTACH is specified in the INPUT declaration.

xx$init
This is the I/O appendage interface. The report_ I/O switch will be given the name xx as it's first parameter. When this entry is called, necessary initialization will be performed. Three entry variables will be passed back to report_; 1) write_record, 2) put_chars, and 3) close. Either 1 or 2 will be an error return depending on whether the input is to be RECORD or STREAM.

xx$eufi
This is an interface which will be called by the end-user-facility. It's interface requirements are to be determined.