DBPA (“**D**ata**B**ase **P**rocess **A**utomation”) is a scripting language and run-time environment to run database process automation scripts. The language is defined in this document using BNF notation.

Upper case indicates a keyword. In the actual language, keywords and variable names are not case sensitive.

Lower case indicates a symbol that has a further definition.

Symbols enclosed in square brackets indicate an [ optional section ]. Symbols enclosed in braces (curly brackets) indicate a mandatory section with { first alternative | second alternative | etc. } If one of the alternatives is the default, it is underlined.

Other punctuation is literal, e.g., comma, parenthesis, equal sign, greater than, less than.

# Process

process ::= [ parameters ] [ variables ] tasks

# Parameters

parameters ::= PARAMETERS varname typename [ , … ] END PARAMETERS

typename ::= inttype | chartype | datetype

inttype ::= TINYINT | INT | INTEGER

chartype ::= { VARCHAR | CHAR | CHARACTER }[( integer )]

datetype ::= DATETIME | DATE

**Notes:**

There is only one INTEGER type. The other spellings are notational conveniences only.

There is only one VARCHAR type. The other spellings are notational conveniences only, as is the length indicator in parentheses, which is ignored.

There is only one DATETIME type. The DATE spelling is a notational convenience only.

Parameters behave exactly like all other variables. They only differ in that their values can be set by the calling program.

Parameters values are set in the order they appear in the PARAMETERS section from values provided by the calling program in order. If the calling program provides more values than parameters in the PARAMETERS section, extra values are ignored. If the calling program provides fewer values than parameters in the PARAMETERS section, trailing parameter values are set to NULL.

# Variables

VARIABLES varname typename [, …] END VARIABLES

**Notes:**

Variable values are initially set to NULL. Variable values can be set by SET and UPDATE tasks.

# Connections

CONNECTIONS conname contypename [, …] END CONNECTIONS

contypename ::= DATABASE | FTP | EMAIL

# Tasks

tasks ::= task [ task2 … ]

task ::= TASK [ taskname ] [ AFTER predecessors ] [ IF condition ] taskbody END TASK

predecessors ::= [ taskname ] [ completion ] [ { AND | OR } predecessor … ]

predecessor ::= taskname [ completion ]

completion ::= SUCCEEDS | FAILS | COMPLETES

condition ::= boolexpression

boolexpression ::= andexpression [ OR andexpression ]

andexpression ::= notexpression [ AND notexpression]

notexpression ::= [ NOT ] { notexpression | boolterm }

boolterm ::= ( boolexpression ) |  
expression compare compatexpression |  
expression IS [ NOT ] NULL

compare ::= < | <= | = | <> | >= | >

taskbody ::= set | update | run | create | append | write | open | load | read | zip | unzip | put | email | delete | rename | copy | make | log | go | fail | process | do | for | on | waitfor | connect

**Notes:**

If a task is given a taskname, the name must be unique within its task set. All tasks at the outer nesting level form a task set. All tasks nested at the same level within the same enclosing task form a task set. If a task is not given a taskname, it is anonymous and can only be used as a predecessor to the immediately next task in the list of tasks.

If the taskname of the first predecessor is omitted, then the predecessor is the immediately preceding task in the list of tasks, which may be anonymous.

All predecessors of a task must be uniformly joined by AND or OR. No mixing of AND and OR is allowed.

Predecessor tasks must be defined in the task set prior to their appearance in another task’s AFTER list. Forward references are not allowed.

compatexpression must be comparison compatible with expression. Expressions of the same type are compatible. Also, chartype and datetype expressions are compatible – the chartype result is interpreted as a datetime. If it cannot be interpreted as a datetime, the comparison causes the task to fail.

When evaluating boolterm, if either expression or compatexpression evaluates to NULL, then boolterm evaluates to false regardless of the compare operator.

If a condition is specified and evaluates to true, the task body executes. If a condition is specified and evaluates to false, the task body does not execute and the task immediately ends with SUCCEEDS status.

# Task Types

## SET

set ::= SET assignment [, assignment2 … ]

assignment ::= varname = expression

expression ::= intexpression | charexpression | dateexpression

intexpression ::= addend [ { + | - } addend ]

addend ::= factor [ { \* | / | % } factor ]

factor ::= [ { + | - } … ] { factor | intterm }

intterm ::= integer |  
NULL |  
ISNULL(intexpression, intexpression) |  
IIF(boolexpression, intexpression, intexpression) |  
DATEPART(datepart, dateexpression) |  
intvarname |  
( intexpression )

datepart ::= YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

charexpression ::= charterm [ + charterm ]

charterm ::= ‘quoted text’ |  
NULL |  
ISNULL(charexpression, charexpression) |  
IIF(boolexpression, charexpression, charexpression) |  
FORMAT(formatable, formatstring) |  
charvarname |  
( charexpression )

formatable ::= intexpression | dateexpression

formatstring ::= stringexpression

dateexpression::=  
NULL |  
ISNULL(dateexpression, dateexpression) |  
IIF(boolexpression, dateexpression, dateexpression) |  
GETDATE() |  
DATEADD(datepart, intexpression, dateexpression) |  
DATEFROMPARTS(intexpression, intexpression, intexpression) |  
DATETIMEFROMPARTS(intexpression, intexpression, intexpression, intexpression, intexpression, intexpression, intexpression) |  
( dateexpression ) |  
charexpression |  
datevarname

**Notes:**

Currently when operating on an intexpression the FORMAT function ignores the formatstring and uses a default format to render the integer as a character string.

When operating on a dateexpression the FORMAT function interprets formatstring according to Java DateTimeFormatter.ofPattern(formatstring).

When a charexpression appears in a dateexpression, the charexpression result is interpreted as a datetime. If it cannot be interpreted as a datetime, the SET task fails at the point of the assignment containing the expression. Previous assignments within the SET task, if any, complete normally but subsequent assignments within the SET task do not execute.

## UPDATE

update ::= UPDATE varname [, varname2 … ] [ FROM ] outstatement

outstatement ::= [ dbconname ] { statement | paramstatement }

dbconname ::= { [ DATABASE ] conname } | { DEFAULT [ DATABASE ] }

statement ::= STATEMENT charexpression

paramstatement ::= SQL [ SUBSTITUTING expression [ , expression2 … ] [INTO] ] paramquery

**Notes:**

The indicated variables are updated from the results of a SQL query. The query must return a single row containing exactly as many columns as variables in the UPDATE list. If it does not, the UPDATE task fails and no variables are updated.

For a STATEMENT update, the charexpression is evaluated and then sent as a string literal to the database server for execution.

For a SQL update, paramquery is composed of all the tokens preceding END TASK. It must contain as many question marks as expressions appearing in the SUBSTITUTING list. The expression results are passed to the question marks in the order they appear in the paramquery. Question marks are only allowed in positions supported by the database server, typically in a WHERE clause. Question marks are typically not allowed to substitute table names, column names, or other database objects. To compose a query using a database object whose name is only known at run time, use the STATEMENT syntax rather than the SQL syntax.

## RUN

run ::= RUN { outstatement | script }

script ::= SCRIPT charexpression [ [ON] dbconname ]

**Notes:**

For RUN STATEMENT or RUN SQL, the indicated SQL statement is executed. It must not return a result set. The RUN task is typically used to execute a stored procedure, database INSERT, or database UPDATE.

For RUN SCRIPT, charexpression names a script file containing SQL statements. The statements in the script file are executed.

## CREATE

create ::= CREATE fileidentifier [ headers ]

fileidentifier ::= flatfileidentifier | sheetfileidentifier

flatfileidentifier ::= { CSV | TSV | TXT } charexpression

sheetfileidentifier ::= XLSX charexpression charexpression [ SHEET ]

headers ::= [ WITH ] { headerspecs | noheaderspec }

headerspecs ::= [ HEADERS ] charexpression [, charexpression2 … ]

noheaderspec ::= NO [ HEADERS ]

**Notes:**

The flat file or workbook worksheet specified by fileidentifier is created. If fileidentifier previously existed, its previous contents are discarded. Data can be written by subsequent APPEND statements for the same fileidentifier.

If the headerspecs clause appears, the indicated charexpressions are evaluated and the results are written as headers.

If NO HEADERS appears, no headers are written.

If headers clause is omitted, the default is NO HEADERS.

## APPEND

append ::= APPEND fileidentifier [ FROM ] { outstatement }

**Notes:**

The fileidentifier must have been previously created by a CREATE statement.

The FROM clause is evaluated and each row is written to fileidentifier. If the CREATE statement resulted in headers being written to the file, then the number of columns in the evaluated FROM clause must match the number of headers written. If they do not match, an error occurs.

## WRITE

write ::= WRITE fileidentifier [ writeheaders ] [ FROM ] { outstatement | tableidentifier }

writeheaders ::= [ WITH ] { writeheaderspecs | noheaderspec }

writeheaderspecs ::= [ HEADERS ] [ charexpression [, charexpression2 … ] ]

tableidentifier ::= [ dbconname ] TABLE charexpression

**Notes:**

The flat file or workbook worksheet specified by fileidentifier is created optionally with headers. The FROM clause is evaluated and each row is written to fileidentifier.

If HEADERS appears without a list of charexpressions, then headers are written using the column names from the metadata of the evaluated FROM clause.

If the writeheaders clause is omitted, the default is WITH HEADERS. This behavior is different from when the headers clause omitted from a CREATE statement.

If the FROM clause specifies a tableidentifier, then the FROM clause is effectively the following:

SELECT \* FROM charexpression

charexpression from tableidentifier is evaluated to determine the table name.

## OPEN

open ::= OPEN flatfileidentifier [ openheaders ]

openheaders ::= [ WITH ] { openheaderspecs | noheaderspec } | ignoreheaderspec

openheaderspecs ::= [ HEADERS ] charexpression [, charexpression2 … ]

ignoreheaderspecs ::= IGNORE [ HEADERS ]

**Notes:**

The flat file specified by fileidentifier is opened, optionally reading headers. Data can be read by subsequent LOAD statements for the same fileidentifier.

If the openheaders clause appears, then headers are read from the first record of the file and they must match the evaluated charexpressions. If they do not match, an error occurs.

If IGNORE HEADERS appears, then headers are read but discarded.

If NO HEADERS appears, then the first record of the file is assumed to contain data and is not read by the OPEN statement.

If the openheaders clause is omitted, the default is IGNORE HEADERS.

## LOAD

load ::= LOAD flatfileidentifier [ columns ] [ INTO ] { instatement }

columns ::= COLUMNS columnidentifier [,columnidentifier2 … ]

columnidentifier ::= intepxression | charexpression

instatement ::= [ dbconname ] { statement | tokenstatement }

tokenstatement ::= SQL tokenquery

**Notes:**

The fileidentifier must have been previously opened by an OPEN statement.

Each data row is read from fileidentifier and the INTO clause is applied to it. If the OPEN statement resulted in headers being read from the file, then the number of data columns read by LOAD must match the number of headers read by OPEN. If they do not match, an error occurs.

If the columns clause is specified, each intexpression or charexpression is evaluated. Each intexpression must evaluate to a number between 1 and the number of data columns read from the fileidentifier. Each charexpression must evaluate to a header read from the fileidentifier. Each columnidentifier indicates the position of a column read from fileidentifier. Only the data columns at the indicated positions are retained. Positions can be specified in any order, and the same column position can be retained multiple times.

If the columns clause is omitted, all data columns are retained in the positions they appear within the fileidentifier. That is, the default is “COLUMNS 1, 2, 3, …” for the number of columns that appear in fileidentifier.

The INTO clause must specify an INSERT or UPDATE statement containing question marks. It must contain the same number of question marks as data columns in the fileidentifier; or, if the columns clause is specified, the same number of question marks as intexpressions in the columns clause. If the number of question marks is not correct, an error occurs.

When each data row is read, each column value is substituted for the question mark at that column position; or, if the columns clause is specified, each column value is substituted for the question mark(s) at the position(s) where an intexpression in the columns clause evaluated to the file column position.

For example, if the columns clause specifies “COLUMNS 3, 1, 2, 3”, then column 1 in fileidenitifier is substituted for the second question mark, column 2 is substituted for the third question mark, and column 3 is substituted for the first and fourth question marks.

After all question marks are substituted, the INTO clause is effectively executed, although actual execution of every INTO clause is deferred until the entire file has been read. All are executed in a single batch.

## READ

read ::= READ fileidentifier [ readheaders ] [ columns ]  
[ INTO ] { instatement | tableidentifier }

readheaders ::= [ WITH ] { readheaderspecs | noheaderspec } | ignoreheaderspec

readheaderspecs ::= [ HEADERS ] [ charexpression [, charexpression2 … ] ]

**Notes:**

The flat file specified by fileidentifier is opened, optionally reading headers. Each data row is read from fileidentifier and the INTO clause is applied to it.

If HEADERS appears with a list of charexpressions, then headers are read from the first record of the file and they must match the evaluated charexpressions. If they do not match, an error occurs.

If HEADERS appears without a list of charexpressions and the INTO clause does not specify tableindentifier, then headers are read but discarded. See below for explanation of the behavior when tableidentifier is specified.

If IGNORE HEADERS appears, then headers are read but discarded.

If NO HEADERS appears, then the first record of the file is assumed to contain data.

If the readheaders clause is omitted, the default is HEADERS without a list of charexpressions.

If the INTO clause specifies a tableidentifier, then the INTO clause is effectively the following:

INSERT INTO charexpression [columnlist] VALUES ( valuelist )

columnlist appears only if HEADERS appeared without a list of charexpressions. In this case, columnlist contains a comma-separated list of the file headers in the order they appear in the file, enclosed in parentheses; or, if the columns clause is specified, columnlist contains a comma-separated list of the file headers at the positions specified in the columns clause in the order the positions appear in the columns clause, enclosed in parentheses.

charexpression from tableidentifier is evaluated to determine the table name.

valuelist is replaced by a comma-separated list of question marks, with as many question marks as columns in fileidentifier; or, if the columns clause is specified, as many question marks as intexpressions in the columns clause.

If the INTO clause specifies a tableidentifier, then either fileidentifier must contain headers or the columns clause must be specified.

## ZIP

zip ::= ZIP  
[ FROM ] charexpression [, charexpression2 … ]  
[ TO ] charexpression

**Notes:**

A new zip archive is created. If the archive previously existed, its previous contents are discarded. The files specified in the FROM clause are written to the archive named by the TO clause.

Each FROM charexpression may optionally contain wildcard characters. All the regular files that match an evaluated charexpression are included in the zip archive.

The TO charexpression is evaluated to determine the name of the zip archive.

The archive that is created has no directory structure within it. All files are written at the root of the archive.

## UNZIP

unzip ::= UNZIP  
[ FROM ] charexpression  
[ TO charexpression ]

**Notes:**

The zip archive named by the evaluated FROM charexpression is read and the files contained are written to the directory named by the evaluated TO charexpression.

If the TO clause is omitted, files are written to the default data directory of the process.

## PUT

put ::= PUT  
[ BINARY | ASCII ]  
[ FROM ] charexpression [ , charexpression2, … ]  
[ TO ] [ ftpconname ] charexpression

ftpconname ::= { DEFAULT [ FTP ] } | { [ FTP ] conname }

**Notes:**

The local files named by the list of evaluated charexpressions are transferred via FTP to the remote directory named by the evaluated TO charexpression.

The FTP protocol and connection parameters are determined by ftp properties of the process unless ftpconname is specified.

The BINARY | ASCII option is ignored if it is not supported by the selected protocol.

## GET

get ::= GET  
[ BINARY | ASCII ]  
[ FROM ] [ ftpconname ] charexpression [ , charexpression2, … ]  
[ TO ] charexpression

ftpconname ::= { DEFAULT [ FTP ] } | { [ FTP ] conname }

**Notes:**

The remote files named by the list of evaluated charexpressions are transferred via FTP to the local directory named by the evaluated TO charexpression.

The FTP protocol and connection parameters are determined by ftp properties of the process unless ftpconname is specified.

The BINARY | ASCII option is ignored if it is not supported by the selected protocol.

## EMAIL

email ::= EMAIL  
[ THROUGH emailaconname ]  
FROM charexpression  
TO charexpression [ , charexpression2 … ]  
[ CC charexpression [ , charexpression2 … ] ]  
[ SUBJECT charexpression ]  
[ BODY charexpression ]  
[ { ATTACH | ATTACHMENT } charexpression [ , charexpression2 … ] ]

emailaconname ::= { DEFAULT [ EMAIL ] } | { [ EMAIL ] conname }

**Notes:**

An email message is sent with the indicated parameters.

The email server and connection parameters are determined by the email properties of the process.

## DELETE

TODO: Add documentation here.

## RENAME

TODO: Add documentation here.

## COPY

TODO: Add documentation here.

## MAKE

make ::= MAKE [ DIRECTORY ] charexpression

The directory named by the evaluated charexpression is created. If it already exists or cannot be created, an error occurs.

## LOG

TODO: Add documentation here.

## GO

TODO: Add documentation here.

## FAIL

TODO: Add documentation here.

## PROCESS

process ::= PROCESS  
[ ASYNC | ASYNCHRONOUSLY | SYNC | SYNCHRONOUSLY ]  
charexpression  
[ [ WITH ] expression [ ,expression2 … ] ]

**Notes:**

The process definition named by the evaluated charexpression is invoked as a nested process. The loader for the current process is used to load the nested process.

The expressions are evaluated and passed as arguments to the nested process.

If SYNC or SYNCHRONOUSLY is specified or defaulted, the nested process is executed synchronously. The invoking PROCESS statement does not complete until the nested process completes. If the nested process ends with failure status, the invoking PROCESS statement fails.

If ASYNC or ASYNCHRONOUSLY is specified, the nested process is submitted to execute asynchronously. The invoking PROCESS statement completes immediately upon submitting the nested process for execution. The invoking process cannot directly determine the completion status of the nested process. The invoking process does not complete until all nested asynchronous processes complete.

If a nested asynchronous process is the last task to complete in the invoking process, then the completion status of the invoking process is taken from the completion status of the nested asynchronous process.

## DO

TODO: Add documentation here.

## FOR

for ::= FOR varname [, varname2 …] [ FROM ] { csvidentifier | valuetable | files | outstatement } tasks

csvidentifier ::= CSV fileidentifier [ readheaders ] [ columns ]

valuetable ::= VALUES valueset [, valueset2 … ]

valueset ::= ( expression [, expression2 …] )

files ::= FILES charexpression

**Notes:**

The list of tasks is invoked repeatedly, each time substituting values into the indicated variables as specified by the FROM clause.

If CSV is specified, the tasks are invoked for each data row in the CSV file. The number of columns retrieved must match the number of variable names specified and with compatible types. The variables are set from the retrieved columns in the file row.

If VALUES is specified, the tasks are invoked for each value set in the value table. The number of values in each set must match the number of variable names specified and with compatible types. Each value in a set can be a literal or an expression including variables.

If FILES is specified, only one variable name can be specified and it must be of VARCHAR type. charexpression is evaluated and its value is treated as a file name mask. The file system is queried for files whose names match the mask and the list of tasks is invoked for each matching file name.

If outstatment is specified, the tasks are invoked for each row in the result set. The result set must return the same number of columns as variable names specified and with compatible types.

## ON

on ::= ON { literal | programmed } tasks

literal ::= schedule [ , schedule2 … ]

schedule ::= { onetime | recurring }

onetime ::=  
DATETIME datetime |  
{[DATE] date | TODAY} AT time |  
TODAY NOW

recurring ::=  
{DATE date | TODAY}  
EVERY { {HOUR | MINUTE | SECOND} | int {HOURS | MINUTES | SECONDS}  
[FROM time] [UNTIL time] } |  
  
{EVERY {DAY | int DAYS} | DAILY}  
[FROM date] [UNTIL date]  
{AT time | EVERY { {HOUR | MINUTE | SECOND} | int {HOURS | MINUTES | SECONDS} }  
[FROM time] [UNTIL time] } |  
  
{EVERY {WEEK | int WEEKS | WEEKDAY} [ON] daysofweeklist |  
[EVERY] daysofweeklist | WEEKDAYS | WEEKLY [[ON] dayofweeklist]}  
[FROM date] [UNTIL date]  
{AT time | EVERY { {HOUR | MINUTE | SECOND} | int {HOURS | MINUTES | SECONDS} }  
[FROM time] [UNTIL time] } |  
  
{EVERY {MONTH | int MONTHS} | MONTHLY} [ON] {DAY int | {FIRST | SECOND |THIRD | FOURTH | LAST}  
{dayofweek | DAY | WEEKDAY | WEEKEND DAY}  
[FROM date] [UNTIL date]  
{AT time | EVERY { {HOUR | MINUTE | SECOND} | int {HOURS | MINUTES | SECONDS} }   
[FROM time] [UNTIL time] } |  
  
HOURLY

programmed ::= SCHEDULE stringexp

**Notes:**

The list of tasks is invoked on the indicated schedule or schedules. Multiple schedules can be specified, separated by commas. One-time and recurring schedules can be mixed.

If the literal syntax is specified, all date, time, datetime, int, daysofweeklist, and dayofweek values must be literals. No variables are recognized and no variable substitution occurs. To use a schedule that is determined dynamically at run time based on variable values, use the programmed syntax.

If the programmed syntax is specified via the SCHEDULE keyword, then stringexp may contain expressions including variables. stringexp is evaluated, its syntax must match the literal syntax, and the specified schedule or schedules are used.

The thread for the ON task sleeps between scheduled times.

If the enclosed task set terminates with error status, the ON task terminates immediately with error status regardless of whether the schedule would otherwise continue.

While the enclosed task set is running, if a later scheduled run time occurs, the later run time is skipped. If it is necessary that the enclosed tasks start at every scheduled run time regardless of whether earlier tasks are still running, then the enclosed tasks should be included in a separate process file launched using a PROCESS ASYNC task.

If the list of schedules includes any recurring schedules without an end date, the ON task continues to run indefinitely until the enclosing task set terminates with error status or the process is interrupted.

When the WEEKLY keyword is specified, the day of week list can be omitted and the default of Sunday is used.

When the MONTHLY keyword is specified, the day of the month can be omitted and the default of day 1 is used.

When the DAILY, WEEKLY, or MONTHLY keyword is specified, the time of day schedule can be omitted and the default of one-time at 12:00 AM is used. Otherwise an AT or EVERY clause must appear to specify the time of day schedule.

In the EVERY clause, if “FROM time” is omitted the default of 'FROM 12:00:00 AM' is used.

In the EVERY clause, if “UNTIL time” is omitted the default of 'UNTIL 11:59:59 PM' is used.

The HOURLY keyword is equivalent to EVERY DAY EVERY HOUR.

## WAITFOR

waitfor ::= WAITFOR { delay | time }

delay ::= DELAY charexpression

time ::= TIME charexpression

**Notes:**

When delay is specified, the charexpression must evaluate to a string in the format hh:mm[:ss]. The task sleeps for the indicated amount of time before completing. A maximum of 24 hours may be specified.

When time is specified, the charexpression must evaluate to a string in the format hh:mm[:ss] [AM|PM]. If AM|PM is omitted, 24 hour time format is assumed. The task sleeps until the indicated time of day.

## CONNECT

connect ::= CONNECT connection [, connection2 … ]

connection ::= conname [ conparams ]

conparams ::= [ TO ] { { DEFAULT [ WITH charexpression ] } | charexpression }

**Notes:**

The conname must have been declared in the CONNECTIONS section. The indicated parameters are associated with the connection. The actual connection is lazy; it is not established until a task is executed that requires the connection.

If conparams is omitted or DEFAULT is specified, the default properties for the connection type are used as established by the run-time environment. If WITH charexpression is specified, the expression is evaluated and the properties it specifies override the defaults established by the runtime environment. If DEFAULT is not specified, no defaults from the environment are used and all required properties for the connection type must be included in the evaluated charexpression.

charexpression must be a character string expression that evaluates to a sequence of keyword value pairs separated by blanks, each keyword specifying a property that is supported by the connection type. If a value is enclosed in double quotes, the quotes are not considered to be part of the value.