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 ] [ connections ] 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

**Notes:**

Default connections are specified in properties files RunDbp.jdbc.properties, RunDbp.ftp.properties, and RunDbp.email.properties. Explicitly declared “named connections” are only required when a script needs to connect to non-default servers.

Various tasks documented below support named connections. Before using a named connection, a CONNECT task must be run to specify the properties of the connection.

# Tasks

tasks ::= task [ task2 … ]

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

predecessors ::= [ taskname | PREVIOUS ] [ 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 | close | open | load | read | zip | unzip | put | get | move | email | delete | rename | copy | make | log | go |stop | fail | process | do | for | on | waitfor | connect | request | prompt

**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 or PREVIOUS is specified, 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 Elements

Some terms are commonly used in several different task types.

## Data Source

datasource ::= outstatement | values | properties

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

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

statement ::= STATEMENT charexpression

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

proccall ::= PROCEDURE charexpression [ [ WITH ] arg [, arg2 …] ] [ RETURNING varname ]

arg ::= expression [ IN ] | varname [ IN | OUT | OUTPUT | INOUT ]

tableidentifier ::= TABLE charexpression

values := VALUES valuelist [ , valuelist2 … ]

valuelist := ( expression [ , expression2 …] )

properties := PROPERTIES charexpression propname [ , propname2 … ]

**Notes:**

A data source specifies the source of data consumed by the task, such as a WRITE task.

If conname is specified, it must name a database connection defined in the CONNECTIONS section and initialized with a value in a CONNECT task preceding the task in which conname is used.

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

If SQL is specified, 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 SELECT list or 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.

If PROCEDURE is specified, the charexpression is evaluated and used as the name of a stored procedure to be invoked on the database server. If arguments are specified, the default mode is IN, which means the argument is input to the stored procedure only. An IN argument may be a variable name or an expression which is evaluated and the value is passed to the stored procedure. An OUT or INOUT argument must be a variable name. The variable accepts output from the stored procedure. An INOUT argument serves as both input and output. OUTPUT is an alias for OUT. If RETURNING is specified, the named variable accepts the return value from the stored procedure.

If TABLE is specified, the charexpression is evaluated and expected to be the name of a table on the database server optionally with other syntactical elements. Effectively the following string is sent to the database server for execution:

SELECT \* FROM charexpression

If VALUES is specified, a SQL VALUES clause must be specified. Each value may be a constant or an expression including variables. The values are evaluated locally without accessing the database. They must be of data types supported locally by the DBPA scripting language.

If PROPERTIES is specified, the charexpression is evaluated and used as the name of a file that must exist in the properties directory indicated in RunDbp.path.properties. If a relative path is specified, it is relative to the properties directory. Each propname expression is evaluated and used as the name of a property in the file, whose value is retrieved to be stored to the variable in the respective position in the variable name list. If the property is not found, NULL is stored to the variable.

## Data Target

datatarget ::= [ dbconname ] { statement | tokenstatement | tableidentifier }

tokenstatement ::= SQL tokenquery

**Notes:**

A data target specifies the target of data consumed by the task, such as a READ task.

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

If SQL is specified, tokenquery is composed of all the tokens preceding END TASK. It is evaluated and sent to the server for execution.

The evaluated charexpression or the tokenstatement must specify a SQL INSERT… VALUES… or UPDATE… WHERE… statement containing question marks. It must contain the same number of question marks as columns in the source consumed by the task, typically rows read from a file. If the number of question marks is not correct, an error occurs. The INSERT or UPDATE statement is executed once for each row of the source, substituting the columns of the source into the question marks in the order they appear.

If TABLE is specified, the charexpression is evaluated and expected to be the name of a table on the database server optionally with other syntactical elements. Effectively the following string is sent to the database server for execution:

INSERT INTO charexpression [columnlist] VALUES ( valuelist )

columnlist only appears if the source consumed by the task provides column names. valuelist is a comma-separated list of question marks, containing as many question marks as columns in the source consumed by the task. The INSERT statement is executed once for each row of the source, substituting the columns of the source into the question marks in the order they appear.

## File Identifier

fileidentifier ::= flatfileidentifier | sheetfileidentifier

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

sheetfileidentifier ::= XLSX filecharexpression sheetcharexpression [ SHEET ]

**Notes:**

A file identifier identifies a flat file or a worksheet of a workbook.

The filecharexpression is evaluated to determine the name of the file. It may include no path, a relative path, or an absolute path. If no path is specified, the file is created in the write or read directory indicated in RunDbp.path.properties depending on the task type. If a relative path is specified, it is relative to the write or read directory.

The sheetcharexpression is evaluated to determine the name of the worksheet within a workbook.

The write directory is the default for a file that is written by a task, except when the write is essentially a file transfer to a file that is assumed will be read by a subsequent task. The write directory is also the default for a file that is read by a task if it Is assumed that the file was written by a previous task. The specific tasks that use the write directory by default are: CREATE, APPEND, WRITE, CLOSE, ZIP (both the FROM and TO files), PUT (FROM file), EMAIL (ATTACHMENT files), DELETE, RENAME, COPY (both the FROM and TO files), and MAKE.

The read directory is the default for a file that is read by a task. The read directory is also the default for a file that is written by a task if the write is essentially a file transfer to a file that is assumed will be read by a subsequent task, e.g., extract from an archive or get from a remote server. The specific tasks that use the read directory by default are: OPEN, LOAD, READ, UNZIP (both the FROM and TO files), GET (the TO file), and FOR fileidentifier.

# 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) |  
CHOOSE(intexpression, intexpression [, intexpression2… ])  
CASE WHEN boolexpression THEN intexpression … ELSE intexpression END |  
CASE expression WHEN expression THEN intexpression … ELSE intexpression END |  
DATEPART(datepart, dateexpression) |  
CHARINDEX(charexpression, charexpression [, intexpression ]) |   
LEN(charexpression) |  
intvarname |  
( intexpression )

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

charexpression ::= charterm [ + charterm ]

charterm ::= ‘quoted text’ |  
NULL |  
ISNULL(charexpression, charexpression) |  
IIF(boolexpression, charexpression, charexpression) |  
CHOOSE(intexpression, charexpression [, charexpression2… ])  
CASE WHEN boolexpression THEN charexpression … ELSE charexpression END |  
CASE expression WHEN expression THEN charexpression … ELSE charexpression END |  
ERROR\_MESSAGE() |  
FORMAT(formatable, formatstring) |  
LEFT(charexpression, intexpression) |  
LTRIM(charexpression) |  
LOWER(charexpression) |  
REPLACE(charexpression, charexpression, charexpression) |  
REPLICATE(charexpression, intexpression) |  
RIGHT(charexpression, intexpression) |  
RTRIM(charexpression) |  
SPACE(intexpression) |  
SUBSTRING(charexpression, intexpresion, intexpression) |  
UPPER(charexpression) |  
charvarname |  
( charexpression )

formatable ::= intexpression | dateexpression

formatstring ::= stringexpression

dateexpression::=  
NULL |  
ISNULL(dateexpression, dateexpression) |  
IIF(boolexpression, dateexpression, dateexpression) |  
CHOOSE(intexpression, dateexpression [, dateexpression2… ])  
CASE WHEN boolexpression THEN dateexpression … ELSE dateexpression END |  
CASE expression WHEN expression THEN dateexpression … ELSE dateexpression END |  
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.

The DATEPART function returns an integer for the corresponding datepart literal as follows:

YEAR The year such as 2017

MONTH The month-of-year from 1 (January) to 12 (December)

DAY The day of the month from 1 to 31

WEEKDAY The day-of-week from 1 (Monday) to 7 (Sunday)

HOUR The hour of the day from 0 to 23

MINUTE The minute of the hour from 0 to 59

SECOND The second of the minute from 0 to 59

The ERROR\_MESSAGE functions returns the error message from the closest failed direct or indirect predecessor of the task in which the charexpression appears.

## UPDATE

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

**Notes:**

The indicated variables are updated from the data source. The data source 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.

## RUN

run ::= RUN { datasource | script }

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

**Notes:**

For RUN datasource, the SQL of the data source 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 ]

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.

A TXT file must always be created with a single column.

By default the file is created in the write directory indicated in RunDbp.path.properties.

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 ] datasource

**Notes:**

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

The data source is evaluated and each row is written to fileidentifier. If the CREATE task 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 | HTML varname } [ options ] [ writeheaders ] [ FROM ] datasource

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

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

options ::= option [ option… ]

option ::= flatfileoption | csvoption | xlsxoption | htmloption

flatfileoption ::= CRLF | LF

csvoptoin ::= NOQUOTES

xlsxoption ::= STYLED

htmloption ::= TABLE STYLE charexpression |  
HEAD STYLE charexpression |  
BODY STYLE charexpression |  
HEAD CELL STYLE charexpression |  
BODY CELL STYLE charexpressoin |  
CELL STYLE charexpression

**Notes:**

If fileidentifier is specified, the flat file or workbook worksheet specified by fileidentifier is created optionally with headers. The data source is evaluated and each row is written to fileidentifier.

A TXT file must always be written from a data source with a single column.

By default the file is created in the write directory indicated in RunDbp.path.properties.

If HTML is specified, the data source is evaluated, formatted as an HTML table, and written to the variable with the specified varname. When writing each value in the result set to the variable, it is enclosed by <th >or <td> tags as appropriate, each row is enclosed by <tr> tags, the header is enclosed by <thead> tags, the body is enclosed by <tdata> tags, and the table is enclosed by <table> tags.

If a value read from the data source is prefixed with an opening <th> or <td> tag, an additional <th> or <td> tag is not added. In that case, the tag may contain a style attribute or any other attributes to apply to the individual cell. However, note that a closing </th> or </td> tag is still added. Therefore, a value in the data source must not be suffixed with closing </th> or </td> tag even when it is prefixed with an opening tag.

If the first value in a row read from the data source is prefixed with an opening <tr> tag, an additional <tr> tag is not added. In that case, the tag may contain a style attribute or any other attributes to apply to the row. However, note that a closing </tr> is still added after the last value in the row. Therefore, the last value in a row from the data source must not be suffixed with closing </tr> tags even when the first value in the row is prefixed with an opening tag.

The first cell in a row may contain a <tr> tag followed by a <th> or <td> tag as appropriate to specify styling for the row as well as styling for the first cell in the row.

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.

A flatfileoption can be applied to TXT, CSV, and TSV files. CRLF ends output lines with carriage return + line feed characters. LF ends output file lines with only line feed characters. If neither option is specified, the output file is written with the line terminator character(s) native to the platform on which DBPA is running.

NOQUOTES can only be applied to CSV files. When specified, character string output fields are only enclosed in double quotes if they contain commas or double quotes. When not specified, all character string output fields are enclosed in double quotes regardless of their content.

STYLED can only be applied to XLSX sheets. When specified, individual character-string values from the data source may be prefixed with an HTML tag of following the same rules that apply when writing HTML, as follows:

prefix ::= <tag style=”css-styling}”>

tag ::= tr | th | td

css-styling ::= property-value [; property-value…]

property-value ::= property : value

If tag is tr, the tag must prefix the first value in a row of the result set. The resulting row in the output sheet is styled to emulate the CSS properties specified in the prefix.

If tag is th, the tag must prefix a value in the header row of the result set. The resulting cell in the output sheet is styled to emulate the CSS properties specified in the prefix.

If tag is td, the tag must prefix a value in a body row of the result set. The resulting cell in the output sheet is styled to emulate the CSS properties specified in the prefix.

The first cell in a row may contain a <tr> tag followed by a <th> or <td> tag as appropriate to specify styling for the row as well as styling for the first cell in the row.

Any htmloption can be specified either when writing to XLSX sheet or HTML variable.

If TABLE STYLE is specified, the charexpression is evaluated and should result in a semi-colon separated list of property:value pairs of CSS properties. When writing XLSX, the properties are emulated as though the output data were written to a HTML <table> element. When writing HTML, the properties are written directly to a style attribute of the <table> element.

If HEAD STYLE is specified, the charexpression is evaluated and should result in a semi-colon separated list of property:value pairs of CSS properties. When writing XLSX, the properties are emulated as though the header output data were written to a HTML <thead> element. When writing HTML, the properties are written directly to a style attribute of the <thead> element.

If BODY STYLE is specified, the charexpression is evaluated and should result in a semi-colon separated list of property:value pairs of CSS properties. When writing XLSX, the properties are emulated as though the body output data were written to a HTML <tbody> element. When writing HTML, the properties are written directly to a style attribute of the <tbody> element.

If HEAD CELL STYLE is specified, the charexpression is evaluated and should result in a semi-colon separated list of property:value pairs of CSS properties. When writing XLSX, the properties are emulated as though the header output data were written to a HTML <thead> element. When writing HTML, the properties are written directly to a style attribute of the <thead> element.

STOPPED WRITING HERE!!!

## CLOSE

close ::= CLOSE fileidentifier

**Notes:**

The fileidentifier must have been previously created or written by a CREATE or WRITE task. If so, the file is closed immediately. Otherwise, the task fails.

A file that is created with CREATE or written with APPEND or WRITE normally remains open until:

* The root script in the stack of nested scripts runs to completion; or,
* the file is read by a task such as ZIP, PUT, EMAIL, COPY, etc.

It remains open so that additional output can be appended to a flat file, or additional worksheets can be written to a workbook. As a result, in a long-running script, many written files can end up with the same time stamp even though they may have been written at different times. To assure a file is timestamped when writing is logically completed, use the CLOSE task.

## OPEN

open ::= OPEN fileidentifier [ openheaders ]

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

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

ignoreheaderspecs ::= IGNORE [ HEADERS ]

**Notes:**

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

Only CSV, TXT, and XLSX files can be opened. A TXT file is always read as a single VARCHAR column.

By default the file is read from the readdirectory indicated in RunDbp.path.properties.

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 fileidentifier [ columns ] [ INTO ] datatarget

columns ::= COLUMNS columnidentifier [,columnidentifier2 … ]

columnidentifier ::= intepxression | charexpression

**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 ] datatarget

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

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

**Notes:**

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

Only CSV, TXT, and XLSX files can be read. A TXT file must be read into a data target with a single VARCHAR column.

By default the file is opened from the read directory indicated in RunDbp.path.properties.

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.

By default the FROM file(s) are read from and the TO archive is written to the write directory indicated in RunDbp.path.properties. This reflects the assumption that the FROM files will have been created in the write directory by the current process.

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.

By default the FROM file(s) are read from the read directory indicated in RunDbp.path.properties.

If the TO clause is omitted, files are written to the read directory. If a relative path is specified, it is relative to the read directory. This reflects the assumption that the TO files will be read by the current 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.

By default the FROM file(s) are read from the write directory indicated in RunDbp.path.properties. This reflects the assumption that the FROM files will have been created in the write directory by the current process.

## 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.

By default the TO directory is the read directory indicated in RunDbp.path.properties. If a relative path is specified, it is relative to the read directory. This reflects the assumption that the TO files will be read by the current process.

## MOVE

move ::= MOVE  
[ BINARY | ASCII ]  
[ FROM ] [ ftpconname ] charexpression  
[ TO ] charexpression

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

**Notes:**

The remote file named by the evaluated FROM charexpressions is moved to the evaluated TO charexpression. This may include both renaming the file and moving it to a different remote directory.

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 [ TEXT | HTML ] ]  
[ { 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.

By default the ATTACHMENT file(s) are read from the write directory indicated in RunDbp.path.properties. This reflects the assumption that the ATTACHMENT files will have been created in the write directory by the current process.

## DELETE

delete ::= DELETE charexpression [, charexpression2 … ]

**Notes:**

One or more files are deleted.

Each charexpression may optionally contain wildcard characters. All the regular files that match an evaluated charexpression are deleted.

By default the file(s) are deleted from the write directory indicated in RunDbp.path.properties. This reflects the assumption that the files will have been created in the write directory by the current process.

## RENAME

rename ::= RENAME [ FROM ] charexpression [ TO ] charexpression

**Notes:**

A single FROM file is renamed to the TO name. If the underlying file system supports it, this may involve a move of the file to a different directory if different paths are specified or implied.

By default the file is located in the write directory indicated in RunDbp.path.properties. This reflects the assumption that the file will have been created in the write directory by the current process.

## COPY

copy ::= COPY  
[ FROM ] charexpression [, charexpression2 … ]  
[ TO ] charexpression

**Notes:**

A single FROM file is copied to the TO file or directory; or, multiple FROM file(s) are copied to the TO directory. If a TO file previously existed, it is overwritten.

Each FROM charexpression may optionally contain wildcard characters. All the regular files that match an evaluated charexpression are copied to the TO directory.

By default the FROM file(s) are read from and the TO file(s) are written to the write directory indicated in RunDbp.path.properties. This reflects the assumption that the FROM files will have been created in the write directory by the current process.

## 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.

If a relative directory is specified, it is relative to the write directory indicated in RunDbp.path.properties.

## LOG

log ::= LOG charexpression [, charexpression2 … ]

Each charexpression is evaluated and each value is written to the log as a separate entry. The log entries have “message” log level, which is the highest level and cannot be suppressed from the log.

## GO

go ::= GO [ charexpression ]

If charexpression is present it is evaluated and if non-NULL is written to the log at “message” log level. If charexpression is omitted or evaluates to NULL, a default message is written at “info” log level, which allows it to be suppressed from the log.

## STOP

stop ::= STOP [ charexpression ]

The task itself terminates with success status but all successors of the task and any enclosing tasks are orphaned. If the process has no concurrent tasks or asynchronous child processes, in effect the process is stopped with success status.

If charexpression is present it is evaluated and if non-NULL is written to the log at “message” log level. If charexpression is omitted or evaluates to NULL, a default message is written at “info” log level, which allows it to be suppressed from the log.

## FAIL

fail ::= FAIL [ charexpression ]

The task terminates with failure status. If charexpression is present it is evaluated and if non-NULL is used as the error message written to the log at “error” log level. If charexpression is omitted or evaluates to NULL, a default message is used as the error message.

## 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

Do ::= DO [ WHILE boolexpression ] tasks

**Notes:**

The list of tasks is executed one or more times.

If WHILE is specified, if boolexpression evaluates to true the list of tasks executes. When it completes, boolexpression is re-evaluated. As long as it remains true, the list of tasks executes repeatedly.

If WHILE is not specified, the list of tasks executes once.

If the list of tasks terminates with failure status, then the enclosing DO task also terminates with failure status and no further iterations are executed.

## FOR

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

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

valuetable ::= VALUES valueset [, valueset2 … ]

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

files ::= FILES charexpression

**Notes:**

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

If fileidentifier is specified, the tasks are invoked for each data row in the flat file or workbook worksheet. Only CSV, TXT, and XLSX files can be read. 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. By default the file is read from the read directory indicated in RunDbp.path.properties

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. By default files are queried in the read directory indicated in RunDbp.path.properties

If daatsource 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.

If the list of tasks terminates with failure status, then the enclosing FOR task also terminates with failure status and no further iterations are executed.

## 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

time ::= quotedtime | NOW | int {HOURS | MINUTES | SECONDS} } FROM NOW

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 | ASYNC }

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.

When ASYNC is specified, the task sleeps until the completion of all asynchronous processes launched by PROCESS ASYNC tasks that are direct or indirect predecessors of this task or descendants of direct or indirect predecessors of this task. Indirect predecessors are predecessors of direct predecessors of this task and direct or indirect predecessors of enclosing tasks of this task.

## 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.

## REQUEST

request ::=  
REQUEST url-template   
[ HEADER header-name-1 [ NULL ] header-value-1 [, … ] ]  
{ get-or-delete-request | put-or-post-request }  
[ { FROM NOTHING |  
FROM data-source [ AS source-column-alias-1 [, source-column-alias-2 … ] ] }  
[ JOIN data-source-1 [ AS join-column-alias-1 [, join-column-alias-2 … ] ] … ] ]  
[ RESPONSE response-template   
{ INTO NOTHING |  
KEEP target-identifier-1 [, target-identifier-2… ] INTO data-target }  
[ KEEP join-identifier-1 [, join-identifier-2… ] JOIN data-target-1 …] }

get-or-delete-request ::=  
GET | DELETE

put-or-post-request ::=  
{ PUT | POST } request-template

For each data-source row, an HTTP request is sent and the response is received into a data-target row, both using JSON encoding for any entities.

url-template is a string expression. It is evaluated and then placeholders are found. A placeholder is a valid identifier enclosed in braces “{identifier}”. The HTTP request is sent to the evaluated URL with placeholders substituted with the values of like-named columns in the FROM data-source.

If the evaluated url-template contains a colon character (e.g., as in “http:”), URL encoding is performed on column values before they are substituted into the url. If the evaluated url-template does not contain a colon, URL encoding is not performed.

Each header-name-n and header-value-n is a string expression. They are evaluated and each pair is added to the request, except that if any header-name-n evaluates to null and the NULL keyword is specified, the name-value pair is silently ignored. If any header-name-n evaluates to NULL where the NULL keyword is not specified, an error occurs. Any header-value-n may be null without error.

request-template is a string expression. It is evaluated and must conform to a modified JSON syntax (except as described below). Where a value is allowed in JSON, an unquoted identifier is allowed in request-template. When the actual JSON request is composed, each unquoted identifier in request-template is replaced by the value of the like-named column from the FROM data-source or possibly a JOIN data-source.

An array in request-template may optionally be of the form [ value-or-structure, … ] where the punctuation characters “[“, “,” “…” and “]” are literals. This is called a dynamic array. When the actual JSON request is composed, this construct is replaced by an array with a varying number of elements as described below.

An object in request-template may optionally be of the form { unquoted-identifier : value-or-structure, … } where the punctuation characters “{“, “,” “…” and “}” are literals. This is called a dynamic object. When the actual JSON request is composed, this construct is replaced by an object with a varying number of fields as described below.

Dynamic arrays and dynamic objects are types of dynamic structures. For each dynamic structure in the evaluated request-template, there must be exactly one JOIN data-source clause. All identifiers referenced within a dynamic structure must appear in the single corresponding JOIN data-source clause and only in that single clause.

Each JOIN data-source must include at least one column name that also appears in the FROM data-source. The join between the FROM data-source and each JOIN data-source is a left outer join between like-named columns. *The FROM data-source and each JOIN data-source must be sorted in the same order on the joined columns. If not, the result is undefined.*

In the request composed for each FROM data-source row, each dynamic structure is composed from the joined rows of the corresponding JOIN data-source.

If FROM NOTHING is specified, the JOIN data-source clauses may still appear. In this case, a single request is sent and each dynamic structure is composed from all the rows returned by the corresponding JOIN data-source.

If the FROM clause is omitted entirely, a single request is sent with no replacement. In that case, no unquoted identifiers may appear in request-template.

When identifier replacement is done on the request, all replacement string values are encoded with JSON escape sequences. No encoding is done on literal quoted strings within the template.

The evaluated request-template may optionally consist of a string enclosed in double quotes. In this case, the request is not sent as JSON but as plain text. If a FROM clause is present, the string within the double quotes must be a single identifier, which is replaced with the value of the like-named column in the FROM data-source. If a FROM clause is not present, there are no restrictions on the string within the double quotes; it is sent in the request entity exactly as specified.

response-template is a string expression. It is evaluated and must conform to the same modified JSON syntax (except as described below). Where a value is allowed in JSON, an unquoted identifier is allowed in request-template. When the actual JSON response is parsed, a value is retrieved wherever an unquoted identifier appears in response-template and is stored to the INTO data-target in the column corresponding to the like-named KEEP target-identifier, or possibly to a JOIN data-target in the column corresponding to the like-named KEEP join-identifier.

Dynamic structures may appear in response-template. For each dynamic structure in the evaluated response-template, there must be exactly one JOIN data-target clause. All identifiers referenced within a dynamic structure must appear in the single corresponding JOIN data-target clause and only in that single clause.

Each KEEP join-identifier list must include at least one identifier that also appears in the KEEP target-identifier list. But if INTO NOTHING is specified, the JOIN data-target clauses may still appear.

For each array element or object field parsed from a dynamic structure in the response, a row is written to the corresponding JOIN data-target.

response-template does not need to include every field known to exist in the actual response. Fields in the actual response that do not appear in response-template are ignored.

The evaluated response-template may optionally consist of an identifier enclosed in double quotes. In this case, the response is not interpreted as JSON but as plain text. He identifier must match a KEEP target-identifier.

If the RESPONSE clause is omitted entirely, the response is not parsed and no data from it is retained.

In addition to unquoted identifiers that appears in response-template, each KEEP list may include the following: column names from the FROM data-source; the special name “status”, which is replaced by the numeric HTTP status code from the response; or, the special name “message”, which is replaced by the HTTP error message string from a failed request or null if the request succeeded. A successful request has a status value between 200 and 299.

A request is sent for every row in FROM data-source even if any request fails. A single REQUEST task may result in any combination of successful and failed requests, each of which may write unique values to the corresponding “status” and “message” columns. After all requests have been processed, if any request failed the REQUEST task terminates with failed status.

If any data-source or data-target is introduced by the keyword SQL, the SQL query must be terminated by the phrase END SQL. The only exception is the final data-target.

## PROMPT

prompt ::= PROMPT prompt-expression [ PASSWORD ] [ INTO ] input-variable1 [ , input-variable2 ... ]

**Notes:**

The prompt-expression is evaluated and used as a console prompt for input from the user. If no console is available, the task fails.

If PASSWORD is specified, user input is not echoed to the console.

If a single input-variable is specified, the entire user input is saved to the variable. If multiple input-variables are specified, the input is parsed and each parsed token is saved to a variable. Tokens are delimited by blanks. To specify a token with embedded blanks, enclose it in double quotes. The double quotes are not part of the token. To include a double quote in a token, include two consecutive double quotes inside a quoted string.

If a variable is INTEGER or DATETIME type, if the input does not match the type, the task fails.