

Transbase[®]
Control Command Language CCL

Transaction Software GmbH
Willy-Brandt-Allee 2
D-81829 München
Germany
Phone: +49-89-62709-0
Fax: +49-89-62709-11
Email: info@transaction.de
<http://www.transaction.de>

Version 6.8.1.40
November 02, 2010

Contents

1	Introduction	3
1.1	Conventions for Notations	4
1.2	Statements and Blocks	4
1.3	Variables, Assignment and Input	4
1.4	Strings, Delimiter Strings, Concatenation	5
1.5	Strings and Variables	6
1.5.1	Expressions	8
1.5.2	Boolean Expressions	8
1.5.3	Comparison Expressions	9
1.5.4	String Expressions	10
1.5.5	Numerical Expressions	11
1.5.6	Precedence of Operators	12
1.6	IF Statement and WHILE Statement	12
1.7	Procedures (block defined) and Parameters	13
1.7.1	Definition of a Block Defined Procedure	13
1.7.2	Executing a Procedure	14
1.7.3	Semantics of Parameter Substitution	15
1.7.4	Definition-Time and Execution-Time of Procedures	16
1.8	Command Files (file-defined procedures)	17
1.9	Local and Global Procedures	18
1.9.1	Visibility of Procedures	19
1.9.2	Local and Global Variables	20
1.9.3	Creation of Variables	20

1.9.4	Resolution of Variable References	21
1.9.5	Restrictions and Properties of Global Procedures	21
1.9.6	Examples	22
1.10	Include and Define	24
1.11	SQL Statements	25
1.11.1	Error Handling of SQL Statements	25
1.11.2	Connect, Disconnect and Login	26
1.11.3	Transaction Handling (CT, AT)	27
1.11.4	Multi Database Handling	27
1.11.5	SELECT Statements, Sample Program	28
1.11.6	SQL Statements executed by RUN	30
1.11.7	Update and Delete positioned	30
1.11.8	Value Representation in CCL and TransbaseSQL	31
1.12	TYPE and DISPLAY: Output of Result Tables	32
1.13	I/O Redirection and Pipes	33
1.14	Exit to Shell	34
1.15	EDIT a File	34
1.16	Change Working Directory	34
1.17	Builtin Procedures	35
1.17.1	System Environment	35
1.17.2	Variable Handling	35
1.17.3	String Handling	36
1.17.4	Database and Transaction Handling	39
1.17.5	Cursor Handling	40
2	CCL Start and Arguments	43
3	CCL Grammar	44
3.1	Conventions for Syntax Notation	44
3.1.1	Productions	44
3.1.2	Program	44
3.1.3	I/O Statements	45

3.1.4	Handling of Variables	45
3.1.5	Procedure Handling	45
3.1.6	Control Structures	46
3.1.7	Redirection Statements	46
3.1.8	System Statements	46
3.1.9	Include and Define Statement	47
3.1.10	Comparison Operators	47
3.1.11	Numerical Operators	47
3.1.12	Concatenation Operators	48
3.1.13	Expression Syntax	48
3.1.14	Comparison Expression	48
3.1.15	Syntax of Numerical Expressions	48
3.1.16	Procedure Call within Expression	49
3.1.17	Syntax of Strings	49
3.1.18	Constants and Variables	49
3.1.19	Database Statements	49
3.1.20	Connections, Login and Transactions	50
3.1.21	Evaluation of Queries	50
3.1.22	Output of Query Results	50
4	Current Limitations of CCL	52

Chapter 1

Introduction

CCL is a command interpreter which offers an interface to the Transbase SQL kernel. The syntax of CCL commands has been designed as an easy to use procedural programming language which offers many capabilities of higher programming languages.

Some of these features are:

- if then else
- while
- local and global variables
- a wide range of expressions
- local and global user defined procedures
- parameters and return values for procedures
- recursive use of procedures
- predefined builtin functions

This introductory chapter is intended to give an overview about the main language principles by some examples and program fragments. Whereas the precise language definition is given in the successive chapters, this introduction serves to give enough intuition for an easier understanding of formal language properties.

1.1 Conventions for Notations

CCL treats keywords case insensitive, therefore in all examples they are written in UPPERCASE. CCL produces certain output on stderr and stdout like the CCL prompt or results of some statements. In all subsequent examples output from CCL is printed in italic.

1.2 Statements and Blocks

A CCL program is a series of statements. CCL accepts a CCL program stored in one or several files. Alternatively, CCL can be called in an interactive mode where the statements are typed one by one and CCL prompts with *ccl* for input. With the exception of assigning values to variables each statement begins with a CCL keyword. There is no special separator such as ';' between statements. Statements are implicitly separated by the keywords their beginning.

A block is a sequence of statements which may be empty. In some CCL constructs blocks are used to execute more than one statement. For example a CCL program is such a block.

For example, ECHO has exactly one argument and echoes it (with a newline):

```
ccl> ECHO hello
hello
ccl> ECHO hello ECHO hello
hello
hello
```

Comments are indicated by `//` and are finished by newline.

```
ccl> // this is a comment
ccl>
ccl> ECHO Hello World // error, because of more than one argument
.. CCL reports an error ..
```

1.3 Variables, Assignment and Input

CCL supports variables with a very weak type concept. Variable names are composed of letters, digits, underscore and dots (e.g. valid names are `var`, `var.0`, `var_5`). Variables are interpreted as of type `STRING` except in the context of some numerical operators as described later. Variables are implicitly created by assigning values to them using the `:=`. Their value is referenced by prefixing the variable

name with the \$-sign. Variables are deleted automatically upon program end but they can be deleted explicitly by the UNSET command:

```
ccl> var := hello
ccl> ECHO $var
hello
ccl> UNSET var
ccl> echo $var
.. CCL reports an error ..
```

The \$-operator is called the dereferencing operator and the := operator is called the assignment operator. Additionally, CCL distinguishes between local and global variables by the keywords LOCAL and GLOBAL. Some more details are explained in chapter 1.9 (Local and Global Procedures).

Note that unlike other similar languages, CCL reports an error if a variable which has not been assigned a value is dereferenced.

Note also that target variables can be computed (formally, the target variable can be an expression):

```
ccl> var := t
ccl> $var := hello2           // here, hello2 is assigned to t
ccl> ECHO $t
hello2
```

The ACCEPT command reads a value (string) from terminal (stdin) and assigns it to a variable. It comes in 2 variants:

```
ccl> ACCEPT var                // here CCL waits for input
ccl> ACCEPT var PROMPT {input:} // here CCL prompts input: then waits
```

1.4 Strings, Delimiter Strings, Concatenation

Some of the rules for constructing strings and dereferencing variables may appear non-intuitive at first glance, but are easily derivable by the fact that CCL is designed to interface with SQL language (i.e. it embeds SQL). As an example, CCL knows the command TYPE which has an SELECT Statement as argument. TYPE writes all the result tuples of the evaluated SELECT Statement to stdout. To delimit the SQL string syntactically, the SQL string is surrounded by delimiters which are not part of the SQL language, namely braces {}.

```
ccl> TYPE { SELECT * FROM SYSTABLE }
.. CCL prints result tuples ..
```

A character sequence enclosed in braces is called a **delimiter string**. The delimiters are the means to construct strings which contain special characters that are part of the SQL Syntax like blank, +, -, *, quotes etc. Note that the braces are not part of the string but only syntactically delimit the string. Strings without delimiters should be considered as a shorthand notation which can be used whenever the string neither contains special characters nor white space characters.

```
ccl> ECHO hello
hello
ccl> ECHO {hello}
hello
ccl> ECHO {*** hello ***}
*** hello ***
```

The last example shows that blanks are not eliminated in delimiter strings; thus the SQL statement above only works because SQL itself allows white space characters inside queries.

Of course, we can also store queries in variables now:

```
ccl> query := { SELECT * FROM SYSTABLE }
ccl> TYPE $query
.. CCL prints result tuples ..
```

1.5 Strings and Variables

Of course, it should be possible to construct parameterized SQL queries. For example one could write:

```
ccl> TYPE { SELECT * FROM $tname }
```

where the variable tname contains the name of a table. This is possible in CCL because no SQL token can begin with a \$-sign. Thus, one important rule to remember is:

In delimiter strings, the variable dereferencing mechanism is applied.

Note, that also inside SQL string literals the CCL variable dereferencing mechanism is applied.

```
ccl> v := abc
ccl> TYPE { SELECT * FROM t WHERE ti = '$v' }
```

Clearly, this SQL statement searches for the string abc on field ti of table t. If the statement should search for a \$-sign the \$-sign has to be escaped by a preceding \-sign (backslash). Analogous arguments hold for the so called "SQL delimiter identifiers" supported in Transbase, e.g. a Transbase table may have \$t as its name which works if the dereferencing operator is escaped; so the following statement works as supposed:

```
ccl> TYPE { SELECT * FROM '$t' }
```

Thus a further rule in CCL is:

In delimiter strings, the effect of special characters can be suppressed by preceding them with the escape character \.

Thus, for CCL the characters \$, \, have special meaning in delimiter strings: \$ triggers a variable substitution, \ prevents CCL from interpreting the subsequent character as special character. \\$ will be interpreted as \$, \\ will be interpreted as \ and \q will be interpreted as q. (q is no special character but also can be escaped) e.g. to produce the string '\$\$\$', we must write '\\$\\$\\$' in CCL Likewise, to assign the string \$t to a variable we write:

```
ccl> v := {\$t}
ccl> ECHO $v
$t
```

This last example also shows that variable dereferencing is not recursive, i.e. if the value of a variable appears to be a valid dereferencing operator on another variable, the dereferencing operation is not recursively applied. However, dereferencing can be nested statically:

```
ccl> t := 567
ccl> v := t
ccl> ECHO $($v)
567
```

Additionally the following CCL rule applies:

Inside delimiter strings, metacharacters other than \ and \$ (e.g. +, - etc) loose their meta meaning.

The reason is that otherwise SQL strings would loose their semantics as in the following example (imagine a table t with fields ta and tb:)

```
ccl> TYPE { SELECT * FROM t WHERE ta + tb = 7 }
```

Outside the delimiter string, CCL would try to evaluate '+' as numerical operator. This, however, is undesired inside the delimiter string.

1.5.1 Expressions

Like other programming languages, CCL offers several classes of operators with different precedences, which can be intermixed in one expression. Expressions can be surrounded by parentheses to overrule the predefined precedences.

Basic elements of expressions are strings, delimiter strings, dereferenced variables and constants.

Example:

```
ccl> x := 5 // assign a value to variable x
ccl> define Y 10 // define constant Y
ccl> echo abc // echo a string
abc
ccl> echo {*** xyz ***} // echo a delimiter string
*** xyz ***
ccl> echo $x // echo dereferenced variable
5
ccl> echo #Y // echo dereferenced constant
10
```

Also procedures with parameters can be invoked and their return values are used for further computations.

Example:

```
ccl> string := {TransAction Software}
ccl> search := {on Soft}
ccl> ECHO substr($string,6) // call procedure substr
Action Software
ccl> ECHO strlen(Software) + 5
13
ccl> ECHO substr($string,strstr($string,$search))
on Software
```

CCL offers boolean, comparison, string and numerical operators which will be explained in the following chapters.

1.5.2 Boolean Expressions

CCL treats the integer value 0 in conjunction with boolean operators as **FALSE** and all remaining integer values as **TRUE**. Non integer numerical values will be

truncated to integer if they are supplied as operands of boolean operators. The result of a boolean operator is either 1 (for TRUE) or 0 (for FALSE).

The keywords OR, XOR, AND and NOT are treated as boolean operators with the usual boolean semantics.

Example:

```
ccl> ECHO 1 OR 0
1
ccl> ECHO 0 OR 5           // 5 is also treated as TRUE
1
ccl> ECHO 1 XOR 1
0
ccl> ECHO 5 AND 3
1
ccl> ECHO NOT 1
0
ccl> ECHO NOT 0.2        // 0.2 will be truncated to 0
1
```

The AND operator has higher precedence than the OR and XOR operator, the NOT operator has higher precedence as the AND operator. All boolean operators have lower precedence than all other operators (see precedence table)

Example:

```
ccl> ECHO 0 OR 1 AND NOT 1 // precedence not, and, or
0
```

1.5.3 Comparison Expressions

CCL supports comparison operators whose result is typically interpreted in IF statements and WHILE statements. The standard operators =, <>, <, >, <=, >= compare their string valued arguments, i.e. they make a lexicographic comparison. Their numerical counterparts are written EQ, NE, LT, GT, LE, GE. The latter may run into an error if one of their arguments cannot be interpreted as a numeric value (this is analogous to +, -, *, /, %). All comparison operators have same precedence which is higher than the precedence of boolean operators but is lower than the precedence of all other operators (see precedence table). The result of a comparison expression is a boolean value 1 (for TRUE) or 0 (for FALSE).

```

ccl> a := 5
ccl> b := 05
ccl> IF a=a ...           // yields TRUE (string compare)
ccl> IF a EQ a ...       // yields error (no numerical values)
ccl> IF $a = $b ...      // yields FALSE (string compare)
ccl> IF $a EQ $b ...     // yields TRUE (numerical compare)
ccl> IF 2 < 10 ...      // yields FALSE (string compare)

```

Note: To compare numerical values one of the operators EQ, NE, LE, GE, LT, GT must be used. Unforeseen results may occur if the string counterparts are used as explained in the following example.

Example:

```

ccl> i := 1
ccl> WHILE $i <= 10      // string compare !!!
>   DO   ECHO $i
>       i := $i + 1
>   OD
1
ccl>

```

The body of the loop is executed only once, because if the condition `$i <= 10` is checked for the second time when `i` has a value of 2 the condition will yield `FALSE`. (2 is greater then 10 if the string compare is used).

1.5.4 String Expressions

For String handling CCL offers explicit concatenation operators, namely `&` and `&&`. They work outside the delimiter string and have the same precedence. `&` concatenates its operators, `&&` additionally puts a blank into the result between the operators. Both have lower precedence than all numerical operators but have higher precedence than all other operators.

```

ccl> ECHO 3+4&var
7var
ccl> ECHO 3 + 4 & var
7var
ccl> ECHO 3+4 && var
7 var
ccl> ECHO 3+4 & { } & var
7 var

```

The last example shows that `&&` has little importance because it can be obtained by subsequent application of `\&`, but is a convenient way to form expressions like `$prename && $lastname`.

Inside a delimiter string `{ .. }`, after variable substitution, the resulting characters including blanks and other special characters are concatenated into one result string, i.e. there is an implicit character concatenation.

1.5.5 Numerical Expressions

The special characters `+, -, *, /, %` are interpreted as operators with the usual semantics in numerical computations.

```
ccl> var := 5
ccl> ECHO $var+1
6
ccl> ECHO $var + 1
6
ccl> ECHO $var + 1.5
6.50000e+000
ccl> ECHO $var + 1.5e2
1.55000e+002
ccl> ECHO $var + 1.5e-2
5.01500e+000
ccl> ECHO $var % 3           // 5 modulo 3
2
ccl>
```

If one of the operands cannot be interpreted as a number, CCL reports an error:

```
ccl> ECHO var + 1
.. CCL reports an error ..
```

CCL performs numerical computations according to the types of the operands: if both operands are digits without fractional point and without exponential number then CCL computes with 4 byte integer precision else CCL computes with C-type double precision.

The usual precedence of numeric operators apply. Additional parentheses can be used (see precedence table).

Note: To compare numerical values one of the operators `EQ`, `NE`, `LE`, `GE`, `LT`, `GT` must be used. Unforeseen results may occur if the string counterparts are used as explained in the following example.

Example:

```

ccl> i := 1
ccl> WHILE $i <= 10      // string compare !!!
>   DO   ECHO $i
>       i := $i + 1
>   OD
1
ccl>

```

The body of the loop is executed only once, because if the condition `$i <= 10` is checked for the second time when `i` has a value of 2 the condition will yield **FALSE**. (2 is greater than 10 if the string compare is used).

1.5.6 Precedence of Operators

The following table defines the precedence of operators. A precedence 1 means highest precedence.

precedence	operators
1	\$ # ()
2	+ - (unary)
3	* / %
4	+ -
5	& &&
6	= <> < <= > >= EQ NE LT LE GT GE
7	NOT
8	AND
9	OR XOR
10	:=

1.6 IF Statement and WHILE Statement

The IF statement is written

```
IF <expr> THEN <block> ELSE <block> FI.
```

The WHILE statement occurs in the variants

```
WHILE <expr> DO <block> OD
```

and

```
DO <block> OD WHILE <expr>
```

In the bodies of the loop, the construct `BREAK` causes an exit of the loop, the construct `CONTINUE` causes a jump to the end of the body.

1.7 Procedures (block defined) and Parameters

A CCL program can be structured by use of procedures. The first discussed type of procedures is called block-defined procedure.

1.7.1 Definition of a Block Defined Procedure

A block-defined procedure contains a header and a body consisting of a CCL block. A procedure may contain formal parameters which are substituted by actual parameters when a procedure is called. The following example defines a block-defined procedure `min` with two formal parameters `x` and `y`.

```
ccl> PROCEDURE min(x,y)      // defining procedure min
>BEGIN                      // with two formal parameters
>  IF    $x LE $y
>  THEN  RETURN $x
>  ELSE  RETURN $y
>  FI
>END
```

In the above defined procedure `min` the first two formal parameters can be accessed via the expressions `$x` and `$y`. The names of the formal parameters are declared in the head of the procedure. Formal parameters are treated like variables which are local to the procedure.

An additional way to access the formal parameters is by the expressions `$1`, `$2` etc. In the above example, `$1` is equivalent to `$x` and `$2` is equivalent to `$y`. The expression `$0` always retrieves the procedure name.

The declaration of formal parameters can be omitted. In this case, the supplied actual parameters can only be accessed by `$1`, `$2` etc.

Note: A procedure may be called with less actual parameters than declared formal parameters. To make better use of this feature it can be checked whether an actual parameter is set.

Example:

```

PROCEDURE max(x,y)
BEGIN
  IF    NOT ISSET(x) OR    // check if formal
        NOT ISSET(y)      // parameters are set
  THEN ECHO {too few parameters in procedure $0}
        RETURN 0
  FI

  IF    $1 GE $2
  THEN  RETURN $1          // $1 is same as $x
  ELSE  RETURN $2          // $2 is same as $y
  FI
END

```

Another way to check how many parameters are set is to use the special term `$#` which always retrieves the number of actually set parameters. Note that in a called procedure the expression `$0` retrieves the procedure name and so `$#` yields 1 if no further actual parameter is supplied.

Example:

```

PROCEDURE sum
BEGIN
  i := 1
  sum := 0

  WHILE      $i LT $#
  DO        sum := $sum + $($i)
           i := $i + 1
  OD

  RETURN $sum
END

```

1.7.2 Executing a Procedure

CCL offers two possibilities to call a procedure. One is to explicitly call a procedure by an `EXEC` statement. When the procedure has finished, the special variable `status` contains the value of the last executed `RETURN` statement.

Example: (requires above definitions of max and min)

```
ccl> EXEC min(5,6)
ccl> ECHO $status
5
ccl>
```

The second way to invoke a procedure is a function call as part of an expression. In this case calculations with the return value (value of variable status) of a function can directly be performed.

Example: (requires above definitions of max and min)

```
ccl> ECHO min(5,6)
5
ccl> ECHO max(5,min(10,20))      // second parameter for max
10                               // will be the return value
                                // of min(10,20)

ccl> i := max(10,20)
ccl> echo $i
20
```

1.7.3 Semantics of Parameter Substitution

When CCL executes a procedure call, then first the list of expressions for the actual parameters is evaluated. Then the body of the procedure is entered.

In the body of procedures the parameters can be accessed via the expressions \$1, \$2 ... or via the declared names in the head of the procedure definition. If the value of a parameter p is accessed via \$p, then CCL substitutes \$p with the value of the previously evaluated expression of the corresponding actual parameter.

Example:

```
ccl> PROCEDURE mult(x,y)
> BEGIN
>   i := 0    // only for demonstration
>
>   RETURN $x * $y
> END
ccl> i := 5
ccl> ECHO mult($i+1,$i-1)
```

```

24
ccl> ECHO $i
0
ccl>

```

Procedure `mult` makes an arithmetic computation and therefore must be called with arithmetic values. It would be an error to call it with the name of a variable.

Note: Although the value of the variable `i` is reset to zero before the parameters `x` and `y` are accessed they have the values 6 and 4 because the list of expressions `($i+1,$i-1)` has already been evaluated at calling time.

Example:

```

ccl>    PROCEDURE incr(x)          // increases value of a variable
>      BEGIN
>        $x := $($x) + 1
>      END
ccl> i := 5
ccl> EXEC incr(i)
ccl> ECHO $i
6
ccl>

```

Procedure `incr` has a transient parameter, therefore it must be called with the name of a variable. To access the value of the variable it must do a double dereference in the body. Simply remember that if `incr` is called as `incr(i)` then inside the body, `$x` is the same as `i`, thus `$($i)` is the same as `$i`.

This all follows from the fact that variables must be explicitly dereferenced by `$` to access their values.

1.7.4 Definition-Time and Execution-Time of Procedures

CCL distinguishes between procedure definition time and procedure execution time. The definition time of a procedure starts if CCL recognizes the keyword `PROCEDURE` and ends at the keyword `END`. During definition time no statement will be executed. The execution time of a procedure starts if an `EXEC` statement is executed or an expression with a function call is evaluated. The execution time of a procedure ends if the end of the procedure is reached or the procedure is explicitly finished via a `RETURN` or `EXIT` statement. **RULE: A procedure can be executed only if it is defined.**

As illustration, consider the following example.

Example:

```

PROCEDURE    main          // Definition of main
BEGIN
  .
  EXEC      abc           // call abc
  .
END

PROCEDURE    abc          // Definition of abc
BEGIN
  ECHO abc
END

EXEC      main          // execution of main

```

During the definition of procedure main the statement "EXEC abc" seems to violate the above rule, because the procedure abc is not yet defined. But the rule only restricts procedure execution and so the EXEC statement in procedure main does not violate the rule, because no statement is executed at procedure definition time. Only an internal structure for the body of the procedure will be generated. At the moment when the procedure main will be executed (execution time) this structure will be used to execute the statements of procedure main.

So, at execution time of main, procedure abc is defined and the example is correct.

1.8 Command Files (file-defined procedures)

CCL allows to write command files and to execute them like block-defined procedures. This feature allows to split large CCL programs into several smaller CCL scripts in separate files. According to the terminology of block-defined procedures, CCL command files are also called file-defined procedures.

The behaviour of file-defined procedures is like block-defined procedures and differs only in the definition mechanism.

Remember that a block-defined procedure is defined (and can be called) if the CCL interpreter has parsed the procedure. In contrast to that, a file-defined procedure f is defined (and can be called with name f), if the CCL interpreter can open the file with name f.

Note: if the current working directory has changed inside a CCL program via a CD command it may be that a file-defined procedure is no longer accessible under the old filename.

The body of a file-defined procedure is all what is written in the corresponding file, syntactically it is not a block but a sequence of statements. Also the header `PROCEDURE` or parameter declarations are missing.

Nevertheless, parameters can be passed at the call and can be accessed inside the command file via the positional notation `$1`, `$2` etc.

Example: Assume a file `sum`:

```
Begin of file sum
  i := 1
  sum := 0

  WHILE      $i LT $#
  DO    sum := $sum + $($i)
        i := $i + 1
  OD

  RETURN $sum

End of file sum
ccl> ECHO sum(1,2,3)
6
ccl> EXEC sum(1,2,3)
ccl> ECHO $status
6
```

Note: It is more likely that procedures like `sum` will be written as block-defined procedures than as file-defined procedures. The latter typically will be larger and perhaps will not contain direct return values. But the above example shows all what is possible with file-defined procedures and that both procedure types principally have the same power.

If CCL works interactively, then conceptually it can be assumed that CCL is processing a special command file, namely `"stdin"`.

1.9 Local and Global Procedures

CCL allows to distinguish block-defined procedures into `LOCAL` and `GLOBAL`. This is useful to define a set of local procedures which may not be called from other files and to avoid name collisions.

1.9.1 Visibility of Procedures

A file-defined procedure is visible if the file containing the CCL commands for this procedure is accessible (readable). If the current directory is changed then the procedure name changes in the same way as the path to this file changes. In general this file-defined procedure is always visible, therefore file-defined procedures are always GLOBAL.

After the CCL interpreter has parsed a block-defined procedure `p`, then `p` is at least visible (callable) in the file which contains the procedure. By default, the visibility of a block-defined procedure is restricted to the file which contains its definition. This can also be made explicit by prefixing the definition with the keyword LOCAL.

Alternatively, a block-defined procedure can be explicitly declared with the keyword GLOBAL. After the CCL interpreter has parsed a GLOBAL block-defined procedure `p`, then `p` can be called from anywhere until CCL finishes. If a procedure `p` is called within a file `f` and the CCL interpreter has seen a local procedure `p` in `f`, then a possibly existing global (block or file-defined) procedure with the same name `p` is temporarily hidden by the local block-defined procedure `p`. This is a usual hiding mechanism known from other languages, too.

GLOBAL procedures have some limitations as far as access to variables and calls of other procedures is concerned. This is described in a later chapter.

There is a special class of procedures called builtin procedures. These procedures are global and already defined when CCL starts. Additionally they cannot be hidden by user specified (file or block defined) procedures because their names are reserved. Note also that the name of a user defined procedure must not be equal to a keyword.

Example:

File `flocal` contains following CCL code:

```
LOCAL PROCEDURE myname
BEGIN ECHO TransAction
END
```

```
EXEC myname
```

End of file `flocal`

File `fglobal` contains following CCL code:

```
GLOBAL PROCEDURE myname
BEGIN ECHO noname
END
```

End of file fglobal

```
ccl> EXEC fglobal           // define global procedure
ccl> EXEC myname           // call global block-defined procedure myname
noname
ccl> EXEC flocal           // define and execute local procedure
TransAction
ccl> EXEC myname
noname
ccl> LOCAL PROCEDURE myname
>BEGIN ECHO Myname
>END
ccl> EXEC myname
Myname
ccl> GLOBAL PROCEDURE myname // global myname is already defined
.. CCL reports an error ..
ccl> PROCEDURE display      // display is a keyword
.. CCL reports an error ..
ccl>
```

1.9.2 Local and Global Variables

In CCL variables are not explicitly declared but are created by the first assignment. By the keywords `LOCAL` or `GLOBAL` the lifetime and visibility of the variable which has to be created can be specified.

1.9.3 Creation of Variables

Assume one of the following expressions:

```
LOCAL v := <expr>          // or
      v := <expr>
```

If the expression occurs outside any block-defined procedure then a "file-local" variable `v` is created. The lifetime of `v` is the file where the expression occurs, i.e. the variable `v` lives until CCL leaves the file (or exits if CCL works interactively). If a file-local `v` already exists it is overwritten.

If the expression occurs inside a block-defined procedure *p* then a "procedure-local" variable *v* is created. The lifetime of *v* is the procedure where the expression occurs, i.e. the variable *v* lives until CCL leaves the procedure. If a procedure-local *v* already exists it is overwritten.

To create a global variable:

```
GLOBAL v := <expr>
```

This expression creates and sets a global variable *v* if there does not yet exist a global *v* otherwise overwrites the global *v*. A global variable has lifetime until CCL exits and can be accessed wherever it is not hidden by a local variable with the same name.

1.9.4 Resolution of Variable References

A variable is referenced by the expression `$var` where *var* is the name of the variable. Within this chapter, also the left hand of assignment "`var := ..`" counts as a reference of *var*.

For a reference of *var* which occurs outside any block-defined procedure (but inside a file *f* or in "stdin"), CCL tries to resolve the reference by a file-local variable *var*. If there is none, then a global variable *var* is searched. If there is none, CCL reports an error.

For a reference *var* which occurs inside a local(!) block-defined procedure *p* which resides in a file (say *f*), CCL first searches *var* within the corresponding procedure-local variables, then within the corresponding file-local variables. If neither in *p* nor in *f* a variable *var* is found then CCL tries to resolve within the global variables which are known at that point. If the resolution finally fails, CCL reports an error.

The expression `ISSET(var)` which tests if *var* is a defined variable works analogously in that it searches through the described hierarchy.

For global block-defined procedures, CCL makes variable resolution only within the procedure-local and global variables, i.e. only a 2 level variable hierarchy is supported. This is discussed in detail in Chapter "Restrictions and Properties of Global Procedures".

1.9.5 Restrictions and Properties of Global Procedures

Block-defined procedures declared `GLOBAL` have the following special properties: they cannot reference file-local variables and they cannot call local block-defined procedures.

In other words, inside a global block-defined procedure, CCL shortcuts the variable resolution from procedure-local directly to global and omits the file-local

variables, and analogously, for procedure name resolution, only searches within global procedures.

The reason for this mechanism is explained by the following example. Assume a file `badfile1`:

```
Begin of file badfile1

... some statements ...
LOCAL fli := 100
...
GLOBAL PROCEDURE pglob
BEGIN
...
... $fli          // error:  tries to reference file-local fli
...
END
...
End of file badfile1
```

A global procedure, by its nature, can be called from anywhere within a CCL program (after the CCL interpreter has seen the procedure). This means that in our example, after CCL has executed `badfile1`, `pglob` is directly callable from a file `badfile2`, i.e. by a call the control flow directly goes to `pglob` without entering `badfile1`. However the existence of file-local variables is coupled to the call of the file (i.e. CCL stacks them), so they do not exist if `pglob` is directly called.

Similar arguments hold for the call of local procedures inside the body of global procedures.

By this restriction, a global procedure can always be isolated from its file environment.

Thus, for global procedures it is best to place them exclusively into one or several files. For clarity, no other code than global procedures should be in these files. A CCL program which wants to use one or several of these procedures `EXECutes` the corresponding file(s) at the beginning (or `INCLUDEs` it as described later).

1.9.6 Examples

Example: (assumes that no variable `i` has been created)

```
ccl> GLOBAL i := 10          // create global i
ccl> ECHO $i
10
```

```

ccl> i := 20                // set global i
ccl> ECHO $i
20
ccl> LOCAL i := 200        // create file-local i
ccl> ECHO $i                // file-local i
200
ccl> i := 300              // set file-local i
ccl> ECHO $i
300
ccl> LOCAL PROCEDURE p
> BEGIN
> ECHO $i
> LOCAL i := 2000
> ECHO $i
>END
ccl> EXEC p
300                        // file-local
2000                       // procedure-local
ccl> ECHO $i                // file-local i
300
ccl> UNSET i                // unset file-local i
ccl> ECHO $i                // global i
20
ccl> UNSET i                // unset global i
ccl> ECHO $i
.. CCL reports an Error ..

```

The following example shows global and local procedures:

Begin of file pglobals :

```

GLOBAL PROCEDURE pr1(a,b)
BEGIN
...
END

```

```

GLOBAL PROCEDURE pr2(x,y)
BEGIN
...
END

```

End of file pglobals :

Begin of file main :

```
EXEC pglobs          // now all procedures of bglobs are callable

PROCEDURE ploc(a,b)
BEGIN
  ...$mainloc ..    // reference of file-local variable
END

mainloc := 5

...pr1(..)
...ploc(..)

End of file main:
```

1.10 Include and Define

CCL offers the ability to split large CCL scripts into several files and procedures. In these different files often some parameters are the same. The `INCLUDE` and `DEFINE` statement make it possible to hold constants in one include file which can be included by several CCL scripts. These constants are defined by the `DEFINE` statement. The usage of a constant requires a `#`-sign to distinguish it from a string.

Example:

File `common.inc`:

```
define no_error      0
define warning       1
define error         2
define fatal_error   3
```

File `module1`:

```
include common.inc

GLOBAL PROCEDURE do_it
BEGIN ...
```

```

    RETURN #no_error
END

```

Note that in defining constants only a subset of a CCL expression is available, namely procedure calls and dereferencing of variables. If a constant will be defined twice with different values then CCL reports an error.

Example:

```

ccl> define x    1024           // correct
ccl> define x    512           // error because different value
ccl> define x    1024         // correct because same value
ccl> define y    5 * $var      // error because variable
ccl> define z    myproc(par1) // error because procedure call

```

The two include files `tbx.ccl` and `tberror.ccl` in the Transbase directory contain the defines for constants dealing with the CCL Transbase interface.

1.11 SQL Statements

The CCL commands described in this chapter call a Transbase database via a Transbase kernel. CCL accesses the database via the TBX interface like any other TBX or ESQL applications. Therefore, the same rules apply as there are:

A `tbkernel` process must have been started. To process interactive abort signals (``, `^C`), a `tbserver` process must have been started.

1.11.1 Error Handling of SQL Statements

Each SQL statement may fail and produce an error. The reserved CCL variables `sql.error`, `sql.errtxt`, `sql.error.scr`, are used for error handling: Whenever an SQL statement produces an error, a Transbase errorcode (a number $\neq 0$) is written into `sql.error`, a textual error message is written into `sql.errtxt` and if variable `sql.error.scr` is set the contents of this variable is interpreted as CCL commands.

The default behaviour of CCL is as follows: Whenever a SQL statement produces an error, CCL outputs an error message including the Transbase error message (`sql.errtxt`) on `stderr` and then finishes if called non-interactively else returns to the main loop.

The CCL programmer can override this default behaviour by setting the CCL variable `sql.error.scr` with a CCL script. After each SQL statement which has

produced an error, CCL inspects the variable `sql.error.scr`. If it has never been set inside the CCL program, then the default mechanism as described above takes place. Otherwise, CCL performs the statements which are stored in `sql.error.scr` and then continues with the next statement unless the error script has changed the control flow. Note that it is a difference whether `sql.error.scr` has not been set or has been set with the value `{}`. In the first case, CCL default behaviour is valid, in the second case the error is ignored and the next statement is executed. To get back the default behaviour the variable `sql.error.scr` has to be deleted by the `UNSET` command.

The following example shows a user controlled error handling:

```
ccl> sql.error.scr := { errorcount := \$errorcount+1
                      ECHO $sql.errtxt }
```

In this example, the error message is printed and a counter is increased to count all SQL errors which occur. Note the backslashes in the script: they are necessary to avoid variable substitution at the time of setting the variable `sql.error.scr`. Instead variable substitution occurs at the time the error script is performed.

1.11.2 Connect, Disconnect and Login

To run statements against a database `db`, one has to `CONN`ect to `db` and to `LOGIN` in `db`. If one wants to finish the session one must disconnect from the connected database(s) (the transaction must be finished before one disconnects). Assume there is a database `sampledb` on a host machine named `hostdb` and a user `smith` with password `mary` wants to connect:

```
ccl> CONN {sampledb@hostdb}    // need braces because of @
ccl> LOGIN smith mary        // full LOGIN command
    .. or ..
ccl> LOGIN smith
    .. here CCL prompts for the password and the terminal does not echo
    .. do CCL statements ..
ccl> DCONN {sample@hostdb}    // disconnect
```

There is a subtle difference in the full `LOGIN` command and the prompting version: if the user has special characters in his password then in the full `LOGIN` command the password must be written as delimiter string, i.e. with braces. For example the empty password must be written as `{}`. In the prompting version, no additional braces must be written.

1.11.3 Transaction Handling (CT, AT)

There is no command to explicitly open a transaction. Whenever a SQL statement is executed, CCL automatically opens a transaction unless a transaction is already open. The commands CT and AT commit or abort the transaction.

```
ccl> CT                // commit the current transaction
ccl> AT                // abort the current transaction
```

The builtin function TASTATE() retrieves the status of the actual transaction. This function can be used to check if an open transaction has been aborted by the kernel after an error. The return values of TASTATE() correspond to the constants TA_UNDEF, TA_ABORTED etc. defined in the file tbx.ccl which resides in the TRANSBASE directory.

If CCL is ended without committing the transaction and without disconnecting from the database, then CCL aborts the transaction and disconnects.

1.11.4 Multi Database Handling

If a CONNect has been issued. then a further connect to the same database is possible and has no effect in that case. However, one can be connected to more than one database (let's say db1 and db2). The current database is the one where the last CONNect has been issued and one can change the current database by issuing a further connect. CCL always sends SQL statements to the current database.

If in a DCONN statement a database name is specified CCL tries to disconnect from this database. This attempt may fail if a transaction is open on this database. A CT or AT statement has to be executed to allow disconnecting from the specified database.

If the DCONN statement is invoked without a database name then CCL tries to disconnect from all previously connected databases. There also may occur errors if a transaction is open, but all databases which are not affected by an open transaction will be closed.

CCL offers the builtin function DBSTATE() to check the connect status with respect to a database. If DBSTATE() is called without parameters then the function retrieves the state of the actual connected database. Optional the function DBSTATE() can be called with a database name as first parameter. In this case the database state of the specified database will be returned.

The return values of DBSTATE() correspond to the constants DB_DCONN, DB_CONTACTED etc. defined in the file tbx.ccl which resides in the TRANSBASE directory.

Example:

```

ccl> CONN db1                // actual database is db1
ccl> CONN db2                // actual database is now db2
ccl> COON db1                // switch to database db1
ccl> LOGIN user
Password:
ccl> ECHO DBSTATE(db2)      // db2 is not actual database
4
ccl> DCONN db2              // disconnect from unused database
ccl> RUN {select count(*)   // db1 is still the
>      from systable}      // actual database
ccl> DCONN                  // disconnect from all databases will fail
                             // because a transaction is open on db1

.. CCL will report an error ..
ccl> AT
ccl> DCONN
ccl>

```

1.11.5 SELECT Statements, Sample Program

For running **SELECT** queries which deliver a set of tuples, CCL offers convenient constructs: one **OPENS** a query under a special name (e.g. name **cur**), **FETCHes** a result row from **cur**, can test if **cur** is exhausted (**EOD(cur)**), and finally **CLOSEs** **cur** (after a loop of **FETCHes**). As long as a result tuple is available, one can access single fields of the result tuple via the query name **cur** and the position. This means that variables **cur.0**, **cur.1**, **cur.2** etc. are declared and can be accessed via **\$cur.0** etc. These are called **positional variables**. Positional variables can also be accessed via **\$cur.fieldname**.

Example:

```

ccl> OPEN c {
>SELECT   tname,segno FROM systable
>        WHERE tname = 'systable'
>}
ccl> FETCH c
ccl> ECHO $c.tname && $c.segno
systable 1

```

Additionally, as long as the query **cur** is open, the variable **cur.fieldno** contains the number of result fields of the query **cur**. The variable **cur.eod** indicates if there are still more tuples and its values correspond to the C-Macros

NO_TUPLE, ONE_TUPLE and MORE_TUPLES defined in the file `tbx.h`. The variable `cur.updatable` indicates if this query could also be opened as a FOR-UPDATE query. The variable `cur.qtype` indicates the type of the query and corresponds to the C-Macros for the query types defined in `tbx.ccl`.

Additionally the builtin functions `DDL_CLASS()` through `LOAD_CLASS()` indicate the type of an open query.

Below, a sample program shows the principles. It reads in a database name, connects and logs in. Then a tablename is read in and a `SELECT` query referring to the table is opened. The program successively reads all tuples and counts the number of tuples which at least contain one `NULL` value. Finally the tuplecount is printed.

```
// Sample Program:

ACCEPT db PROMPT {db:}
CONN $db
LOGIN tadmin {}
    // prompt for tablename
ACCEPT tablename PROMPT {tablename:}
selquery := { SELECT *
              FROM $tablename }
    // init tuple counter, open query and fetch first tuple
tuples := 0
OPEN cur $selquery
FETCH cur
// test whether the last FETCH has delivered a tuple
WHILE NOT EOD(cur)
DO
    fields := 0
    // loop over all result fields
    WHILE $fields NE $cur.fieldno
    DO
        // test on SQL NULL value by builtin procedure
        IF ISNULL(cur.$fields)
        THEN tuples := $tuples+1
            BREAK
        FI
        fields := $fields+1
    OD
    FETCH cur
OD
CLOSE cur
AT
```

```

DCONN $db
ECHO { $tuples tuples had a NULL value }

```

Note that in OPEN, FETCH, EOD, CLOSE the query name cur must not be dereferenced (cur is not a variable) whereas the usage of cur.fieldno, cur.0 etc. must be like that of a variable.

Note furthermore that SQL NULL values are represented as string NULL in CCL variables, but the builtin procedure ISNULL yields true if the variable contains a NULL value. See more about value representation in "Value Representation in CCL and Transbase SQL".

1.11.6 SQL Statements executed by RUN

A number of SQL query types can be executed with a simple RUN construct. This includes the following query types: INSERT, UPDATE, DELETE, UPDATE POSITIONED, DELETE POSITIONED Statements, SPOOL statements, all DDL statements, LOCK statements and LOAD statements. See the Transbase manuals for the definition of the RUN statements.

Example:

```

ccl> RUN { DELETE FROM t WHERE ti = 5 }
ccl> ECHO { $run.ntuples deleted }
ccl> RUN { INSERT INTO t SELECT * FROM v }
ccl> ECHO { $run.ntuples inserted }
ccl> RUN { UPDATE t SET ti = ti+1 WHERE tj = 10 }
ccl> ECHO { $run.ntuples updated }
ccl> RUN { SPOOL INTO spoolfile SELECT * FROM systable }
ccl> ECHO { $run.ntuples spooled }
ccl> RUN { CREATE TABLE t (t1 INTEGER, t2 CHAR(5) ) }
ccl> RUN { LOCK suppliers EXCLUSIVE }
ccl> RUN { LOAD DISK CACHE TABLE t }

```

The execution of a RUN command implicitly defines two variables namely **run.ntuples** and **run.tried**. The variable run.ntuples contains the number of tuples which have actually been deleted, inserted, updated, spooled. The variable run.tried contains the number of tuples for which the operation has been tried (might be bigger than run.ntuples if some duplicate tuples had to be ignored). For DDL, LOCK and LOAD queries both variables are undefined.

1.11.7 Update and Delete positioned

The RUN statement also is used to delete or update the current tuple of a cursor ("delete positioned" and "update positioned").

Example:

```

OPEN c { SELECT * FROM table FOR UPDATE }
...
FETCH c

IF    ...
THEN  RUN    { UPDATE table SET field = new value
              WHERE CURRENT OF c }
        // update actual tuple of cursor c

ELSE  RUN    { DELETE WHERE CURRENT OF c }
        // delete actual tuple in cursor c
FI

```

Finally, the RUN construct can be used as a shorthand concept to evaluate SELECT queries which deliver only 1 tuple or to evaluate the first tuple only of a general SELECT query.

```

ccl> RUN { SELECT * FROM suppliers WHERE suppno=50 }
ccl> IF EOD(run)
>THEN ECHO { no tuple }
>ELSE ECHO { $run.0  $run.1 } FI
.. CCL prints 2 field values if there is a tuple ..

```

Here, the RUN delivers the first tuple if the query delivers any tuple, EOD tests if there is a tuple delivered and the variables run.0, run.1 etc. as well as run.fieldno are defined as if "run" were a name of an open query. However, it is illegal to CLOSE such a query.

Note carefully that one RUN statement overwrites the result (i.e. values of variables) of the last RUN statement.

1.11.8 Value Representation in CCL and TransbaseSQL

Whenever a tuple of a SELECT query is fetched (via FETCH or via RUN), an implicit assignment of the field values to positional variables occurs (as described in chapter "SELECT Statements,..").

As all CCL variables are of type string, the Transbase SQL values are converted to a string representation. Note that this representation is not necessarily the SQL literal representation. Non-compatibilities occur in case of the types CHAR(), BINCHAR(), DATETIME and TIMESpan. For example, a string literal in SQL is single quoted (e.g. 'Smith') whereas the CCL representation is not quoted, because it is

undesired in the application. Likewise, the representation of `DATETIMES` in CCL is `1992-10-5` etc. which is in contrast to the somewhat syntax oriented TB/SQL literal `DATETIME[YY:DD]` (`1992-10-5`).

Therefore, in parameterized SQL statements like

```
ccl> RUN { SELECT * FROM T WHERE tname = $s }
```

where `tname` is a string field, it is somewhat awkward to generate the SQL literal representation in variable `s`; assuming that variable `s` holds the search value `Smith` (e.g. read in via `ACCEPT`), one must precede the above SQL statement by the following statement to surround the value by single quotes:

```
ccl> s := {'$s'}
```

Another possibility is to write directly:

```
ccl> RUN { SELECT * FROM T WHERE tname = '$s' }
```

For SQL `NULL` values in result tuples, the default representation is the string `NULL`. CCL also offers a possibility to explicitly define a CCL string representation for SQL `NULL` values. CCL distinguishes between three different representations of `NULL` values. For this purpose the values of the three variables `sql.null`, `sql.numeric.null` and `sql.string.null` can be set to define a specific CCL representation of SQL `NULL` values.

For the SQL fieldtypes `TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC`, `FLOAT` and `DOUBLE` the CCL `NULL` value representation is defined by the variable `sql.numeric.null`.

For the SQL fieldtypes `CHAR`, the CCL `NULL` value representation is defined by the variable `sql.string.null`.

For all other SQL fieldtypes the CCL `NULL` value representation is defined by the variable `sql.null`.

1.12 TYPE and DISPLAY: Output of Result Tables

The commands `TYPE` and `DISPLAY` both accept an arbitrary `SELECT` query as argument.

`TYPE` outputs the whole result table with column names as headline. `DISPLAY` outputs part of the result table and then prompts for continuation. The number of tuples printed per output portion and the width of the output depends on

the variables `sql.win.width`, `sql.win.height` which can be set by the CCL programmer.

Furthermore, the variable `sql.string.width` can be set to determine the number of characters printed in fields with variable sized character strings (`CHAR(*)`, `BINCHAR(*)`, `BITS(*)`).

If the variable `sql.header` is not set the `TYPE` and `DISPLAY` statement both will print a header. The variable `sql.header` can be set to 0 (`FALSE`) to suppress the headlines in the `DISPLAY` and `TYPE` statement. If the variable `sql.field.sep` is not set the default field separator for the `DISPLAY` and `TYPE` statement will be '|', else the variable `sql.field.sep` contains a user defined field separator.

1.13 I/O Redirection and Pipes

CCL knows two commands for redirecting the stdout output to a file. `REDIRECT` causes output to be written into a newly created file, `APPEND` appends into a existing file if it exists otherwise creates it.

```
ccl> v := {/tmp/ccl}
ccl> REDIRECT $v
ccl> TYPE { SELECT * FROM .... } // all output is written ..
ccl> ECHO ... // into file /tmp/ccl
ccl> REDIRECT // output on stdout again
ccl> ..
ccl> APPEND $v // now output is appended to /tmp/ccl
ccl> TYPE { SELECT * FROM .... } // file /tmp/ccl
ccl> APPEND // output on stdout again
```

`REDIRECT` or `APPEND` without argument switch off the redirection and make stdout again valid as output stream. If a `REDIRECT` or `APPEND` with argument are given when another `REDIRECT` or `APPEND` are still active then an implicit close is performed.

Similarly, the `PIPE` command takes one argument, interprets it as a shell command and directs all further stdout output of CCL into the stdin of the command. `PIPE` without any argument closes the pipe and directs the output of CCL back to stdout.

```
ccl> PIPE wc
ccl> TYPE { SELECT * FROM .... } // output to wc
ccl> ..
ccl> PIPE
.. output of wc on stdout ..
```

```
ccl> PIPE { wc > /tmp/x }
ccl> TYPE { SELECT * FROM .... } // output to wc
ccl> ..
ccl> PIPE
.. output of wc into file ..
```

1.14 Exit to Shell

The command `SH` exits to the Shell (which Shell is started depends on the Environment variable `SHELL`). Supplied with an argument (expression) it calls the shell to execute the argument.

```
ccl> SH
% ...
% ^D
ccl> SH {vi x}
.. edit file ..
ccl>
```

1.15 EDIT a File

The command `EDIT` calls the editor which is denoted in the shell variable `EDITOR` (editor `vi` if `EDITOR` is not set). Optionally it has one CCL expression as argument.

```
ccl> EDIT {x y z} // call editor with files x, y, z
```

1.16 Change Working Directory

The command `CD` changes the working directory of CCL. This may be of importance for the commands `EXEC`, `EDIT`. If no argument is supplied, `CD` changes to the home directory.

```
ccl> CD // change to home directory
ccl> CD {/tmp}
ccl> EDIT x
...
ccl> EXEC x
ccl> CD
```

1.17 Builtin Procedures

CCL offers several builtin procedures to make the handling of variables, strings and cursors easier. The names for the builtin procedures are treated as keywords, so they are case insensitive. In the following, all procedure names are written in bold upper case, all parameters are written in lower case and are underlined>.

1.17.1 System Environment

GETENV(string) returns the contents of environment variable string if exists; empty string else.

Example:

```
ccl> ECHO GETENV( TRANSBASE )  
/usr/transbase/tb  
ccl>
```

GETPID() returns the process id of current CCL process.

Example:

```
ccl> ECHO GETPID( )  
123  
ccl>
```

GETCWD() returns the current working directory

Example:

```
ccl> ECHO GETCWD( )  
/usr/transbase  
ccl>
```

1.17.2 Variable Handling

ISNULL(varname) returns 1 if the variable `varname` is a SQL null value and 0 in all other cases.

Example:

```
ccl> i := 5
ccl> ECHO ISSET( i )
1
ccl>
```

ISSET(varname) returns 1 if the variable **varname** exists and 0 in all other cases.

Example:

```
ccl> RUN {SELECT NULL FROM systable}
ccl> ECHO ISNULL( run.0 )
1
ccl>
```

1.17.3 String Handling

CCL counts characters in strings from 1 to n where n is the last character.

STRLEN(string) returns string length of **string**

Example:

```
ccl> ECHO STRLEN( TransActionSoftware )
19
ccl> ECHO STRLEN( {} )
0
ccl>
```

SUBSTR (string , n) returns substring of string from the n-th character to the end if n is less than 1 i will be set to 1 if n is greater than [STRLEN(string) n will be set to **STRLEN**(string)

Example:

```
ccl> ECHO SUBSTR( TransActionSoftware , 6 )
ActionSoftware
ccl> ECHO SUBSTR( TransActionSoftware , 1000 )

ccl> ECHO SUBSTR( TransActionSoftware , -5 )
TransActionSoftware
ccl>
```

SUBSTR (*string* , *n* , *m*) returns substring of *string* from the *n*-th to the *m*-th character if *m* is less than *n*, *m* will be set to **STRLEN**(*string*) if *m* is greater than **STRLEN**(*string*) *m* will be set to **STRLEN**(*string*)

Example:

```
ccl> ECHO SUBSTR( TransActionSoftware , 6 , 11 )
Action
ccl> ECHO SUBSTR( TransActionSoftware , 6 , 5 )
ActionSoftware
ccl> ECHO SUBSTR( TransactionSoftware, 6 , 1000 )
ActionSoftware
ccl>
```

STRSTR (*string* ,*search*) returns first occurrence of *search* found in *string*
0 means no occurrence found

Example:

```
ccl> ECHO STRSTR( TransActionSoftware , Action )
6
ccl> ECHO STRSTR( TransActionSoftware , action )
0
ccl>
```

LOWER (*string*) returns *string* with all uppercase letters transformed to lowercase letters

Example:

```
ccl> ECHO LOWER( TransActionSoftware )
transactionsoftware
ccl>
```

UPPER (*string*) returns *string* with all lowercase letters transformed to uppercase letters

Example:

```
ccl> ECHO UPPER( TransActionSoftware )
TRANSACTIONSOFTWARE
ccl>
```

TRANSLATE (*string*, *from*, *to*) returns *string* after replacing all characters in *from* with the corresponding characters in *to*. If there is no corresponding character the matching characters are deleted.

Example:

```
ccl> ECHO TRANSLATE( TransActionSoftware , TAS , 123 )
1rans2ction3oftware
ccl> ECHO TRANSLATE( TransActionSoftware , TAS , X )
Xransctionoftware
ccl>
```

TRIM (*string*) returns *string* after removing all leading and trailing blanks

Example:

```
ccl> ECHO X & TRIM( {   TransActionSoftware   } ) & X
XTransActionSoftwareX
ccl>
```

TRIM (*string*, *set*) returns *string* after removing all leading and trailing occurrences of characters in *set*

Example:

```
ccl> ECHO TRIM( 0815TransActionSoftware4711 , 0123456789 )
TransActionSoftware
ccl>
```

LTRIM (*string*) returns *string* after removing all leading blanks

Example:

```
ccl> ECHO X & LTRIM( {   TransActionSoftware   } ) & X
XTransActionSoftware   X
ccl>
```

LTRIM (*string*, *set*) returns *string* after removing all leading occurrences of characters in *set*

Example:

```
ccl> ECHO LTRIM( 0815TransActionSoftware4711 , 0123456789 )
TransActionSoftware4711
ccl>
```

RTRIM (string) returns `string` after removing all trailing blanks

Example:

```
ccl> ECHO X & RTRIM( { TransActionSoftware } ) & X
X TransActionSoftwareX
ccl>
```

RTRIM (string, set) returns `string` after removing all trailing occurrences of characters in `set`

Example:

```
ccl> ECHO RTRIM( 0815TransActionSoftware4711 , 0123456789 )
0815TransActionSoftware
ccl>
```

1.17.4 Database and Transaction Handling

The return codes of the following functions refer to constants in the file `tbx.ccl` which resides in the `TRANSBASE` directory and can be included.

DBSTATE () returns Status of actual connected Database

Example:

```
ccl> CONN {sample@host}
ccl> ECHO DBSTATE( )
2
ccl>
```

DBSTATE (dbname) returns Status of database with name `dbname`

Example:

```
ccl> CONN {sample@host}
ccl> ECHO DBSTATE( {sample@host} )
2
ccl>
```

TASTATE () returns State of actual transaction

Example:

```
ccl> ECHO TASTATE( )
1
ccl>
```

1.17.5 Cursor Handling

EOD (cursor) returns 1 if cursor has no more tuples; 0 in all other cases.

Example:

```
ccl> OPEN c {select distinct 'TransActionSoftware' from systable}
ccl> FETCH c
ccl> ECHO EOD( c )
0
ccl> FETCH c
ccl> ECHO EOD( c )
1
ccl>
```

DDL_CLASS (cursor) returns 1 if cursor is opened with a DDL-Statement; 0 in all other cases

Example:

```
ccl> RUN { create table t ( f1 integer )}
ccl> ECHO DDL_CLASS( run )
1
ccl>
```

DML_CLASS (cursor) returns 1 if cursor is opened with a DML-Statement; 0 in all other cases

Example:

```
ccl> RUN { insert into t ( 5 ) }  
ccl> ECHO DML_CLASS( run )  
1  
ccl>
```

SEL_CLASS (cursor) returns 1 if cursor is opened with a SELECT-Statement; 0 in all other cases.

Example:

```
ccl> OPEN c {select * from t}  
ccl> ECHO SEL_CLASS( c )  
1  
ccl>
```

UPD_CLASS (cursor) returns 1, if cursor is opened with a UPDATE-Statement; 0, in all other cases

Example:

```
ccl> RUN { update t1 set f1 = 10 }  
ccl> ECHO UPD_CLASS( run )  
1  
ccl>
```

SPOOL_CLASS (cursor) returns 1 if cursor is opened with a SPOOL-Statement; 0 in all other cases

Example:

```
ccl> RUN { spool into file_t select * from t}  
ccl> ECHO SPOOL_CLASS( run )  
1  
ccl>
```

LOCK_CLASS (cursor) returns 1 if cursor is opened with a LOCK-Statement; 0 in all other cases.

Example:

```
ccl> RUN { lock t read }
ccl> ECHO LOCK_CLASS( run )
1
ccl>
```

LOAD_CLASS (cursor) returns 1 if cursor is opened with a LOAD-Statement; 0 in all other cases.

Example:

```
ccl> RUN { load disk cache table t }
ccl> ECHO LOAD_CLASS( run )
1
ccl>
```

Chapter 2

CCL Start and Arguments

Syntax: (UNIX)

```
ccl [-x] [-c command] [ [progfile [ arguments ] ] ]
```

Syntax: (Windows)

```
wincccl [-x] [-c command] [ [progfile [ arguments ] ] ]
```

Effect: With the `-x` option, substituted variables are traced on stderr.

With the `-c` option, the successive argument directly is interpreted as a CCL program.

If a progfile is not specified then CCL enters interactive mode otherwise CCL takes one progfile as program to be executed and the arguments if specified become parameters to the program. The progfile variant only works without the `-c` option.

Exit Code: CCL exits with the argument value of the CCL `EXIT` command if there is one given else with value 0.

Example: `ccl -x` enters interactive mode with debug output.

`ccl pf 123 456` executes programfile `pf` with parameters 123 and 456.

`ccl -c 'EXEC pf(123,456)'` executes programfile `pf` with parameters 123 and 456.

`{ccl -c 'sql.error.scr := {} EXEC pf(123,456)'` executes programfile `pf` with parameters 123 and 456 but ignore errors in database access.

Chapter 3

CCL Grammar

3.1 Conventions for Syntax Notation

Bracket [] are delimiters for an optional part.

The vertical line | separates alternatives.

An ellipsis ... indicates that the preceding item may be repeated arbitrarily often.

To distinguish terminal from non-terminal symbols, all non-terminal symbols are surrounded by angle brackets < > and are written in lowercase, all terminal symbols are represented by themselves.

All keywords are written in uppercase letters.

3.1.1 Productions

3.1.2 Program

```
<program> ::= <block>
<block>   ::= <stat> <block>
           | <stat>
<stat>    ::= <echo_stat>
           | <set_stat>
           | <unset_stat>
           | <proc_stat>
           | <accept_stat>
           | <if_stat>
           | <do_while_stat>
           | <while_do_stat>
           | <break_stat>
           | <continue_stat>
```

```

|      <exec_stat>
|      <return_stat>
|      <append_stat>
|      <redirect_stat>
|      <pipe_stat>
|      <shell_stat>
|      <edit_stat>
|      <cd_stat>
|      <database_stat>
|      <exit_stat>
|      <include_stat>
|      <define_stat>

```

3.1.3 I/O Statements

```

<echo_stat> ::= ECHO <expr>

<accept_stat> ::= ACCEPT <target_expr> [ PROMPT <expr> ]

<target_expr> ::= <expr>

```

3.1.4 Handling of Variables

```

<set_stat> ::= [ LOCAL | GLOBAL ] <target_expr> := <source_expr>
<target_expr> ::= <expr>
<source_expr> ::= <expr>
<unset_stat> ::= UNSET <target_expr>
<target_expr> ::= <expr>

```

3.1.5 Procedure Handling

```

<proc_stat> ::= [ LOCAL | GLOBAL ] PROCEDURE
               <proc_name> [ ( [ <par_list> ] ) ]
               BEGIN <block> END

               <proc_name> ::= <var_name>

<par_list> ::= <param> [ , <par_list> ]

               <param> ::= <var_name>

<exec_stat> ::= EXEC <file_name_expr> [ <parameter_list> ]

```

<file_name_expr> ::= <expr>

<parameter_list> ::= ([<expr> [, <expr>]] ..)

<return_stat> ::= RETURN [<expr>]

3.1.6 Control Structures

<if_stat> ::= IF <expr> THEN <block> [ELSE <block>] FI

<do_while_stat> ::= DO <block> OD WHILE <expr>

<while_do_stat> ::= WHILE <expr> DO <block> OD

<break_stat> ::= BREAK

<continue_stat> ::= CONTINUE

<exit_stat> ::= EXIT [<expr>]

3.1.7 Redirection Statements

<append_stat> ::= APPEND [<file_name_expr>]

<file_name_expr> ::= <expr>

<redirect_stat> ::= REDIRECT [<file_name_expr>]

<file_name_expr> ::= <expr>

<pipe_stat> ::= PIPE [<command_expr>]

<command_expr> ::= <expr>

3.1.8 System Statements

<shell_stat> ::= SH [<command_expr>]

<command_expr> ::= <expr>

<edit_stat> ::= EDIT [<file_name_expr>]

```

        <file_name_expr> ::= <expr>

<cd_stat> ::=          CD [ <dir_name_expr> ]

        <dir_name_expr> ::= <expr>

```

3.1.9 Include and Define Statement

```

<include_stat> ::=          INCLUDE <file_name_expr>

        <file_name_expr> ::= <expr>

<define_stat> ::=          DEFINE <const_name> <expr>

        <const_name> ::= <non_delimiter_string>

```

3.1.10 Comparison Operators

```

<comp_op> ::=              <lex_compop>
                          |              <num_compop>

<lex_compop> ::=          =
                          |          <>
                          |          <
                          |          >
                          |          <=
                          |          >=

<num_compop> ::= EQ
                |   NE
                |   LT
                |   GT
                |   LE
                |   GE

```

3.1.11 Numerical Operators

```

<dual_op> ::=             <add_op>
                          |             <mult_op>

<add_op> ::=             +

```

		-
<mult_op> ::=		*
		/
		%
<unary_op> ::=		+
		-
		NOT

3.1.12 Concatenation Operators

<concat_op> ::=	&&
	&

3.1.13 Expression Syntax

<expr> ::=	<bool_term> [OR <expr>]
	<bool_term> [XOR <expr>]
<bool_term> ::=	<bool_factor> [AND <bool_term>]
<bool_factor> ::=	NOT <bool_factor>
	<comp_expr>
	(<expr>)

3.1.14 Comparison Expression

<comp_expr> ::=	<str_expr> [<comp_op> <str_expr>]
<str_expr> ::=	<num_expr> [<concat_op> <str_expr>]

3.1.15 Syntax of Numerical Expressions

<num_expr> ::=	<term> <add_op> <num_expr>
	<term>
<term> ::=	<factor> <mult_op> <term>
	<factor>
<factor> ::=	<proc_call>
	(<str_expr>)
	<non_delimiter_string>

```

|           <delimiter_string>
|           <var_deref>
|           <const_deref>
|           <unary_op> <factor>

```

3.1.16 Procedure Call within Expression

```

<proc_call> ::=          <pname_expr> ( [ <expr_list> ] )

<expr_list> ::=         <expr> [ <expr_list> ]

<pname_expr> ::=       (<expr> )
|                       <const_deref>
|                       <var_deref>
|                       <non_delimiter_string>
|                       <delimiter_string>

```

3.1.17 Syntax of Strings

```

<non_delimiter_string> ::= //   non-empty string of
                             letters, digits, dot and underscore

<delimiter_string> ::=    //   (possibly empty) string of
                             all printable characters

```

3.1.18 Constants and Variables

```

<var_deref> ::=         $ <var_name>
|                       $ ( <var_deref> ... )
|                       <non_delimiter_string>

<var_name> ::=         <non_delimiter_string>

<const_deref> ::=     # <const_name>

<const_name> ::=     <non_delimiter_string>

```

3.1.19 Database Statements

```

<database_stat> ::=    <conn_stat>
|                       <login_stat>
|                       <ta_stat>

```

```

|           <open_stat>
|           <fetch_stat>
|           <close_stat>
|           <run_stat>
|           <display_stat>
|           <type_stat>

```

3.1.20 Connections, Login and Transactions

```

<conn_stat> ::=          CONN <expr>
|              DCONN [ <expr> ]

<login_stat> ::=          LOGIN [ <user_name> [ <passwd> ] ]

    <user_name> ::= <expr>

    <passwd> ::= <expr>

<ta_stat> ::=          CT
|              AT

```

3.1.21 Evaluation of Queries

```

<open_stat> ::=          OPEN <query_name> <query>

    <query> ::= <delimiter_string>

<fetch_stat> ::=          FETCH <query_name>

    <query_name> ::= <expr>

<close_stat> ::=          CLOSE <query_name>

    <query_name> ::= <expr>

<run_stat> ::=          RUN <query>

    <query> ::= <delimiter_string>

```

3.1.22 Output of Query Results

```

<display_stat> ::=          DISPLAY <query>

```

```
          <query> ::= <delimiter_string>
<type_stat> ::=          TYPE <query>
          <query> ::= <delimiter_string>
```

Chapter 4

Current Limitations of CCL

CCL 1.1 does not support the following features which are expressed in terms of the TBX programming interface:

- Stored Queries
- User-defined Sortorders (`GET_SORTORDER`, `SET_SORTORDER`)
- Tbmode Statements
- `SET_DAT_DIR`
- `SET_TIME_OUT`
- `SET_CONSISTENCY`
- Handling of Blobs (`GETBLOB`, `SETBLOB`)