

OpenCOMAL documentation

Jos Visser <josv@osp.nl>

Version: 0.3.0

Date: Wed 4 Apr 2018 23:13:33 IST

1 ALSO SEE

- `whitepaper1.txt` for information on STATIC and MODULES

2 TODO

- Document the array rvalue stuff that changed in 0.2.5-pre3

3 Intro Info

This documentation file describes OpenComal. OpenComal is an implementation of the Comal language. The OpenComal sources are covered by the GNU General Public License.

OpenComal is written in C and special care has been taken to guarantee its portability to other hardware platforms. All system dependent code has been externalized in a separate module with clearly defined entry points. A port should involve a re-design of this system dependent module and a recompilation of the OpenComal sources. A system independent stub (`pdcdsys.c`) has been provided which maps system dependent functions onto standard C functions. The interpreter will function as normal but special features such as cursor positioning, line editing and advanced memory allocation are not or only rudimentarily available.

The system dependent module of OpenComal also contains entry points to facilitate the extension of OpenComal with new PROCedures and FUNCTions whose definition can be in C. It is the intention of the author that interfaces to windowing and database systems and the like are implemented through these defined entry points.

The author can be contacted at:

Jos Visser

<josv@osp.nl>

<http://www.josvisser.nl/opencomal>

4 What is Comal

In the early history of micro computing, resources (CPU, memory, disk) were scarce, slow and expensive. Micro computers were usually distributed with limited memory capacity

and limited on board software. The famous-in-it's-own-time ZX80 (by Sinclair) probably set an all time record low with 4K ROM and 1K RAM.

BASIC was the language of choice back then. Almost all micro computers featured their own dialect of BASIC, usually present in ROM. These BASICs were basic indeed. Most of the time they lacked about everything necessary to develop well-structured and maintainable programs.

This situation led to bad-programming habits spreading like wildfire. Would-be programmers were not encouraged to produce clear program code, and in order to make the programs smaller, faster and more attractive many tricks were used like directly modifying system variables (the notorious POKE instruction), using multi statement lines, using GOTO's etc. etc.

The principles of structured programming were available of course, but structured languages like PASCAL were not really available for small micro's due to limitations in the micro's resource capacity and the then available compiler technology.

Somewhere in Denmark a teacher named Borge Christensen understood the potential dangers of the BASIC habits and he leaded a team of people who were determined to modify an available BASIC interpreter so as to create a new language which should support structured programming concepts like PROCedures, FUNCtions and GOTO-less loops. The first Comal (COMmon Algorithmic Language) interpreter was born.

5 Commercially available Comal

Comal has been available on a number of popular micro computers and/or operating systems. A short, and surely non-exhaustive, list follows:

MsDos	Comal, by UniComal
CP/M	Comal-80, by the Danish Regnecentrale
Commodore 64	Comal cartridge, by UniComal
BBC Micro	Comal ROM, by AcornSoft

6 OpenComal

In 1992 I went on an unplanned sabbatical of four months or so. With all this extra time on my hands, I decided to create a free Comal interpreter with the following features:

- Free software (although I did not know that phrase at that time).
- Containing numerous ethically justifiable extensions to the basic Comal specifications
- Highly portable
- Containing no built in limits to program and variable size (which is somewhat of a challenge on MsDos).

I created OpenComal (then called Public Domain Comal (or PDComal)) on an XT PC (8088 CPU) running MsDos using Turbo C. I did however fathom porting to Unixes and other platforms, so everything was written in pretty portable K&R C, with all operating system dependencies separated out in a set of routines in one source file (e.g. pdlinux.c).

Over the last ten years, I have been tweaking with OpenComal a bit, but not regularly. The Comal movement has all but disappeared, which is a shame because I think it is a nice language to start programming with...

And right now, in 2002, I am on yet another (this time planned) sabbatical, and I decided to rename PDComal to OpenComal, attach the GPL to it, improve Linux keyboard/screen support (using ncurses) and release it to the general public. Let's see what happens....

Apart from the interpreter (opencomal), a run-time only unit is provided (opencomalrun) which contains all the code necessary to load and execute a SAVED OpenComal program.

7 (Open)Comal concepts

7.1 Program

A Comal program consists of a number of Comal program lines. Each program line contains a line number, an optional Comal statement and an optional comment. Line numbers are used for editing purposes only! Comal commands are available to save, load, run and manipulate programs. Only syntactically correct lines can be entered into the program.

Only one program can be in memory at the same time, however, OpenComal supports EXTERNAL PROCedures and FUNCtions which are loaded from disk on demand.

Comal program listings (on screen or to file) are automatically indented and capitalized in order to create an esthetically appealing view of the structure of the program.

Before execution (RUN) of a program, it is SCANNed for errors such as incorrect program structure use (e.g. LOOP not closed with ENDLOOP, IMPORT not from within a PROC or FUNC, RETURN <exp> within a PROC). Only correct programs can be RUN.

7.2 Variables

Comal supports three types of variables:

- Integer
- Floating point
- String

A variable's type is reflected in its name. String variables are suffixed with a \$, integer variables with a #. E.g:

Integer	a#, sum#, aap#
Floating point	angle, mortgage, banana
String	name\$, title\$, fruit\$

In early Comal's, all string variables had to be explicitly DIMensioned to a certain length, like:

DIM aap\$ OF 20

In OpenComal this restriction has been relaxed. If a string variable is DIMensioned, the interpreter will maintain the string with the given maximum length in mind. Assignments to the string will be truncated if necessary. However, in OpenComal a string variable need not be dimensioned. Such an undimensioned string variable has no set maximum length.

The value ranges of each type depends on the C compiler OpenComal has been compiled with. See the system specific documentation file for more information on this.

7.3 Assisted entry

Comal allows a lot of flexibility in entering Comal statements. = may be input for :=, # for FILE and cosmetic keywords like DO, THEN, OF etc. need not be input at all.

7.4 Escape

The execution of OpenComal programs, INPUT and the LIST command can be interrupted with an escape. What the escape key is depends on the OpenComal implementation as most of the escape key processing is handled in the system dependent module (for Linux it is Ctrl-C).

Processing of the escape key while running a program can be modified with the TRAP ESC statement.

8 OpenComal keywords and tokens

8.1 Operators

Comal operators are used to combine two values to form a new one according to the operator's definition.

General syntax: <exp> <operator> <exp>

Used on numeric, returns numeric:

Used on strings and numerics, returns strings and numeric:

Used on strings, returns numerics:

Used on strings and numerics, returns numeric (False=0, True= not 0)

8.2 Assignment operators

The three Comal assignment operators assign a value to a variable (possibly a member of an array).

General syntax: <lvalue> <assignop> <exp>

OpenComal knows three assignment operators:

x-y	x minus y
x*y	x times y
x/y	x divided by y
x^y	x to the power y
x DIV y	integer division of x and y
x MOD y	remainder after integer division x and y
x EOR y	x exclusive or y
x OR y	x or y
x AND y	x and y
x AND THEN y	x and y, y is not evaluated if x is false
x OR THEN y	x or y, y is not evaluated if x is true
x BITAND y	the bitwise AND of x and y
x BITOR y	the bitwise OR of x and y
x BITXOR y	the bitwise exclusive OR of x and y, the operation is performed bit by bit on the two arguments

x+y	x plus y
-----	----------

When assigning to a string lvalue the **<exp>** string is truncated as is necessary for the DIMensioned string length. When assigning to a substring the **<exp>** string is truncated or padded with spaces as is required for the substring.

Multiple assignments can be specified on 1 program line, separated by a semicolon.

If the target of the assignment is an array then all elements of the array are assigned the same value:

8.2.1 Example

```

a:=7
b:+1; c:=9; a$:="Jos Visser"
k:-1
counter(8):+2
b$(8,6):="aap"
a$(1:4):="xxxx" // Substring assignment
b$(8)(1:2):="yy"
c$( :n#:) = a$( :2:)
```

```

DIM a(8)
a=99
```

8.3 OpenComal built-in Functions

General syntax: **<func>(<exp>)**

Takes numeric argument, return numeric:

x in y	returns position of string x in string y.
--------	---

x=y	x equals y
x<y	x smaller then y
x>y	x greater then y
x<=y	x less then or equal to y
x>=y	x greater then or equal to y
x<>y	x not equal to y

Takes string argument, returns numeric:

Takes numeric argument(s), returns string:

Takes string argument, returns string:

Takes no argument, returns numeric:

Takes no argument, returns string:

An overview of all OpenComal keywords:

8.4 APPEND

Modifier of OPEN to indicate that the sequential file is open for WRITE and data should be appended to the file.

See OPEN.

8.5 AUTO

Command facilitate the input of program lines:

General syntax: AUTO [<start>[,<increment>]]

The AUTO command generates line numbers as is specified by <start> and <increment>. The program lines can immediately be input.

8.6 BYE

The BYE command ends the OpenComal interpreter. Control is usually passed to the calling environment.

8.7 CASE

This OpenComal program structure indicates an n-way selection:

8.7.1 General syntax

```
CASE <exp> OF
  WHEN <exp1>,<exp2>,...,<expn>
    <comal line>
```

:=	simple assignment (may be entered as =)
:+	adds the <exp> to the current value of <lvalue>
:-	subtracts <exp> from the <lvalue>

ABS	absolute value
ACS	arc cosine
ASN	arc sine
ATN	arc tangent
COS	cosine
DEG	how many degrees (0-360) is the radian argument
EOF	end of file reached on specified file?
EXP	argument to the power of e
FRAC	Fractional part of argument
INT	nearest integer less than or equal to argument
LOG	natural logarithm (e log)
LOG10	10 log
NOT	not x (not true=false, not false=true)
RAD	how many radians (0-2pi) is the degree argument
ROUND	Rounds argument with respect to fractional part 0.5
RND	yields a floating point random number in the (closed) interval [0, 1]
RND(x)	yields a whole random number in the interval [0, x]
RND(x,y)	yields a whole random number in the interval [x, y]
SGN	returns -1,0,1 depending on sign of argument
SIN	sine
SQR	square root
TAN	tangent

```

    <comal line>
WHEN <relop><exp3>,<exp4>,...
    <comal line>
    <comal line>
...
OTHERWISE
    <comal line>
    <comal line>
ENDCASE

```

The expression <exp> is evaluated (once!) and compared with the different WHEN expressions. If no relational operator is provided the expressions are tested for equality. If no WHEN expressions matches the next WHEN statement is processed. If no WHEN statements match, program flow is passed to the statements following the OTHERWISE clause. If a matching WHEN exists the statements after that WHEN are executed, after which control is passed to the statement after the ENDCASE. An OTHERWISE need not be present.

VAL	numeric value of string argument
ORD	numeric systems collating sequence code of first character in string argument (Latin-9 encoding)
LEN	length of argument string

CHR\$	returns string containing of 1 character, the character whose position in the systems collating sequence is specified by the numeric argument.
GET\$	the number of characters specified by the second argument from the file specified by the first argument. The file must previously have been opened as a read type file. If the end of file is reached before the specified number of characters are retrieved, only those retrieved prior to EOF will be returned (there is no padding of spaces and no error occurs unless you attempt to read from the file again).
INKEY\$	wait and return the key pressed. If argument is -1 or not specified then wait is infinite; if 0 there is no delay; if a number ≤ 25 delay for that many seconds.
STR\$	Returns string representation of numeric argument

8.7.2 Example

```

10 FOR f:=1 TO 10 DO
20   PRINT "Kees is ";f
30   CASE f OF
40     // test kees
50   WHEN 1
60     PRINT "een"
70   WHEN <=2
80     PRINT "kleiner gelijk 2:"
90   WHEN 6, 7
100    PRINT "zes of zeven"
110   WHEN >9
120    PRINT "groter negen"
130   OTHERWISE
140    PRINT "none of the above"
150   ENDCASE
160   //
170 ENDFOR

```

8.8 CHDIR

Change the current directory.

General syntax: CHDIR <dir>

Example: CHDIR "/tmp"

8.9 CLOSE

CLOSE is used to close open files.

LOWER\$	converts the string argument to lowercase Latin-9 characters
UPPER\$	converts the string argument to uppercase Latin-9 characters

CURCOL	the current column position of the cursor on the text screen, columns are counted from left to right and the leftmost column is 1.
CURROW	the current row of the text screen that the cursor is on, rows are counted top to bottom and the top row is 1.
EOD	end of data reached (all DATA statements processed)
ERR	last error
ERRLINE	line number of line where last error occurred
ESC	TRUE if the escape key (^C) has been pressed, this is only useful if the escape key is disabled (see TRAP).
FALSE	0=1
TRUE	0=0
PI	ACS(-1)
ZONE	the current zone setting (see ZONE statement)

General Syntax: CLOSE [FILE <exp1>[,<exp2>,<exp3>,...]]

CLOSE with no parameters closes all open FILES.

8.9.1 Example

```
CLOSE
CLOSE FILE 1,5,infile#
```

8.10 CLOSED

CLOSED is a modifying keyword to specify that PROCedures and FUNCtions have their own variable pool.

See PROC.

8.11 CON

With the CON command an interrupted program (STOP, escape or error) can continue it's processing at the interrupted line. In case of a STOP interruption the program is continued at the line after the STOP.

Certain program modifications during interruption (such as modifying the current line or adding a program structure line) inhibit continuation.

General syntax: CON

8.12 CURSOR

With the CURSOR statement the video output cursor can be positioned somewhere on the output medium.

ERRTEXT\$	text of last error
KEY\$	first character in the keyboard buffer, if no key has been pressed the null string (") is returned.

General syntax: `CURSOR <exp1>,<exp2>`

The CURSOR statement is executed by the system dependent part of OpenComal. Its behaviour is not defined by OpenComal.

8.13 DATA

With the DATA statement a fixed list of constant or variable expressions can be specified for subsequent READ actions.

General Syntax: `DATA <exp>,<exp>,<exp>,...`

Every program has a current DATA pointer. At the start of the program this pointer is initialized to the first DATA line of the program. Each READ statement reads the current DATA value and advances the DATA pointer to the next DATA value, skipping lines if necessary. The EOD function becomes true if the DATA pointer does not point to any valid DATA value (e.g. no DATA lines in program or last DATA value has been READ).

With RESTORE and RESTORE <id> the DATA pointer can be moved. See RESTORE for more information.

OpenComal allows variable and constant expressions as DATA values.

8.13.1 Example

```
10 b:=7
20 WHILE NOT(EOD) DO
30   READ a
40   PRINT a
50 ENDWHILE
60 //
70 DATA 1,2,b+8,"Jos Visser"
```

8.14 DELAY

Suspend process execution for an interval measured in seconds.

General Syntax: `DELAY <seconds>`

8.15 DELETE

The DELETE statement is used to delete external files.

8.15.1 General Syntax

DELETE <stringexp>

8.16 DEL

The DEL command is used to delete program lines.

8.16.1 General Syntax

DEL <line range>

A DEL line range can be specified like this:

100	only line 100
-100	from 1st line up to and including line 100
100-	from line 100 onwards
100-200	from line 100 up to and including line 200

Specifying no line range defaults to the whole program (line 1-MAXINT). However, this is not allowed with DEL.

8.17 DIM

The DIM statement is used to define arrays and string lengths.

8.17.1 General syntax

DIM <stringvar> OF <exp>, ...

DIM <var>(<dimlist>) [OF <exp>], ...

When dimensioning a string variable or a string array to a maximum length the interpreter will take care to maintain the string to it's given maximum length.

When dimensioning an array an upper bound to it's dimension must be specified. If no lower bound is specified it defaults to 1.

8.17.2 Example

```
DIM a$ OF 4, b$ OF 6
```

```
DIM matrix(8,9), reeks(-100:100)
```

```
DIM naam$(2:6,-1:1) OF 56
```

8.18 DIR and DIR\$

The DIR statement can be used to obtain a directory listing. The DIR\$ string function yields the present working directory.

8.18.1 General syntax

```
DIR [<stringexp>]
DIR$
```

The exact output of DIR depends on the platform specific implementation. For instance under Linux, the DIR statement calls the external "ls" command so the output resembles an "ls -l" display.

8.18.2 Example

```
DIR "*.cml"
DIR
PRINT DIR$
```

8.19 DO

DO is an auxilliary keyword used with WHILE. See WHILE.

8.20 DOWNTO

DOWNTO is used to specify a FOR loop with a decreasing counter value. See FOR.

8.21 DYNAMIC

The DYNAMIC keyword is used to specify that an EXTERNAL PROC or FUNC should be reloaded at every call. See PROC.

8.22 EDIT

The EDIT command is used to edit a range of existing program lines.

General syntax: EDIT <line range>

The EDIT line range specification adheres to the previously explained DEL line range semantics. EDIT without a line range specification offers every line in the program for editing.

8.23 ELIF

ELIF is used in constructing a multi-way IF. See IF.

8.24 ELSE

The ELSE keywords specifies the start of the alternative statement section in a multi line IF. See IF.

8.25 END

The END statements halts all program execution and returns control to the interpreter's command loop. The program can not be CONTinued after an END.

When executing in command mode, the END command clears the program run environment.

8.26 ENDCASE

Specifies the end of an n-way selection CASE statement. See CASE.

8.27 ENDFOR

Specifies the end of a multi-line FOR loop. See FOR.

8.28 NEXT

Synonym for ENDFOR.

8.29 ENDFUNC

Specifies the end of a FUNCtion definition. See FUNC.

8.30 ENDIF

Specifies the END of a multi-line IF construct. See IF.

8.31 ENDLOOP

Specifies the end of a LOOP-ENDLOOP construct. See LOOP.

8.32 ENDPROC

Specifies the end of a PROCedure definition. See PROC.

8.33 ENDTRAP

Specifies the end of an error trapping TRAP/HANDLER/ENDTRAP construct. See TRAP.

8.34 ENTER

The ENTER command is used to input program lines from a file.

General syntax: ENTER "filename"

With ENTER program lines are input from the file and inserted in the program as if the program lines has been entered from the keyboard. The file can only contain program lines and no direct statements or commands.

8.35 ENDWHILE

Specifies the end of a multi-line WHILE loop. See WHILE.

8.36 ENV

8.36.1 General syntax

ENV

ENV <id>

The ENV command is used to switch between different programs that are in memory simultaneously. Each environment can contain an entire program with its own procedures, functions and variables. At each moment in time, one environment is the current environment. Commands like NEW, LOAD, SAVE et cetera act on the current environment.

The ENV command can be used to switch from environment to environment. Each environment has a name. An environment is automatically created upon first activation of it. Using ENV without a name argument provides a list of the environments in existence.

8.37 EXEC

The EXEC statement is included for compatibility with older Comal versions. It can prelude procedure call statements. See Procedure Call.

8.38 EXIT

The EXIT statement is used to exit from the nearest enclosing LOOP/ENDLOOP. See LOOP.

8.39 EXTERNAL

The EXTERNAL keyword is used to specify that a PROCedure or FUNCtion definition is not included in the program but that it should be loaded from an external file that was previously SAVED.

See PROC.

8.40 FILE

The FILE keyword is used with OPEN, READ, WRITE, PRINT, INPUT and CLOSE to specify the file number and optionally the random file record number a certain statement is to act on.

8.41 FOR

The FOR statement specifies a loop that is iterated a fixed number of times under the control of a FOR (or index) variable.

8.41.1 General syntax

```
FOR <lval>:=<exp> TO|DOWNT0 <exp> [STEP <exp>] DO
...
...
ENDFOR
```

```
FOR <lval>:=<exp> TO|DOWNT0 <exp> [STEP <exp>] DO <stat>
```

The FOR loop variable is assigned and gets incremented or decremented at every iteration. If a STEP clause is not present it defaults to 1. DOWNT0 signifies a negative step. The FOR loop behaves like a WHILE loop, i.e.: the test whether the loop should be left or iterated is evaluated beforehand. For example, in the following cases the loop statements are not executed:

```
FOR f:=10 TO 0 DO
FOR f:=0 TO 10 STEP -1 DO
FOR f:=10 DOWNT0 0 STEP -1 DO
```

Short and long forms exist of the FOR loop. In the short form the DO is immediately followed by a simple statement (e.g., no program structure statement): `FOR f:=-1 to 1 DO PRINT f`

No ENDFOR need be present in the short form.

8.41.2 Examples

```
FOR a(9):=10 DOWNT0 0 STEP 2 PRINT a(9)
```

```
FOR angle:=0 TO 2*PI STEP 0.1 DO
...
ENDFOR
```

```
FOR i:=i^2 TO i^3 STEP i DO PRINT i
```

8.42 FUNC

Specifies the start of a FUNCTION definition.

See PROC.

8.43 IMPORT

The IMPORT statement imports variables into a CLOSED environment.

General syntax: `IMPORT [<id>:],<id>[()],...`

In a CLOSED PROCedure or FUNCTION no variables from any of the calling environments are visible. Furthermore all new variables are local to the current environment and do not influence any calling environment. With the IMPORT statement the definitions

of variables from a calling environment can be imported into the current environment. Modification of an IMPORTED variable also affects the variable in its own environment.

OpenComal allows the IMPORT statement to specify from which calling environment the variable should be imported. The calling stack is reversly traversed and the variables are imported from the first environment with the specified name (PROC/FUNC name, `_program` for the main program).

If no identifier is specified variables are imported from the current environments parent environment. The parent environment is the nearest enclosing CLOSED environment. For a non-nested PROC/FUNC this means the enviroment of the main program. For a nested PROC/FUNC the parent environment is the environment of the most nested enclosing CLOSED PROC or FUNC.

`()` must be used to import an array.

8.43.1 Examples

```
IMPORT aap:a, b(), c$
IMPORT _program: tabel()
IMPORT any_var$, bny_var$
```

8.44 GLOBAL

Synonym for IMPORT.

8.45 HANDLER

Specifies the start of the error handling statements of a TRAP/HANDLER/ENDTRAP structure. See TRAP.

8.46 IF

The IF statement is used to selectively execute OpenComal statements according to the truth value of an expression.

8.46.1 General syntax

```
IF <exp> THEN <simplestat>

IF <exp> THEN
    ...
ELIF <exp>
    ...
ELSE
    ...
ENDIF
```


The short form IF evaluates the expression and then either executes the simple statement or not. Control is then passed to the statement following the IF.

The long form IF evaluates the expression and then either executes the statements between the IF and the corresponding ELIF, ELSE or ENDIF. If the IF expression does not evaluate to TRUE OpenComal hops to the next ELIF and evaluates that expression. If this evaluates to TRUE the statements between the ELIF and the corresponding ELIF, ELSE or ENDIF are executed. Multiple ELIFs can be present. If all ELIF's have been dealt with and no ELIF expression has evaluated to TRUE the ELSE statements are executed.

No ELIF or ELSE need be present in the long form IF. If an ELSE clause is present it may not be followed by an ELIF.

8.46.2 Example

```
IF a=b THEN PRINT "eq1"
```

```
IF a=b THEN
  PRINT "eq1"
ENDIF
```

```
IF a=b THEN
  PRINT "eq1"
ELSE
  PRINT "neq"
ENDIF
```

```
IF a=b THEN
  PRINT "eq1"
ELIF a=c
  PRINT "a eq1 c"
ELSE
  PRINT "neq"
ENDIF
```

8.47 INPUT

With the INPUT statement data is read from a file or from the keyboard and assigned to program variables.

8.47.1 General syntax

```
INPUT FILE <exp>[,<exp>]: <lvallist>
INPUT [AT <row>,<col>[,<len>]:] [<string>:]<lvallist>[<mark>]
```

INPUT FILE reads data from the specified file. If the file has been opened for RANDOM a record number must be specified. The other form of INPUT reads data from

the keyboard or the current input file (see also: SELECT INPUT). Optionally a string is printed before the data is input.

If a variable is an array, INPUT reads one value and assigns that value to every member of the array!!! (as opposed to READ which reads and assigns n values)

INPUT FILE can process data from files written by OpenComal programs with PRINT FILE.

INPUT reads data from the keyboard. Multiple values must be separated by a comma. If a comma must be input as a part of a string variable, enclose the keyboard input with ".".

A 0 for the row or column means not to change it (stay in the same row or column). If the <mark> is a comma, the cursor remains where it is after the reply. If it is a semicolon, spaces are printed to the next zone (one space by default if ZONE is not specified), then the cursor remains at that position.

8.47.2 Example

```
INPUT "Enter your choice :":c
INPUT FILE 5,recno#:a,b,c
INPUT name,number
INPUT AT 5,2,10: "Post Code: ": post'code$
```

8.48 LIST

The LIST command is used to list program lines.

8.48.1 General syntax

```
LIST <linerange>[,<filename>]
LIST <id>[,<filename>]
```

The LIST line range can be specified as with the DEL command. If no line range is mentioned the whole program is listed. If the LIST command specifies an identifier the PROC or FUNC in the program with that name is listed. By specifying a filename LIST's output is redirected to a file. The file thus created can be ENTERed.

Depending on the system, LIST can provide paged output, where listing is temporarily halted after a screenful of data. On Linux, <space> and <return> continue the listing (for another page or just one line), whereas Ctrl-C and 'q' terminate the listing.

8.48.2 Example

```
LIST
LIST -200
LIST "aap"
LIST 100-200,"aap"
```

```
LIST print_proc,"file"
```

8.49 LOCAL

The LOCAL statement introduces variables and arrays that are local to the current environment even though that environment is not CLOSED.

General syntax: LOCAL <var>,<var\$> OF <exp>, <var>(<dimlist>)

Any variables that a non-CLOSED PROC or FUNC introduces are added to the parent environment of the PROC or FUNC (for a description of the parent environment, see IMPORT). With LOCAL you can make new variables that are not added to the parent environment but to the current environment. Any variables with the same name in any of the calling environments are not affected.

LOCAL can contain DIM-like definitions of strings and arrays.

8.49.1 Example

```
LOCAL keuze#, a$ OF 20, matrix(20,20)
LOCAL name$(10) OF 40
```

8.50 LOAD

The LOAD command is used to replace the program in memory with a previously SAVED program.

8.50.1 General syntax

```
LOAD "filename"
LOAD
```

If no filename is given the filename last LOADED or SAVED is used.

8.51 LOOP

The LOOP keyword signifies the beginning of a LOOP/ENDLOOP iteration of OpenComal statements.

8.51.1 Syntax

```
LOOP
  ..
  ..
ENDLOOP
```

The statements between LOOP and ENDLOOP are repeatedly executed. A LOOP/ENDLOOP loop can be left with an EXIT statement:

```
EXIT
EXIT WHEN <exp>
IF <exp> THEN EXIT
```

EXIT transfers control to the statement after the ENDLOOP of the nearest enclosing LOOP. EXIT itself may be nested inside other IF's, REPEAT's, WHILE's etc.

8.52 MKDIR

The MKDIR statement creates the directory named as an operand.

General syntax: MKDIR <dir>

Example: MKDIR "/tmp/comal"

8.53 NAME

The NAME keyword is used to specify pass-by-name parameter passing. See PROC.

8.54 NEW

The NEW command removes the current program from memory.

8.55 NULL

Does nothing.

8.56 OF

OF is used as a cosmetic keyword with CASE.

8.57 OPEN

With the OPEN statement external files can be opened for subsequent processing.

General syntax: OPEN FILE <exp>,<stringexp>,<type>

with type =	READ
	WRITE
	RANDOM <exp> [READ ONLY]
	APPEND

The external file is designated by it's name (the string expression of OPEN). The specified file type determines which access is possible to the file:

READ	only READ FILE and INPUT file allowed
WRITE	READ FILE, INPUT FILE, PRINT FILE and WRITE FILE allowed
APPEND	like WRITE but opens an existing file and appends data at the file's tail
RANDOM	See below.

The user specified file number is the file's identification within the program. Ideally all OPEN files should be CLOSED as well, however, OpenComal closes open files when necessary (e.g., at BYE, NEW, RUN).

Random files are files which are logically segmented into records with a fixed and pre-determined record length. This record length must be specified at OPEN. OpenComal keeps no administration of this record length. It is the user's responsibility to specify the correct length at every OPEN.

Random files are opened for READ and WRITE by default, however by specifying READ ONLY the file is opened for read only.

Actions on random files require a record number to be specified:

FILE <exp>,<exp>

The first <exp> specifies the file number by which a file has been opened. The second <exp> specifies the random record number.

Random record sizes should be carefully calculated beforehand. Unfortunately, the size of a data item can vary from one implementation to another so no basic guidelines can be given. OpenComal writes a data item to a file as follows:

Integer	1 byte type, k bytes data
Float	1 byte type, n bytes data
String	1 byte type, k bytes length, p bytes data

with	p=LEN(string\$)
	k=sizeof an integer
	n=sizeof a floating point value

Example for an Msdos/Turbo C implementation of OpenComal:

Integer	1 byte type, 4 bytes integer data
Float	1 byte type, 8 bytes floating point data
String	1 byte type, 4 bytes length of string, p bytes data

In my personal opinion this aspect of Comal has always been very weak since it requires the programmer to be much too familiar with the underlying machine implementation. I am working on a different (structured) approach towards working with files in OpenComal.

8.57.1 Example

OPEN FILE 1,"aap",READ

```
OPEN FILE 2,filename$,RANDOM 40 READ ONLY
```

8.58 OTHERWISE

The OTHERWISE clause is used to specify the start of the OpenComal program lines that should be executed if no WHEN-clause of a CASE statement can be activated. See CASE.

8.59 PAGE

The PAGE statement usually clears the screen. However, because PAGE is entirely executed by the system dependent module no exact definition can be given.

8.60 PASS

With the PASS statement the programmer can send commands to the host operating system.

General syntax: PASS <stringexp>

OpenComal does not process the command in any way but passes it on to the operating system.

8.60.1 Example

```
PASS "ls >dir.dir"
PASS "rm "+file$
PASS "Z EOD"
```

8.61 PRINT

With the PRINT statement data can be printed on the output screen or to an external file.

8.61.1 General syntax

```
PRINT FILE <exp>[,<exp>]:<exp list>
PRINT FILE <exp>[,<exp>]: USING <stringexp>:<num exp list>
PRINT [AT <row>,<col>:] <exp list>
PRINT [AT <row>,<col>:] USING <stringexp>:<num exp list>
```

Basic PRINT evaluates the expressions in the expression list and prints their values on the screen or onto the current SELECT OUTPUT file. Values in the list may be separated by , and ;. There is no difference between , and ;. Data is printed as is. If the last element of the expression list is followed by ; or , then the cursor will not skip to a new line, otherwise it will.

PRINT USING is used to display numeric expressions in a predetermined format. The format string looks like `##.####` hereby signifying how many digits should be placed before and after the decimal point.

PRINT FILE writes data to previously opened files. The file should be opened for WRITE, APPEND or RANDOM. The main difference between PRINT FILE and WRITE FILE is that WRITE FILE uses binary format and PRINT FILE uses text format. Also, WRITE FILE can write variables (incl. arrays) and PRINT FILE writes expressions.

8.61.2 Example

```
PRINT "Output value = ";oval;" Kbytes"
PRINT USING "##.####": angle;
PRINT FILE 4,recno: name$,number,78
PRINT AT 9,1: USING "$###.##": amount
```

8.62 PROC (and FUNC)

Comal fully supports named procedures and functions that can take parameters.

8.62.1 General syntax

```
PROC <id>[(<parlist>)] [CLOSED] [[DYNAMIC] EXTERNAL <stringexp>]]
..
..
ENDPROC

FUNC <id>[(<parlist>)] [CLOSED] [[DYNAMIC] EXTERNAL <stringexp>]]
..
..
ENDFUNC
```

Procedures and functions can be defined anywhere in the program. The call to the PROC or FUNC need not succede the routines definition like in PASCAL.

Procedure calls are simple OpenComal statements. Specifying their name and all parameters if necessary leads to a shift in flow control to the first statement of the PROCEDURE. Functions can be called from any expression.

A procedure name must be an ordinary identifier, a function name can be any type identifier. The type of the function name must equal the return type of the function, e.q.:

first\$	returns a string
aap	returns a floating point value
fiet\$#	returns an integer value

The procedure returns to the caller upon execution of a RETURN statement. If the ENDPROC of a procedure is reached, an implicit RETURN is executed.

Functions must explicitly return a value to the caller with the RETURN statement. The execution of an ENDFUNC statement yields an error.

Procedures and functions can be fully nested with OpenComal. A nested PROC or FUNC can call all PROCs and FUNCs defined at the same level in the parent PROC/FUNC, plus all PROCs and FUNCs defined at a higher level.

Each procedure and function creates a local variable environment upon call. All passed parameters become variables in this local environment and do not interfere with any variables with the same name in any other environment.

Optionally, a routine can be declared CLOSED. This means that all variables used in the PROC/FUNC are a part of the local environment. New variables will belong to this local environment and will not interfere with existing variables with the same name in any other environment. Variables from the parent environment are not visible within a CLOSED PROC or FUNC. If they are to be used they must be imported with the IMPORT statement. With the LOCAL statement new variables can be introduced in the local environment in a non-CLOSED PROC/FUNC.

OpenComal differs significantly from other Comal's in the concept of OPEN/CLOSED procedures and functions. Other Comal's usually don't allow CLOSED procedures to call OPEN ones. OpenComal does.

The variables that are introduced and used in a non-CLOSED PROC/FUNC (apart from the PROC/FUNC's parameters and explicitly LOCALLY defined variables) belong to the parent environment (also called the OPEN environment) of the procedure. This parent environment is the environment of the most deeply nested CLOSED parent procedure or function of the PROC/FUNC meant here. The main program is treated as a CLOSED procedure at level -1.

Now onto the subject of parameters. Comal allows to type of parameter passing: by value and by reference. OpenComal supports these two types and adds call-by-name and the passing of PROC and FUNC type parameters. The parameter list definition contains a list of identifiers with optionally a modifier signifying the type of parameter pass. () must be used to specify the passing of an entire array:

aap	call by value of into local var aap
bert#()	call by value into local integer array bert#
REF name\$	call by reference into name\$
REF table()	call by reference into array table
NAME bus	call by name into locale name value bus
PROC a	passing of a procedure name into a
FUNC b\$	passing of a string function into b\$

The PROC/FUNC passing mechanism allows the programmer to specify the name of a OpenComal defined procedure or function to a procedure or function. The call of the parameter in the PROC/FUNC will lead to a call to the passed PROC/FUNC. A short example:

```
10 aap(a)
20 aap(b)
30 //
```



```

40 PROC aap(PROC c)
50   c
60 ENDPROC
70 //
80 PROC a
90   PRINT "Hello from a"
100 ENDPROC
110 //
120 PROC b
130   PRINT "Hello from b"
140 ENDPROC

```

When calling a procedure or a function a passed PROC/FUNC takes priority over a PROC/FUNC explicitly defined with the same name.

The other exotic feature of OpenComal is call-by-name. This allows the programmer to pass an entire expression to a PROC/FUNC. This expression is not evaluated upon call but is evaluated every time the name value is used in the PROC/FUNC! Example:

```

10 aap(voornaam$+" visser")
20 //
30 PROC aap(NAME name$)
40   PRINT name$
50   PRINT name$
60   PRINT name$(1:3)
70 ENDPROC
80 //
90 FUNC voornaam$ CLOSED
100   IF EOD THEN RETURN "End of DATA"
110   READ a$
120   RETURN a$
130 ENDFUNC
140 //
150 DATA "jos", "patrick"

```

The expression `voornaam$+" visser"` is evaluated three times within PROC `aap`, yielding a different result each time!! The NAME expression is evaluated in the environment of the caller!!

Usually PROCedures and FUNCtions are incorporated into the program (inline so as to speak). However, OpenComal also features the possibility to store the program code of PROCedures and FUNCtions on an external file.

If this is the case, the program must contain a stub for the PROC/FUNC containing it's name, the keyword `EXTERNAL` and an string expression which must evaluate to the name of the file containing a SAVED program text:

```

1066 PROC aap EXTERNAL "aap.sq"

```

Upon calling `'aap'` the file `aap.sq` is LOAded and SCANned. This file may contain only one routine (with nested PROC/FUNC's). Its program text is then executed. The PROC/FUNC header stub may contain a parameter list definition, however this is not

compulsary. If it contains one it must match the PROC/FUNC parameter definition in the file. If the stub contains:

```
1066 PROC aap(a, REF b) EXTERNAL "aap.sq"
```

and the PROC in the file aap.sq looks like this:

```
10 PROC aap(a#, REF b)
```

an error is generated because the first parameters do not match. The names of the parameters are not matched and may be different.

The name of the external file containing the routine's code need not be a constant. The following is perfectly legal:

```
2000 FUNC banaan$ EXTERNAL prog$+".sq"
```

The string expression is evaluated IN THE CALLING VARIABLE ENVIRONMENT and should result in a filename.

If an external PROC/FUNC file is pointed to by a constant string expression (like "aap.sq") it is called a STATIC external. If a variable string expression is used to specify the PROC/FUNC code filename it is called a DYNAMIC external.

The difference is that static external procedures and functions are maintained in memory after the first call. Subsequent calls need not load the file again, resulting in enormous speed gains. However, changes in the procedure/function program file are not processed until a rerun of the program. Dynamic external routines are discarded from memory upon PROC/FUNC return. Another call will lead to a re-evaluation of the string expression and will the resulting program file. Multiple calls of the same external procedure will result in considerable overhead.

OpenComal's educated guess about the nature of an external routine can be modified by the STATIC and DYNAMIC keywords:

```
1200 PROC aap DYNAMIC EXTERNAL "aap.sq"
1300 //
2020 FUNC fruit$ STATIC EXTERNAL prog$
```

A reason for this might be that an external routine is very large and you do not want to tie up memory with it after a call, or the program filename of an external routine is determined once and not changed after that, so you want to load it only once.

External procedures and functions may not contain DATA lines. All DATA is read from the main program. External routines may call other external routines. If a dynamic external routine calls a static external routine, the program code of the static routine is removed from memory upon removal of the calling dynamic external.

8.63 RANDOM

The RANDOM keyword is used to specify that the file to be OPENed is to be treated as a random record file. File actions (WRITE, READ etc.) must specify which record number in the file is to be processed. See OPEN.

8.64 RANDOMIZE

Randomizes the random number generator.

General syntax: RANDOMIZE [<seed>]

This generates a series of pseudo random numbers. You only need to use the RANDOMIZE command once in your program (such as right the the very beginning).

Specify a "seed" number after RANDOMIZE and you cause a specific series of random numbers to be generated, The series of numbers will be the same each time that specific seed is used. This is helpful while testing a program that uses random numbers.

8.64.1 Example

```
RANDOMIZE
RANDOMIZE num
```

8.65 READ

lines or from external files.

General syntax: READ [FILE <exp>[,<exp>]:] <lvalist>

The READ/DATA variant evaluates the expression pointed to by the current data pointer, assigns it's value to the variable in the lval list, advances the data pointer and processes the next element in the list.

READ FILE reads data elements from the specified file and optionally the specified record. If the variable in the variable list is an array then READ FILE reads as many elements from the file as there are elements in the array.

8.65.1 Example

```
READ name$,wpl$,salary,age#
READ FILE 4,6: noels#,table
```

The READ ONLY clause is used to specify that the random record file to be opened is opened for READ only (sounds logical). See OPEN.

8.66 REF

The REF keyword is used to specify pass-by-reference parameter passing. See PROC.

8.67 RENUM

The RENUM command is used to renumber program lines.

General syntax: RENUM [<start>][,<inc>]

If no start or increment is used, they default to 10.

8.67.1 Example

```
RENUM 100,20
RENUM ,100
RENUM
```

8.68 REPEAT

REPEAT signifies the beginning of a REPEAT/UNTIL loop: an iteration of OpenComal statements with a test at the end of the iteration.

8.68.1 General syntax

```
REPEAT
    ...
    ...
UNTIL <exp>
```

The statements between the REPEAT and the UNTIL are executed as long as the <exp> is FALSE. If the <exp> evaluates to TRUE at the execution of the UNTIL, control is transferred to the statement after the UNTIL.

8.68.2 Example

```
REPEAT
    INPUT "Enter number neq 0: ":a
UNTIL a<>0
```

8.69 REPORT

REPORT causes an error (optionally you can specify what error number to generate).

General syntax: REPORT [<error code>]

Part of the error handler structure. This is useful when using multiple nested handlers. REPORT puts you into the next outer handler. If REPORT is issued while not in a handler section, the error is reported to the system. REPORT is very system dependent.

8.69.1 Example

```
REPORT
REPORT 256
```

8.70 RESTORE

The RESTORE statement moves the current DATA pointer to a different location.

8.70.1 General syntax

```
RESTORE
RESTORE <id>
```

Plain RESTORE moves the DATA pointer to the first DATA line in the program. RESTORE <id> locates the line marked <id>: in the program and moves the DATA pointer to the first DATA line after the mentioned line.

8.70.2 Example

```
RESTORE
RESTORE aap
```

8.71 RETRY

The RETRY statement retries a TRAP part. See TRAP.

8.72 RETURN

With the RETURN statement the processing of a called PROCedure or FUNCtion ends and control is returned to the caller. See PROC.

8.73 RMDIR

The RMDIR statement removes the directory entry specified by the *directory* argument.

General syntax: RMDIR <dir>

Example: RMDIR "/tmp/comal"

8.74 RUN

load and execute a program from an external file (RUN statement).

8.74.1 General syntax

```
RUN
RUN <stringexp>
```

8.74.2 Example

```
RUN
RUN prog$+".sq"
```

8.75 SAVE

SAVE is used to store the current OpenComal program in memory to an external file.

8.75.1 General syntax

```
SAVE "filename"
SAVE
```

If no filename is specified the filename last SAVED or LOADED is used. The format of the SAVED file is highly implementation dependent and can't be LOADED by a different OpenComal implementation.

8.76 SELECT OUTPUT

The SELECT OUTPUT statement specifies that all output from the running program (PRINT etc.) should be written to a file instead of to the screen.

General syntax: SELECT OUTPUT <stringexp>

A file with the specified name is opened and all program output is redirected to it. If the file exists the program output is appended at the end of the file. Output is directed to the screen again by specifying an empty string as filename.

A standard destination location is "ds:", Data Screen, which returns to the default.

8.76.1 Example

```
SELECT OUTPUT ""
SELECT OUTPUT "filenout"
SELECT OUTPUT "ds:"
```

8.77 SELECT INPUT

The SELECT INPUT statement specifies that all input into the running program (INPUT etc.) should be read from a file instead of from the keyboard.

General syntax: SELECT INPUT <stringexp>

The file with the specified name is opened and all subsequent INPUT statements read data from the file. Input is directed from the keyboard again by specifying an empty string as filename.

8.77.1 Example

```
SELECT INPUT "data.in"
SELECT INPUT ""
```

8.78 SCAN

The SCAN command invokes the OpenComal pre-RUN program checks without running the program if it is correct. Any program (structure) errors are reported.

8.79 STATIC

The STATIC keyword is used in the definition of EXTERNAL routines. It signifies that the called procedure or function should be kept in memory upon return. See PROC.

8.80 SYS

The SYS statement, SYS() and SYS\$() functions have been made available as methods to gain access to non-standard extensions of OpenComal in a specific implementation.

8.80.1 General syntax

statement	SYS <exp list>
functions	SYS(<exp list>)

When the SYS statement or one of the SYS() functions is executed control is passed to predetermined entry points in the system dependent module. The expressions in the list are *not* evaluated by the OpenComal interpreter. If evaluation is necessary this should be invoked by the extensions program code. This gives the extensions the possibility to use unevaluated expressions as sub-keywords.

A number of SYS items have been incorporated in the default extensions part of OpenComal:

SYS <flag>,on off	Set internal flag
SYS(<flag>)	Return status of internal flag
SYS\$(<flag>)	Return status of internal flag
SYS(version)	Returns OpenComal version no.
SYS\$(host)	Returns name of OpenComal implementation
SYS\$(interpreter)	Return interpreter name (opencomalrun, opencomal)
SYS\$(version)	Return interpreter version
SYS SYSOUT,"file"	Send all screen output to file "file"
SYS SYSIN,"file"	Read all input (incl. commands) from file "file"

Internal flags are:

debug	Switches internal debugging state
yydebug	Switches YACC parser debugging
show_exec	Whether EXEC keyword shown in listings
prog_trace	Same as TRACE statement
short_circuit	Short circuit boolean exp eval?

The Linux version of OpenComal supports `SYS(sbrk)` and `SYS(now)`. `SYS(sbrk)` which gives the current end address of the data segment as a Comal integer. This is really handy when monitoring the OpenComal interpreter's memory usage from within a Comal program. `SYS(now)` gives the current time of the system in seconds since the epoch (midnight on 1st january 1970).

8.81 STEP

The STEP clause is used to specify the value with which a FOR loop variable should be incremented (TO) or decremented (DOWNTO). See FOR.

8.82 SPC\$

General syntax: `SPC$(<numexp>)`

The SPC\$ function returns a string containing the specified number of spaces.

Example: `a$:=SPC$(16)`

8.83 STOP

8.83.1 General syntax

`STOP`

`STOP <stringexp>`

The STOP statement halts program execution and gives control to the OpenComal interpreter command loop. The program can be continued with the CON command. STOP is not trapped by a TRAP/ENDTRAP construct. If the optional string expression is present, it is evaluated and printed when the program stops. Come to think of it, you can get the OpenComal interpreter to loop in infinite recursion when using "STOP f" in a FUNCTION 'f'. You don't want to do this...

8.84 TAB

Prints spaces up to the column specified.

General syntax: `TAB(<column number>)`

If that position is already exceeded, it goes to the specified position on the next line. TAB is always part of a PRINT statement.

Example: `PRINT TAB(col),name$`

8.85 THEN

Specifies the statement(s) that should be executed if the expression of an IF statement evaluates to TRUE. See IF.

8.86 TO

Cosmetic keyword in FOR statements

8.87 TRACE

The TRACE statement can be used to switch OpenComal program tracing on and off. When tracing is ON each line is listed before it is executed.

General syntax: TRACE on|off

8.88 TRAP

The TRAP keyword plays a role in two kinds of statements: The TRAP ESC statement to switch escape processing on and off, and the TRAP/HANDLER/ENDTRAP statement to catch all kinds of program errors. The TRAP statement signifies the beginning of an error trapping program structure:

8.88.1 General syntax

```
TRAP ESC[-|+]
```

```
TRAP
```

```
...
```

```
...
```

```
HANDLER
```

```
...
```

```
RETRY
```

```
...
```

```
ENDTRAP
```

As soon as TRAP ESC- is executed all escape conditions will be ignored by the program. After a TRAP ESC+ depressing the escape key will result in a program STOP as usual.

In a long form TRAP, the statements between TRAP and HANDLER are executed. If an error occurs during the execution, control is passed to the statements after HANDLER. If no error occurs control is passed to the statements after ENDTRAP as soon as HANDLER is executed. A HANDLER section need not be present.

If the HANDLER part contains a RETRY statement, the execution of the HANDLER part is terminated and control is passed back to the statement following the TRAP.

8.88.2 Example

```
TRAP ESC-
```

```
TRAP ESC+
```

```
TRAP
```

```
DEL "ofile"
```

```

ENDTRAP

TRAP
    PRINT 1/0
HANDLER
    PRINT "Error: ";Err$
ENDTRAP

```

8.89 UNIT and UNIT\$

The UNIT statement sets the current disk drive of the Comal interpreter. The UNIT\$ returns the currently selected UNIT.

8.89.1 General syntax

```

UNIT <stringexp>
UNIT$

```

The exact implementation depends on the operating system. For instance under Linux, the UNIT statement does nothing whereas the UNIT\$ function always returns "C:".

8.89.2 Example

```

UNIT "1:"
PRINT UNIT$

```

8.90 UNTIL

The UNTIL statement marks the end of a REPEAT/UNTIL loop. See REPEAT.

8.91 USING

The USING clause of PRINT allows the formatted printing of numeric values. See PRINT.

8.92 WHEN

The WHEN statement is used to specify the selection criteria in an n-way selection CASE. See CASE.

8.93 WHILE

WHILE signifies the beginning of a WHILE/ENDWHILE loop: an iteration of Open-Comal statements with a test at the beginning of the iteration.

8.93.1 General syntax

```
WHILE <exp> DO
    ...
    ...
ENDWHILE
```

```
WHILE <exp> DO <simplestat>
```

The statements between the WHILE and the ENDWHILE (long form) or the statement after the DO (short form) are executed as long as the <exp> is TRUE. If the <exp> evaluates to FALSE at the execution of the WHILE, control is transferred to the statement after the ENDWHILE.

8.93.2 Example

```
WHILE a<>0 DO INPUT "Enter number neq 0: ":a
```

```
WHILE keuze<>0 DO
    PAGE
    prt_menu
    keuze:=menu_keuze
ENDWHILE
```

8.94 WRITE

WRITE writes variables to previously opened external files.

General syntax: WRITE FILE <exp>[,<exp>]: <varlist>

WRITE FILE operates just like PRINT FILE with this difference that WRITE FILE can be used to write complete arrays to a file.

8.94.1 Example

```
WRITE FILE t: a, b$, c#
WRITE FILE 1,1: root#
```

8.95 ZONE

Sets the zone to the number specified.

General syntax: ZONE <tab interval>

The zone determines the interval in tab positions on an output line. The default zone is 1 (a zone at each column). The semicolon is the zone separator (for PRINT statements and at the end of INPUT statements). By default, a semicolon outputs one space (if a ZONE statement is previously used, it outputs spaces to the next zone). When a comma

is used as a separator in a PRINT statement (or at the end of an INPUT statement) no spaces are printed, and the cursor remains where it is (null separator).

Example: ZONE 5

8.96 <id>:

The <id>: statement has no execution behaviour but simply marks a program line for reference by RESTORE. In this way a self documenting, non-line number dependent program line reference is possible.

8.96.1 Example

```
10 RESTORE aap
1200 aap:
1202 DATA 1,4,9,16,25,36
```

8.97 Procedure call

The procedure call statement is used to call OpenComal defined PROCedures.

Procedures are called by their name. Older Comal implementations demanded the keyword EXEC to signify a procedure call. OpenComal allows the use of EXEC when entering program lines but will not show it in program listings.

The procedures parameters should be enclosed in parenthesis. A procedure with no parameters may not specify an empty parameter list like ().

8.97.1 Example

```
menu("Main menu",1)
build_screen
lees(a,b$,"xxx")
```

The following part is an adaptation of the OpenComal YACC input with which valid OpenComal input lines are parsed.

All terminals (ending in SYM) are returned from the lexical analyzer of functions are recognized into 5 different terminal tokens:

rnSYM	Returns Num	like EOD, ERR
rsSYM	Returns String	like ERRTEXT\$
tnrnSYM	Takes Num, Returns Num	like ABS, COS
tnrsSYM	Takes Num,Returns String	like STR\$

The grammar as here presented is not the most beautiful one imaginable for OpenComal. However, beauty has been discarded in favour of ease of processing and the limits an LALR(1) parser generator like YACC imposes.

9 The Grammar

```

a_comal_line    :   comal_line eolnSYM
                  |   error eolnSYM
                  ;

comal_line      :   command
                  |   intnumSYM program_line optrem
                  |   simple_stat
                  |   optrem
                  ;

optrem          :   remSYM
                  |   /* epsilon */
                  ;

command         :   quitSYM
                  |   list_cmd
                  |   saveSYM optfilename
                  |   loadSYM optfilename
                  |   enterSYM stringSYM
                  |   envSYM optid2
                  |   runSYM
                  |   newSYM
                  |   scanSYM
                  |   autoSYM autolines
                  |   contSYM
                  |   delSYM line_range
                  |   editSYM line_range
                  |   renumberSYM renumlines
                  ;

list_cmd        :   listSYM line_range
                  |   listSYM stringSYM
                  |   listSYM line_range commaSYM stringSYM
                  |   listSYM id
                  |   listSYM id commaSYM stringSYM
                  ;

line_range      :   /* epsilon */
                  |   intnumSYM
                  |   minusSYM intnumSYM
                  |   intnumSYM minusSYM
                  |   intnumSYM minusSYM intnumSYM
                  ;

renumlines      :   intnumSYM
                  |   intnumSYM commaSYM intnumSYM

```

```

        | commaSYM intnumSYM
        | /* epsilon */
        ;

autolines : intnumSYM
        | intnumSYM commaSYM intnumSYM
        | commaSYM intnumSYM
        | /* epsilon */
        ;

program_line : complex_stat
        | simple_stat
        | /* epsilon */
        ;

complex_stat : case_stat
        | data_stat
        | elif_stat
        | exit_stat
        | for_stat
        | func_stat
        | if_stat
        | proc_stat
        | until_stat
        | when_stat
        | while_stat
        | label_stat
        | complex_1word
        ;

simple_stat : close_stat
        | cursor_stat
        | del_stat
        | dim_stat
        | dir_stat
        | local_stat
        | exec_stat
        | import_stat
        | input_stat
        | open_stat
        | os_stat
        | print_stat
        | read_stat
        | restore_stat
        | return_stat
        | run_stat
        | select_out_stat
        | select_in_stat
        | stop_stat

```

```

        | sys_stat
        | trace_stat
        | trap_stat
        | unit_stat
        | write_stat
        | xid
        | assign_stat
        | simple_1word
    ;

complex_1word    :    elseSYM
    | endcaseSYM
    | endfuncSYM optid
    | endifSYM
    | loopSYM
    | endloopSYM
    | endprocSYM optid2
    | endwhileSYM
    | endforSYM optnumlvalue
    | otherwiseSYM
    | repeatSYM
    | trapSYM
    | handlerSYM
    | endtrapSYM
    ;

simple_1word      :    nullSYM
    | endSYM
    | exitSYM
    | pageSYM
    ;

case_stat        :    caseSYM exp optof
    ;

close_stat       :    closeSYM
    | closeSYM optfileS exp_list
    ;

cursor_stat      :    cursorSYM numexp commaSYM numexp
    ;

data_stat        :    dataSYM exp_list
    ;

del_stat         :    delSYM stringexp
    ;

dir_stat         :    dirSYM opt_stringexp

```

```

;

unit_stat  :   unitSYM stringexp
;

local_stat :   localSYM local_list
;
local_list :   local_list commaSYM local_item
|             local_item
;

local_item :   numid opt_dim_ensions
|             stringidSYM
|             stringidSYM ofSYM numexp
|             stringidSYM dim_ensions of numexp
|             stringidSYM dim_ensions
;

dim_stat   :   dimSYM dim_list
;

dim_list   :   dim_list commaSYM dim_item
|             dim_item
;

dim_item   :   numid dim_ensions
|             stringidSYM ofSYM numexp
|             stringidSYM dim_ensions of numexp
|             stringidSYM dim_ensions
;

of          :   ofSYM
|             timesSYM
;

opt_dim_ensions :   dim_ensions
|                   /* epsilon */
;

dim_ensions :   lparenSYM dim_ension_list rparenSYM
;

dim_ension_list :   dim_ension_list commaSYM dim_ension
|                   dim_ension
;

dim_ension :   numexp
|             numexp colonSYM numexp
|             numexp becminusSYM numexp

```



```

;

elif_stat  :   elifSYM numexp optthen
;

exit_stat  :   exitSYM ifwhen numexp
;

ifwhen     :   ifSYM
              |   whenSYM
;

exec_stat  :   execSYM xid
;

for_stat    :   forSYM numlvalue assign1 numexp todownto numexp optstep optdo
optsimple_stat
;

todownto    :   toSYM
                |   downtoSYM
;

optstep     :   stepSYM numexp
                |   /* epsilon */
;

func_stat   :   funcSYM id procfunc_head optclosed opt_external
;

if_stat     :   ifSYM numexp optthen optsimple_stat
;

import_stat :   importSYM id colonSYM import_list
                |   importSYM import_list
;

import_list :   import_list commaSYM oneparm
                |   oneparm
;

input_stat  :   inputSYM input_modifier lval_list
;

input_modifier :   file_designator
                  |   stringSYM colonSYM
                  |   /* epsilon */
;

open_stat   :   openSYM fileSYM numexp commaSYM stringexp commaSYM open_type

```

```

;

open_type    :    readSYM
              |    writeSYM
              |    appendSYM
              |    randomSYM numexp optread_only
              ;

os_stat      :    osSYM stringexp
              ;

print_stat   :    printi
              |    printi print_list optpr_sep
              |    printi usingSYM stringexp colonSYM prnum_list optpr_sep
              |    printi file_designator print_list
              ;

printi       :    printSYM
              |    semicolonSYM
              ;

prnum_list   :    prnum_list pr_sep numexp
              |    numexp
              ;

print_list   :    print_list pr_sep exp
              |    exp
              ;

pr_sep       :    commaSYM
              |    semicolonSYM
              ;

optpr_sep    :    pr_sep
              |    /* epsilon */
              ;

proc_stat    :    procSYM idSYM procfunc_head optclosed opt_external
              ;

read_stat    :    readSYM optfile lval_list
              ;

restore_stat :    restoreSYM optid2
              ;

return_stat  :    returnSYM optexp
              ;

run_stat     :    runSYM stringexp

```

```

;

select_out_stat :  select_outputSYM stringexp
;

select_in_stat  :  select_inputSYM stringexp
;

stop_stat      :  stopSYM optexp
;

sys_stat       :  sysSYM exp_list
;

until_stat     :  untilSYM numexp
;

trace_stat     :  traceSYM numexp
;

trap_stat      :  trapSYM escSYM plusorminus
;

plusorminus    :  plusSYM
                  |  minusSYM
;

when_stat      :  whenSYM when_list
;

when_list      :  when_numlist
                  |  when_strlist
;

when_numlist   :  when_numlist commaSYM when_numitem
                  |  when_numitem
;

when_numitem   :  relop numexp
                  |  numexp
;

when_strlist   :  when_strlist commaSYM when_stritem
                  |  when_stritem
;

when_stritem   :  relop stringexp
                  |  stringexp
                  |  inSYM stringexp

```

```

;

relop      :   gtrSYM
            |   lssSYM
            |   eqlSYM
            |   neqSYM
            |   geqSYM
            |   leqSYM
            ;

while_stat :   whileSYM numexp optdo optsimple_stat
            ;

write_stat :   writeSYM file_designator lval_list
            ;

assign_stat :   assign_list
            ;

assign_list :   assign_list semicolonSYM assign_item
            |   assign_item
            ;

assign_item :   numlvalue nassign numexp
            |   strlvalue sassign stringexp
            ;

nassign     :   assign1
            |   assign2
            ;

sassign     :   assign1
            |   becplusSYM
            ;

assign1     :   eqlSYM
            |   becomesSYM
            ;

assign2     :   becplusSYM
            |   becminusSYM
            ;

label_stat :   idSYM colonSYM
            ;

xid         :   idSYM
            |   idSYM lparenSYM exp_list rparenSYM
            ;

```

```

exp      :   numexp
          |   stringexp
          ;

numexp    :   numexp2
          ;

numexp2   :   numexp2 eqlSYM numexp2
          |   numexp2 neqSYM numexp2
          |   numexp2 lssSYM numexp2
          |   numexp2 gtrSYM numexp2
          |   numexp2 leqSYM numexp2
          |   numexp2 geqSYM numexp2
          |   numexp2 andSYM numexp2
          |   numexp2 andthenSYM numexp2
          |   numexp2 orSYM numexp2
          |   numexp2 orthenSYM numexp2
          |   numexp2 eorSYM numexp2
          |   numexp2 plusSYM numexp2
          |   numexp2 minusSYM numexp2
          |   numexp2 timesSYM numexp2
          |   numexp2 divideSYM numexp2
          |   numexp2 powerSYM numexp2
          |   numexp2 divSYM numexp2
          |   numexp2 modSYM numexp2
          |   stringexp2 eqlSYM stringexp2
          |   stringexp2 neqSYM stringexp2
          |   stringexp2 lssSYM stringexp2
          |   stringexp2 gtrSYM stringexp2
          |   stringexp2 leqSYM stringexp2
          |   stringexp2 geqSYM stringexp2
          |   stringexp2 inSYM stringexp2
          |   minusSYM numexp2 %prec USIGN
          |   plusSYM numexp2 %prec USIGN
          |   intnumSYM
          |   floatnumSYM
          |   numlvalue2
          |   tsrnSYM lparenSYM stringexp2 rparenSYM
          |   tnrnSYM lparenSYM numexp2 rparenSYM
          |   rnSYM
          |   sysSYM lparenSYM exp_list rparenSYM
          |   lparenSYM numexp2 rparenSYM
          ;

stringexp :   stringexp2
          ;

stringexp2 :   stringexp2 plusSYM string_factor
          |   string_factor

```

```

;

opt_stringexp  :  stringexp
                |  /* epsilon */
;

string_factor  :  strlvalue2
                |  string_factor substr_spec
                |  tnrsSYM lparenSYM numexp2 rparenSYM
                |  rsSYM
                |  stringSYM
                |  syssSYM lparenSYM exp_list rparenSYM
                |  lparenSYM stringexp2 rparenSYM
;

substr_spec   :  lparenSYM substr_spec2 rparenSYM
;

substr_spec2  :  numexp colonSYM numexp
                |  colonSYM numexp
                |  numexp colonSYM
;

optnumlvalue  :  numlvalue
                |  /* epsilon */
;

optexp        :  exp
                |  /* epsilon */
;

optid         :  id
                |  /* epsilon */
;

optid2        :  idSYM
                |  /* epsilon */
;

optfile       :  file_designator
                |  /* epsilon */
;

optfileS      :  fileSYM
                |  /* epsilon */
;

lval_list     :  lval_list commaSYM lvalue
                |  lvalue

```

```

;

lvalue      :   numlvalue
             |   strlvalue
;

numlvalue   :   numlvalue2
;

numlvalue2  :   xid
             |   intidSYM
             |   intidSYM lparenSYM exp_list rparenSYM
;

strlvalue   :   strlvalue2
;

strlvalue2  :   stringidSYM
             |   stringidSYM lparenSYM exp_list rparenSYM
             |   stringidSYM substr_spec
             |   stringidSYM lparenSYM exp_list rparenSYM substr_spec
;

file_designator :   fileSYM numexp colonSYM
                  |   fileSYM numexp commaSYM numexp colonSYM
;

opt_external   :   externalSYM stringexp
                 |   dynamicSYM externalSYM stringexp
                 |   staticSYM externalSYM stringexp
                 |   /* epsilon */
;

procfunc_head  :   lparenSYM parmlist rparenSYM
                 |   /* epsilon */
;

parmlist      :   parmlist commaSYM parmlitem
                 |   parmlitem
;

parmlitem     :   oneparm
                 |   refSYM oneparm
                 |   nameSYM id
                 |   procSYM idSYM
                 |   funcSYM id
;

oneparm       :   id

```

```

        | id lparenSYM rparenSYM
        ;

id      : numid
        | stringidSYM
        ;

numid   : idSYM
        | intidSYM
        ;

exp_list : exp_list commaSYM exp
        | exp
        ;

optsimple_stat : simple_stat
              | /* epsilon */
              ;

optfilename : stringSYM
            | /* epsilon */
            ;

optof      : ofSYM
           | /* epsilon */
           ;

optdo      : doSYM
           | /* epsilon */
           ;

optthen    : thenSYM
           | /* epsilon */
           ;

optread_only : read_onlySYM
            | /* epsilon */
            ;

optclosed  : closedSYM
           | /* epsilon */
           ;

```