Amos II User's Manual

Amos II Beta Release 5

Staffan Flodin, Vanja Josifovski, Timour Katchaounov, Tore Risch, Martin Sköld, and Magnus Werner
June 23, 2000
Latest revision April 25, 2003

Abstract

This working document describes details of how to use the AMOS II system. In particular it describes the syntax and semantics of the AMOSQL language and the various system functions available. You are recommended to read the document *AMOS II Concepts* first which describes the principles of the AMOS II system and gives an overview of the AMOSQL concepts. This manual is continously updated as AMOS II evolves; the subtitle refers to the current system release.

Acknowledgements

The following persons have contributed to the development of the AMOS II project:
Silvio Brandani, Kristoffer Cassel, Daniel Elin, Marcus Eriksson, Gilles Fabre, Gustav Fahl, Staffan Flodin, Jörn Gebhardt, Björn Hellander, Vanja Josifovski, Jonas Karlsson, Timour Katchaounov, Salah-Eddine Machani, Joakim Näs, Kjell Orsborn, Thomas Padron-McCarthy, Tore Risch, Andreas Sjöstedt, Martin Sköld, Rickard Svensson, and Magnus Werner.

1 Running AMOS II

It is recommended that each user creates a private directory for AMOS II, `<privdir>`. You should then do `cd <privdir>` and copy the following files to `<privdir>`:

```
amos2.exe
amos2.dmp
amos.dll
```

AMOS II is then ready to run in `<privdir>` by the command:

```
amos2 [<db>]
```

where `[<db>]` is an optional name of an AMOS II database image (default is `amos2.dmp`).

The system enters an AMOS II top loop where it reads AMOSQL statements, executes them, and prints their results. You need not connect to any particular database, but instead, if `<db>` is omitted, the system enters an empty database, where only the system objects are defined.

When the AMOS II database is defined and populated, it can be saved on disk with the AMOSQL statement:

```
save "filename";
```

NOTICE: You cannot save using the name '`amos2.dmp`' since that would overwrite the system database.

In a later session you can connect to the saved database by starting AMOS II with:

```
amos2 filename
```

The prompter in the AMOS II top loop is:

```
Amos n>
```

where `n` is a *generation number*. The generation number is increased every time an AMOSQL database update statement is executed. For example:

```
AMOS 1> create type person;
AMOS 2> create type student under person;
```

Database changes can be undone by using the `rollback` statement with a generation number as argument. For example, the statement:

```
AMOS 3> rollback 2;
```

will restore the database to the state it had at generation number 2. It thus undoes the effect of the statement:

```
create type student under person;
```

After the rollback above, the type `student` is removed from the database, but not type `person`.

The statement `commit` makes changes non-undoable, i.e all updates so far cannot be rolled back any more and the generation numbering starts over from 1.

For example:

```
AMOS 2> commit;
AMOS 1> ...
```

To shut down AMOS II orderly first save the database and then type:

```
AMOS 1> quit;
```

JavaAMOS is a version of the AMOS II kernel connected to the Java virtual machine. With JavaAMOS AMOS

II foreign functions can be written in Java (the *callout* interface) and Java functions can call AMOS II functions and send AMOSQL statements to AMOS II for evaluation (the *callin* interface). To start JavaAMOS use the script

```
JavaAMOS
```

instead of `amos2`. It will enter a top loop reading and evaluating AMOSQL statements as `amos2`.

The multi-database browser GOOVI [CR01] is a graphical browser for AMOS II written as a Java application. You can start the GOOVI browser from the JavaAMOS top loop by calling the AMOS II foreign function

```
goovi();
```

It will start the browser in a separate thread.

2 AMOSQL

This section describes the syntax of AMOSQL and explains some semantic details not described elsewhere. For the syntax we use BNF notation with the following special constructs:

A ::= B C: A consists of B followed by C.
A ::= B | C, alternatively (B | C): A consists of B or C.
A ::= [B]: A consists of B or nothing.
A ::= B-list: A consists of one or more Bs.
A ::= B-commalist: A consists of one or more Bs separated by commas.
'xxx': The string (keyword) xxx.

AMOSQL statements are always terminated by a semicolon (;).

Identifiers

Identifiers have the syntax:

```
identifier ::=
        ('_' | letter) [identifier-character-list]
identifier-character ::=
        alphanumeric | '_'
```

AMOS II keywords are case sensitive; they are always written with lower case letters. AMOS II identifiers are NOT case sensitive; i.e. they are always internally capitalized.

Variables

Variables are of two kinds:

- *Local variables* are identifiers for data values in AMOSQL queries and functions. *Local variables* must be declared in function signatures by from clauses (``Query Statements''), or by the declare construct (``Database procedures'').

Syntax:
```
variable-name ::= identifier
```

- *Interface variables* hold temporary query results during a session. Interface variables cannot be referenced in function bodies and are not consirered as being parts of the database.

Syntax:
```
interface-variable-name ::= ':' identifier
gen-variable-name ::= variable-name | interface-variable-name
```

Constants

Constants can be integers, reals, strings, or booleans.
Syntax:

```
constant ::=
        integer-constant | real-constant | boolean-constant |
        string-constant | 'nil'
integer-constant ::=
        ['-'] digit-list
real-constant ::=
        ['-'] digit-list '.' [digit-list]
boolean-constant ::=
        'true' | 'false'
string-constant ::=
        string-separator character-list string-separator
string-separator ::=
        ''' | '"'
```

The surrounding string separators in string constants must be the same.

Comments

The comment statement can be placed anywhere outside identifiers and constants.
Syntax:

```
comment ::=
        '/*' character-list '*/'
```

Statements

The following statements can be entered to the AMOS II top loop. Their details are described hereafter.

```
        create-type-stmt |
        delete-type-stmt |
        create-object-stmt |
        delete-object-stmt |
        add-type-stmt |
        remove-type-stmt |
        create-function-stmt |
```

```
        delete-function-stmt |
        update-stmt |
        query-stmt |
        open-cursor-stmt |
        fetch-cursor-stmt |
        close-cursor-stmt |
        block |
        for-each-stmt |
        set-variable-stmt |
        if-stmt |
        result-stmt |
        quit-stmt |
        set-interface-variable-stmt
query-stmt ::=
        select-stmt | function-call
set-interface-variable-stmt ::=
        'set' interface-variable-name '=' function-call
```

2.1 Creating regular types

The `create type` statement creates a new user type.

Syntax:

```
create-type-stmt ::=
        'create type' type-spec ['under' type-name-commalist]
                ['properties' '(' attr-function-commalist ')']
type-spec ::= identifier | 'bag of' identifier
attr-function ::=
        function-name type-spec ['key']
```

Type names must be unique across all types.

The new type will be an subtype of all the supertypes in the `subtype of` clause. If no supertypes are specified the new type becomes a subtype of the system type `userobject`.

The `attr-function-commalist` clause is optional, and provides a way to define attributes for the new type. The attributes are functions having a single argument and a single result and are initially stored functions (but can be later redefined as other kinds of functions). The argument type of an attribute function is the type being created and the result type is specified by the `type-spec`. The result type must be previously defined.

If 'key' is specified for a property, it indicates that each value of the attribute is unique.

2.1.1 Deleting types

The `delete type` statement deletes a type and all its subtypes.

Syntax:

```
delete-type-stmt ::=
        'delete type' type-name
```

Functions using the deleted type will be deleted as well.

2.2 Creating objects

The `create object` statement creates one or more objects and makes the new object(s) instance(s) of a given user type and all its supertypes.

Syntax:

```
create-object-stmt ::=
        'create' type-name
        ['(' identifier-commalist ')'] 'instances' initializer-commalist
initializer ::=
        gen-variable-name |
        [gen-variable-name] '(' simple-init-value-commalist ')'
simple-init-value ::=
        single-value | collection-value
single-value ::=
        gen-variable-name | constant
collection-value ::=
        bag-value | vector-value
bag-value ::=
        'bag(' single-value-commalist ')'
vector-value ::=
        'vector(' single-value-commalist ')' |
        '{' single-value-commalist '}'
```

Example:

```
   create person (name,age) instances
          :adam ('Adam',26),:eve ('Eve',32);
   create person instances :olof;
   create person (parents) instances
          :tore (bag(:adam,:eve));
```

The new objects are assigned initial values for the specified attributes. The attributes can be any updatable AMOSQL functions of a single argument and value.

One object will be created for each initializer. Each initializer can have an optional variable name which will be bound to the new object. The variable name can subsequently be used as a reference to the object.

The initializer also contains a comma-separated list of initial values for the specified functions. Initial values are specified as constants or variables.

The types of the initial values must match the declared result types of the corresponding functions.

Bag valued functions are initialized using the keywords `bag of` (syntax `bag-value`).

Vector result functions are normally initialized with a comma-separated list of values enclosed in curly brackets (syntax `vector-value`).

It is possible to specify NIL for a value when no initialization is desired for the corresponding function.

2.2.1 Deleting objects

Objects are deleted from the database with the `delete` statement.

Syntax:

```
delete-object-stmt ::=
        'delete' gen-variable-name
```

Referential integrity is maintained by the system which will automatically delete references to the deleted object. It is thus also removed from all stored functions where it is referenced.

Deleted objects are printed as

```
#[OID nnn *DELETED*]
```

The objects may be undeleted by `rollback`. The garbage collector physically removes the OIDs from the database only when their creation has been rolled back or their deletion committed, and they are not references from some variable or external system.

2.2.3 Updating type memberships

The `add-type-stmt` changes the type of one or more objects to the specified type.

Syntax:

```
add-type-stmt ::=
        'add type' type-name ['(' [function-name-commalist] ')']
        'to' new-instances
```

The updated objects may be assigned initial values for all the specified property functions in the same manner as in the create object statement.

The `remove-type-stmt` makes one or more objects no longer belong to the specified type.

Syntax:

```
remove-type-stmt ::=
        'remove type' type-name 'from' variable-name-commalist
```

Referential integrity is maintained so that all references to the objects as instances of the specified type cease to exist.

An object will always be an instance of some type. If all user defined types have been removed, the object will still be member of `userobject`.

2.3 Creating functions

The `create function` statement creates a new user function.

Syntax:

```
create-function-stmt ::=
      'create function' function-name argl-spec '->' resl-spec [fn-implementation]
function-name ::=specific-function-name |
       type-name-list '.' specific-function-name '->' type-name-list
type-name-list ::= type-name | type-name '.' type-name-list
specific-function-name ::= identifier
argl-spec ::='(' [arg-spec-commalist] ')'
arg-spec ::=simple-arg-spec | 'bag of' simple-arg-spec
simple-arg-spec ::=     type-name [variable-name] ('key' | 'nonkey')
resl-spec ::=   arg-spec | multiple-result-spec
multiple-result-spec ::=        ['<'] simple-arg-spec-commalist ['>']
fn-implementation ::=   'as' (derived-body | procedure-body |
                        foreign-body | 'stored')
derived-body ::= simple-select-stmt
foreign-body ::= 'foreign' [string-constant]
```

The `argl-spec` and the `resl-spec` specify the signature of the function.

Semantics:

- The types used in the declarations must be previously defined. The name of an argument or result parameter can be left unspecified if it is never referenced in the function implementation. The names of the argument and result parameters for a given function definition must be unique.
- `bag of` specifications on a single result parameter declares it to be a bag. Derived functions always have implicit `bag of` results.
- Derived functions can also have arguments declared `bag of` making them aggregation operators. NOTICE: Stored functions cannot be aggregation operators. `bag of` declarations for results of derived functions are ignored.
- AMOSQL functions may also have multiple results, indicating that a logical tuple of values is returned. This is indicated by bracketing the result declarations (see syntax for `multiple-result-spec` and example below).

Derived AMOSQL functions are defined by a single query (select statement).

2.3.2 Specifying cardinality constraints

A *cardinality constraint* is a system maintained restriction on the number of allowed occurrences in the database of an argument or result of a function. For example, a cardinality constraint could be that there can be at most one salary and one name per person, while a person may have any number of parents. The only cardinality constraint which is currently supported in AMOSQL is to make a specified argument or result of a stored function unique, by specifying it as a *key*. If you regard the extent of a function as a relation of tuples, the `key` annotation for an argument or result specifies that the argument/result is a key to the extent relation. For example:

```
create function name(person key) -> charstring key as stored;
```

In this case the system guarantees that there is a one to one relationship between OIDs of type `person` and their names.

If the 'key' cardinality constraint is violated by a database update the following error message is printed:

```
Update would violate upper object participation (updating function ...)
```

The keyword 'nonkey' specifies that the parameter has no cardinality constraint.

The default cardinality constraint for a stored function is `key` for the first argument of a stored function and `nonkey` for all other arguments and the result(s). This implies that stored functions are by default single valued.

Since the first argument of a stored function by default is declared as `key` the function `name` could thus also have been written:

```
create function name(person) -> charstring key as stored;
```

For foreign functions it is up to the implementor to guarantee that specified cardinality constraints hold. Cardinality constraint declarations are ignored for derived functions.

Another example:

```
create function married(person husband,person wife key)->boolean as stored;
```

Polygamous marriages are refused to be stored by the function `married`, since the first argument has the default cardinality constraint `key`.

The `bag of` declaration on the result of a stored function actually just overrides the default `key` declaration of its 1st argument with `nonkey`. Thus the function `parents` above could also have been written:

```
create function parents(person nonkey) -> person as stored;
```

2.3.3 Deleting functions

Functions are deleted with the `delete function` statement.

Syntax:

```
delete-function-stmt ::=
        'delete function' function-name
```

For example:

```
 delete function married;
```

Deleting a type also deletes all subtypes and all functions using the deleted types.

2.3.4 Overloaded functions and late binding

Function names may be overloaded, i.e., functions having the same name may be defined differently on different

argument types. This allows generic functions to apply to several different object types. Each specific implementation of an overloaded function is called a *resolvent*.

For example, assume the two following AMOS II function definitions:

```
create function less(number i, number j)->boolean
        as select true where i < j;
create function less(charstring s,charstring t)->boolean
        as select true where s < t;
```

Its resolvents will have the signatures:

```
  less(number,number) -> boolean
  less(charstring,charstring) -> boolean
```

Internally the system stores the resolvents under different function names. The name of a resolvent is obtained by concatenating the type of its arguments with the name of the overloaded function followed by the symbol '->' and the type of the result.(syntax in ``Query Statements''). The two resolvents above will be given the names number.`number.less->boolean` and `charstring.charstring.less->boolean`.

Overloaded function resolvents are allowed to differ on their argument types and the result types. The query compiler resolves the correct resolvent to apply based on the types of the arguments; the type of the result is not considered. If there is an ambiguity, i.e. several resolvents qualify, or if no resolvent qualify an error will be generated by the query compiler.

When overloaded function names are encountered in AMOSQL function bodies, the system will try to use local variable declarations to choose the correct resolvent (early binding).
For example:

```
 create function younger(person p,person q)->boolean
        as select less(age(p),age(q));
```

will choose the resolvent `number.number.less->boolean`, since `age` returns integers and the resolvent `number.number.less->boolean` is applicable to integers by inheritance. The other function resolvent `charstring.charstring.less->boolean` does not qualify since it is not legal to apply to arguments of type integer.

On the other hand, this function:

```
create function nameordered(person p,person q)->boolean
        as select less(name(p),name(q));
```

will choose the resolvent `charstring.charstring.less->boolean`. In both cases the resolution will be done at compile time.

Dynamic type resolution is also done for top level function call to choose the correct resolvent. For example,

```
less(1,2);
```

will choose `number.number.less->boolean`

To avoid the overhead of dynamic type resolution one may use the 'dot notation':

```
number.number.less->boolean(1,2);
```

AMOS II also supports *late binding* of overloaded functions where the overload resolution is done at run time instead of at compile time. For example, suppose that managers are employees whose incomes are the sum of the income as a regular employee plus some manager bonus:

```
 create type employee under person;
 create type manager under employee;
 create function mgrbonus(manager)->integer as stored;
 create function income(employee)->integer as stored;
 create function income(manager m)->integer i
        as select employee.income->integer(m) + mgrbonus(m);
```

Now, suppose that we need a function that returns the gross incomes of all persons in the database, i.e. we use `manager.income->integer` for managers and `employee.income->integer` for non-manager. In AMOS II such a function is defined as:

```
 create function grossincomes()->integer i
        as select income(p)
        from employee p; /*
income(p) late bound */
```

Since `income` is overloaded with resolvents `employee.income->integer` and `manager.income->integer` and both qualify to apply to employees, the resolution of `income(p)` will be done at run time. If only incomes of employees are sought the desired resolvent has to be explicitly specified as `employee.income->integer`.

Since the detection of the necessity of dynamic resolution is done at compile time, overloading a function name may lead to a cascading recompilation of functions defined in terms of that function name. This may take some time. For a more detailed presentation of the management of late bound functions see [FR95].

2.3.5 Function updates

Information in AMOSQL can be thought of as mappings from function arguments to results. These mappings are either defined at object creation time (``Creating Objects''), or altered by one of the function update statements 'set', 'add', or 'remove'.

Syntax:

```
update-stmt ::=
        update-op update-item [for-each-clause] [where-clause]
update-op ::=
        'set' | 'add' | 'remove'
update-item ::=
        function-name '(' single-value-commalist ')' '=' res-values
```

Not every function is updatable. AMOS II defines a function f to be updatable if it is a stored function, or if it is derived from a single updatable function without a join.

Semantics:

- `set` sets the value of an updatable function given the arguments.

  A boolean function can be set to either 'true' or'false'. Setting the value of a boolean function to 'false' means that the truth value is removed from the function.

- `add` adds the specified tuple(s) to the result of an updatable bag result function, analogous to `set`.
- `remove` removes the specified tuple(s) from the result of an updatable bag result function, analogous to `set`.

The update statements are not allowed to violate the cardinality constraints ('key') (See ``Query Statements''.) specified by the `create-type-stmt` or the `create-function-stmt`.

2.4 Query statements

Queries retrieve objects having specified properties. They are specified using the 'query' statement denoting either function calls or select statements.

Syntax:

```
query-stmt ::=
        select-stmt | function-call
```

2.4.1 Function calls

Syntax:

```
function-call ::=
        function-name '(' [parameter-value-commalist] ')'
parameter-value ::=
        function-call | single-value | '(' simple-select-stmt ')'
simple-select-stmt ::=
        'select' expr-commalist
        [for-each-clause] [where-clause]
expr   ::=
        function-call | single-value
```

The built-in functions `+,-,*,/` have equivalent infix syntax with the usual priorities.
For example:

```
 (income(:eve) + income(:ulla)) * 0.5;
```

is equivalent to:

```
 times(plus(income(:eve),income(:ulla)),0.5);
```

2.4.2 The select statement

The *select statement* provides the most flexible way to specify queries.

Syntax:

```
select-stmt ::=
        'select' ['distinct'] expr-commalist
                [into-clause]
                [from-clause]
                [where-clause]
into-clause ::=
        'into' gen-variable-name-commalist
from-clause ::=
        'from' variable-declaration-commalist
variable-declaration ::=
        type-name variable-name |
        'bag of' type-name variable-name
where-clause ::=
       'where' predicate-expression
```

The `expr-commalist` defines the object(s) to be retrieved. See ``Predicate expressions'' for definition of `function-call`.

The `from-clause` declares types of local variables used in the query.

The `where-clause` gives a selection criteria for the search. The details of the where clause is described below in ``Predicate expressions'' .

The result of a select statement is a bag of single result tuples. Duplicates are removed by using the keyword 'distinct', in which case a set is returned.

An `into-clause` is available for specifying variables to be bound to the result. In case more than one result tuple is returned, the variables will be bound only to the elements of the first encountered tuple.
For example:

```
select p into :eve2 from person p where name(p) = 'Eve';
name(:eve2);
```

This query retrieves into the environment variable :eve2 the person whose name is `'Eve'`.

.4.3 Predicate expressions

The general syntax of predicate expressions is:

```
predicate-expression ::=
        predicate-expression 'and' predicate-expression |
        predicate-expression 'or' predicate-expression |
        '(' predicate-expression ')' |
        simple-predicate
simple-predicate ::=
        function-call |
        simple-relterm relop simple-relterm |
        relterm '=' relterm
simple-relterm ::=
        function-call | single-value
relterm ::=
```

```
        simple-relterm | multiple-value | select-statement
multiple-value ::=
        '<' single-value-commalist '>'
relop ::=
        < | > | <= | >= | !=
```

In a function call, the types of the actual parameters and results must be the same as, or subtypes of, the types of the corresponding formal parameters or results.

Resolution of overloaded functions is described in``Overloaded Functions and Late Binding" .

Query variables can be bound to bags on which aggregation operators can be applied.

The comparison operators (=, !=, <, <=, and >=) are treated > as binary boolean functions. They are all defined for any object > type.

1.4.4 Subqueries and aggregation operators

Aggregation operators are defined as functions where one or several arguments are declared as bags:

```
bag of type x
```

The following system aggregation operators are defined:

```
sum(bag of number x) -> number
count(bag of object x) -> integer
maxagg(bag of object x) -> object
minagg(bag of object x) -> object
some(bag of object x) -> boolean
notany(bag of object x) -> boolean
```

Notice that `sum` must be applied only to 'uniform' bags of numbers.

Subqueries always return bags as their results; thus the result of a subquery must be passed to only aggregation operators.

Local variables in queries may be declared as bags. For example `totalincomes` could also have been written:

```
create function totalincomes()->integer
       as select sum(b) from bag of Integer b
       where b = (select i from Person p,Integer i
                       where income(p)=i);
```

2.5 Cursors

For queries and function calls returning bag valued results, the `open-cursor-stmt` and the `fetch-cursor-stmt`, statements are available to iterate over the result.

Syntax:

```
open-cursor-stmt ::=
        'open' cursor-name 'for' select-stmt
cursor-name ::=
        gen-variable-name
fetch-cursor-stmt ::=
        'fetch' cursor-name (into-clause | next-clause)
next-clause ::=
        'next' integer-constant
close-cursor-stmt ::=
        'close' cursor-name
```

For example:

```
create person (name,age) instances :Viola ('Viola',38);
open :c1  for select p from person p where name(p) = 'Viola';
fetch :c1 into :Viola1;
close :c1;
name(:Viola1);
--> "Viola";
```

A cursor is created by the `open-cursor-stmt` and is represented by a bag of result tuples containing objects with unknown types.

The result of the query is always materialized and stored in the cursor bag.

The `fetch-cursor-stmt` fetches the first result tuple from the cursor; i.e. the tuple is removed from the front of the cursor bag. NIL is returned if there are no more result tuples left in the cursor.

If present in a `fetch-cursor-stmt`, the `into` clause will bind elements of the first result tuple to AMOSQL interface variables. There must be one interface variable for each element in the result tuple.

If present in a `fetch-cursor-stmt`, the `next-clause` will display the specified number of result tuples and remove them from the cursor bag.

If neither a next nor an into clause is present in a `fetch-cursor-stmt`, a single result tuple is fetched and displayed.

The `close-cursor-stmt` empties the cursor.

It is sometimes useful to count the number of result tuples in a cursor bag:

```
create function ageofpersonnamed(charstring nm)-> integer a
        as select age(p) from person p where name(p)=nm;
open :c1 for select ageofpersonnamed('Eve');
count(:c1);
1
```

3 Multidatabase system functions

The following AMOSQL system functions and procedures are available for inter-mediator communication:

```
amos_servers()->bag of charstring
```
Returns the names of the mediator servers registered in the nameserver.

```
listen()
```
Starts the mediator server listening loop. The loop can be interrupted with CTRL-C and resumed again by calling `listen()`.

```
nameserver(charstring name)->charstring
```
Makes the current stand-alone database into a nameserver and registers there itself as a mediator server with the given `name`.

```
register(charstring name)->charstring
```
Registers the current stand-alone database as a client mediator with the given `name` in the nameserver running on the local host. The system will complain if the name is already registered in the nameserver.

```
register(charstring name, charstring host)->charstring
```
Registers the current database as a client mediator in the nameserver running on the given `host`.

```
ship(charstring name, charstring cmd)-> object
```
Ships the AMOSQL command `cmd` for execution in the mediator server `name`. The result is shipped back to the caller.

```
this_name()->charstring name
```
Returns the `name` of the mediator where the call is issued. Returns `"NIL"` if issued in a stand-alone database.

3.1 Data integration primitives

More to come here about derived and IUTs.

3.2 ODBC Wrapper

The basic AMOS II system contains a wrapper for ODBC relational data sources. It allows transparent access to data in relational databases through OO views.

ODBC data sources in AMOS II are represented by the type `odbc_ds`, subtype of `relational`.

ODBC data access functions

The following functions are used to connect to and access data in an ODBC data source:

Constructor for odbc_ds objects
```
odbc_ds(charstring logname) -> odbc_ds ods
```

Connect to an ODBC datasource using a default user and password
```
connect(odbc_ds ds1, charstring dsn) -> odbc_ds ds2
```

Connect to an ODBC datasource using a DSN, username, password
```
connect(
```

```
    odbc_ds ds1,
    charstring dsn,
    charstring usr,
    charstring pass) -> odbc_ds ds2
```

Connect to an ODBC datasource using a DBMS dependent connect string

```
connect_native(
     odbc_ds ds1,
     charstring conn_string) -> odbc_ds ds2
```

Disconnect from a datasource. This is also done automatically at system exit.

```
disconnect(odbc_ds ds) -> boolean
```

Map a relational table to a type. The mapping is done as follows:
- a table corresponds to an AMOS type
- each column of the table corresponds to an AMOS function (attribute) of this type

```
import_table(relational ds, charstring table_name) -> type mapped_type
```

Execute an SQL statement, given

- the name of an ODBC data source
- an SQL statement as a string
- a vector of parameters to the query (successive '?' in the statement are substituted with the corresponding vector elements)
- the maximum number of rows to return (-1 means unlimited).

The result is a bag of result rows represented as vectors.

```
sql(odbc_ds ds, charstring query, vector params, integer maxrows)
    -> bag of vector res
```

ODBC data meta-data functions

Return information about the tables in a datasourse

```
tables(odbc_ds ds)
    -> <charstring table,
        charstring catalog,
        charstring schema,
        charstring owner>
```

Return information about the columns in a table in a datasourse

```
columns(odbc_ds ds, charstring table)
    -> <charstring column_name, charstring amos_type>
```

Return information about the primary keys in a table in a datasourse

```
primary_keys(odbc_ds ds, charstring table)
    -> <charstring column_name, charstring constraint_name>
```

Retrive all ODBC data sources

```
odbc_datasources() -> <charstring dsname, charstring description>
```

There are several more functions, which take an integer, instead of a datasource object:
(this is to be fixed later)

```
odbc_catalog(integer hconn) -> charstring x
odbc_driver_info(integer hconn)
    -> <charstring dname,
        charstring version,
        integer odbc_major,
        integer odbc_minorv>
odbc_dbms_info(integer hconn)
    -> <charstring prodname, charstring prodversion>
```

relational_ds_named(charstring nm)->relational rds

Example usage session

/* make a new datasource object, named ds, and connect it to an ODBC data source named ODBCTEST */

```
set :a = odbc_ds('ds');
connect(:a, 'ODBCTEST');
import_table('ds', 'IDAEMP');
```

/* retrieve meta-data from the ODBC data source */

```
tables('ds');
columns('ds', 'IDAEMP');
primary_keys('ds', 'IDAEMP');
```

/* direct SQL execution */

```
sql('ds', 'select name, dept from idaemp', {},-1);
sql('ds', 'select name from idaemp where dept = ?', {'IDA'},-1);
sql('ds', 'select name from idaemp where ssn = ?', {5},10);
```

/* Some example queries to the ODBC Wrapper */

```
select name(e), salary(e), ssn(e), hobby(e), age(e), dept(e) from idaemp@ds e;

select salary(e), dept(e) from idaemp@ds e;

select salary(e) from idaemp@ds e where salary(e) = 4;

select d from idaemp@ds e, integer s, charstring d
  where s = salary(e) and s = 5 and d = dept(e);
```

/* finally disconnect from the data source */

```
disconnect(:a);
```

4 Database procedures

A database procedure is an AMOS II function defined as a sequence of AMOSQL statements that may have side effects (i.e. database update statements or variable assignments). Procedures may return results by using a special result statement. Procedures should not be used in queries (but this restriction is currently not enforced). Most, but not all, AMOSQL statements are allowed in procedure bodies as can be seen by the syntax below.

Syntax:

```
procedure-body ::=
    block |
    create-object-stmt |
    delete-object-stmt |
    for-each-stmt |
    update-stmt |
    add-type-stmt |
    remove-type-stmt |
    set-variable-stmt |
    query-stmt |
    if-stmt |
    result-stmt |
    quit-stmt
block ::=
    'begin' procedure-body-semicolonlist 'end' |
    'begin'
           'declare' variable-declaration-commalist ';'
           procedure-body-semicolonlist
    'end'
result-stmt ::=
    'result' expr
for-each-stmt ::=
    'for each' ['distinct'] variable-declaration-commalist
            [where-clause] procedure-body
if-stmt ::=
    'if' predicate-expression
    'then' procedure-body
    ['else' procedure-body]
set-variable-stmt ::=
    'set' gen-variable-name '=' function-call
```

Examples:

```
create function creperson(charstring nm,integer inc) -> person p
               as
               begin
                 create person instances p;
                 set name(p)=nm;
                 set income(p)=inc;
                 result p;
               end;
set :p = creperson('Karl',3500);
create function makestudent(object o,integer sc) -> boolean
               as add type student(score) to o (sc);
makestudent(:p,30);
create function flatten_incomes(integer threshold) -> boolean
               as for each person p where income(p) > threshold
                    set income(p) = income(p) -
                                    ((income(p) - threshold) / 2);
flatten_incomes(1000);
```

Procedures are compiled at creation time.

Procedures may return (bags of) results. The `result-stmt` is used for this, where `expr` is returned as the result from the procedure.

The `for-each-stmt` construct can be used to iterate over the result of a query. For example the following function adds `inc` to the incomes of all persons with salaries higher than `limit` and returns their old incomes:

```
create function increase_incomes(integer inc,integer limit)
                                        -> integer oldinc
           as for each person p, integer i
              where i > limit
                and i = income(p)
             begin
               result i;
               set income(p) = i + inc
             end;
```

NOTICE: `result-stmt` does not not change the control flow (different from, e.g., return in C), but it only specifies that a value is to be added to the result bag of the function and then the procedure evaluation is continued as usual. The `for-each-stmt` does not return any value at all unless `result-stmt` is used within its body.

NOTICE: Queries and updates embedded in procedure bodies are optimized at compile time. The compiler saves the optimized query plans in the database so that dynamic query optimization is not needed when procedures are executed.

5 Sagas for long-running transactions

The AMOS II transaction system has been extended with *sagas*. Sagas are first class objects and can be used to chain a sequence of committed transactions with compensating transactions. The sequence of sagas can be nested by defining sub-sagas. Abortion of a saga causes all the compensations to be executed and the sagas (and sub-sagas) to be deleted. Committing a saga just causes deletion (since the transactions are already committed). Compensation of one saga is done in one complete sequence (unless stopped). If an application needs to schedule sagas (forward and backward) in smaller steps it is possible to orchestrate many sagas through a saga layer (as part of the application) outside AMOS II.

Sagas are created by the following function calls:

```
set :s = create_saga();
```

or

```
set s = create_sub_saga();
```

(only to be used within another saga)

The syntax for executing something in a saga is as follows:

```
saga-stmt ::=
```

'saga' saga procedure-body

'compensation' procedure-body

Sagas are committed (and deleted) by:

```
commit_saga(:s);
```

and are aborted (and deleted) by

```
abort_saga(:s);
```

During abortion of a saga all the compensations are executed until the beginning or until stopped by a call to

```
stop_compensation();
```

Sagas can be passed to procedures to be executed in the body of the procedure. Note that any local variables defined outside the saga statement will have the values in the compensation that they had at the end of execution of the forward transaction of the associated saga statement. If such variables are changed after the saga statement is executed this will not be seen in the compensation. To support such behavior it is possible to associate data with a saga through functions that are indexed with the current saga (can be accessed by `current_saga()`).

6 Physical database design

This section describes some AMOSQL commands for database space and performance tuning.

6.1 Indexing

The system supports indexing on any single argument or result of a stored function. Indexes can be *unique* or *non-unique*. A unique index disallows storing different values for the indexed argument or result. The cardinality constraint 'key' of stored functions (See ``Cardinality Constraints".) is implemented as unique indexes. Thus by default the system puts a unique index on the first argument of stored functions. That index can be made non-unique by suffixing the first argument declaration with the keyword 'nonkey' or to specify 'bag of' for the result, in which case a non-unique index is used instead.

For example, in the following function there can be only one name per person:

```
create function name(person)->charstring as stored;
```

By contrast, `names` allow more than one name per person:

```
create function names(person p nonkey)->charstring nm as stored;
```

Alternative definition of `names`:

```
create function names(person p)->bag of charstring nm as stored;
```

Any other argument or result declaration can also be suffixed with the keyword 'key' to indicate the position of a unique index. For example, the following definition puts a unique index on `nm` to prohibit two persons to have the same name:

```
create function name(person p)->charstring nm key as stored;
```

Indexes can also be explicitly created on any argument or result with a system procedure `create_index`:

```
create_index(charstring function, charstring argname, charstring index_type,
             charstring uniqueness)
```

For example:

```
create_index("person.name->charstring", "nm", "hash", "unique");
create_index("names", "charstring", "mbtree", "mutiple");
```

The parameters of `create_index` are:

`function`: The name of a stored function. Use the resolvent name for overloaded functions.

`argname`: The name of the argument/result parameter to be indexed. When unambigous, the names of types of arguments/results can also be used here.

`index_type`: Type kind of index to put on the argument/result. The supported index types are currently hash indexes (type `hash`) and ordered B-tree indexes (type `mbtree`). The default index for key/nonkey declarations is `hash`.

`uniqueness`: Index uniqueness indicated by `unique` for unique indexes and multiple for non-unique indexes.

Indexes are deleted by the system procedure:

```
drop_index(charstring functionname, charstring argname);
```

The meaning of the parameters are as for function `create_index`. There must always be at least one index left on each stored function and therefore the system will never delete the last remaining index on a stored function.

To save space it is possible to delete the default index on the first argument of a stored function. For example, the following stored function maps parts to unique identifiers through a unique hash index on the identifier:

```
create type part;
create function partid(part p)->integer id as stored;
```

`partid` will have two indexes, one on `p` and one on `id`. To drop the index on `p`, do the following:

```
drop_index('partid', 'p');
```

6.2 Clustering

Functions can be *clustered* by creating multiple result stored functions, and then each individual function can be defined as a derived function.

For example, to cluster the properties name and address of persons one can define:

```
delete function person.name->charstring;
create function personprops(person p) ->
        <charstring name,charstring address> as stored;
create function name(person p) -> charstring nm
        as select nm from charstring a
                where personprops(p) = <nm,a>;
create function address(person p) -> charstring a
        as select a from charstring nm
                where personprops(p) = <nm,a>;
```

Clustering does not improve the execution time performance significantly in a main-memory DBMS such as AMOS II. However, clustering can decrease the database size considerably.

7 System functions and commands

7.1 Comparison operators

The built-in, infix comparison operators are:

```
=(object x, object y) -> boolean   (infix operator =)
!=(object x, object y) -> boolean  (infix operator !=)
>(object x, object y) -> boolean   (infix operator >)
>=(object x,object y) -> boolean   (infix operator >=)
<(object x, object y) -> boolean   (infix operator <)
<=(object x,object y) -> boolean   (infix operator <=)
```

All objects can be compared. Strings are compared by characters, lists by elements, OIDs by identifier numbers. Equality between a bag and another object denotes set membership of that object. The comparison functions can, of course, be overloaded for user defined types.

7.2 Arithmetic functions

```
abs(number x) -> number y
div(number x, number y)   -> number z (infix operator /)
max(object x, object y)   -> object z
min(object x, object y)   -> object z
minus(number x, number y) -> number z (infix operator -)
mod(integer x, integer y) -> integer z
plus(number x, number y)  -> number z (infix operator +)
times(number x, number y) -> number z (infix operator *)
iota(integer l, integer u)-> bag of integer z
sqrt(number x) -> number z
```

`iota` constructs a bag of integers between l and u.
For example, to execute n times AMOSQL statement stmt do:

```
for each integer i where i = iota(1,n)
        stmt;
```

7.3 String functions

String concatenation is made using the '+' operator, e.g.
`"ab" + "cd" + "ef";` returns `"abcdef"`

```
char_length(Charstring)->Integer
```
Count number of characters in string.

```
itoa(Integer)->Charstring
```
Convert integer to string.

```
lower(Charstring)->Charstring
```
Lowercase string.

```
like(Charstring string, Charstring pattern) -> Boolean
```
Test if string matches regular expression pattern where '*' matches sequence of characters and '?' matches single character. For example:
`like("abc","??c")` returns TRUE
`like("ac","a*c")` returns TRUE
`like("ac","a?c")` fails

```
like_i(Charstring string, Charstring pattern) -> Boolean
```
Case insensitive `like`.

```
substring(Charstring string,Integer start, Integer end)->Charstring
```
Extract substring from given character positions. First character has position 0.

```
upper(Charstring)->Charstring
```
Uppercase string.

7.4 Aggregation functions

Some of these system functions are described in ``Subqueries and Aggregation Operators''.
Number of objects in bag o (``Subqueries and Aggregation Operators''):

```
count(bag of object o) -> integer c
```

Extract elements of collections:

```
in(bag of object b) -> bag of object o
in(vector v) -> bag of object o
```

Largest object in bag:

```
maxagg(bag of object x) -> object y
```

Smallest number in bag:

```
minagg(bag of object x) -> object y
```

Test if bag empty. Logical NOT EXISTS:

```
notany(bag of object o) -> boolean b
```

Test if there are any elements in bag. Logical EXISTS:

```
some(bag of object x) -> boolean b
```

Sum uniform bags of numbers

```
sum(bag of number x) -> number s
```

7.5 Temporal functions

Amos II supports three data types for referencing *Time, Timeval,* and *Date*.
Type *Timeval* is for specifying absolute time points including year, month, and time-of-day.
The type *Date* specifies just year and date, and type *Time* specifies time of day. A limitation is that the internal operating system representation is used for representing *Timeval* values, which means that one cannot specify value too far in the past or future.

Constants of type *Timeval* are written as |*year-month-day*/*hour*:*minute*:*second*|, e.g. |1995-11-15/12:51:32|.
Constants of type *Time* are written as |*hour*:*minute*:*second*|, e.g. |12:51:32|.
Constants of type *Date* are written as |*year-month-day*|, e.g. |1995-11-15|.

The following functions exist for types *Timeval, Time,* and *Date*:

```
now() -> Timeval
```
The current absolute time.

```
time() -> Time
```
The current time-of-day.

```
date() -> Date
```
The current year and date.

```
timeval(Integer year,Integer month,Integer day,
        Integer hour,Integer minute,Integer second) -> Timeval
```
Construct *Timeval.*

```
time(Integer hour,Integer minute,Integer second) -> Time
```
Construct *Time.*

```
date(Integer year,Integer month,Integer day) -> Date
```
Construct *Date.*

```
time(Timeval) -> Time
```
Extract *Time* from *Timeval.*

```
date(Timeval) -> Date
```
Extract *Date* from *Timeval.*

```
date_time_to_timeval(Date, Time) -> Timeval
```
Combine *Date* and *Time* to *Timeval.*

`year(Timeval) -> Integer`
Extract year from *Timeval.*

`month(Timeval) -> Integer`
Extract month from *Timeval.*

`day(Timeval) -> Integer`
Extract day from *Timeval.*

`hour(Timeval) -> Integer`
Extract hour from *Timeval.*

`minute(Timeval) -> Integer`
Extract minute from *Timeval.*

`second(Timeval) -> Integer`
Extract second from *Timeval.*

`year(Date) -> Integer`
Extract year from *Date.*

`month(Date) -> Integer`
Extract month from *Date.*

`day(Date) -> Integer`
Extract day from *Date.*

`hour(Time) -> Integer`
Extract hour from *Time.*

`minute(Time) -> Integer`
Extract minute from *Time.*

`second(Time) -> Integer`
Extract second from *Time.*

`timespan(Timeval, Timeval) -> <Time, Integer usec>`
Compute difference in *Time* and microseconds between two time values

7.6 Sorting functions

Currently AMOSQL has no special syntax for sorting. Instead there are several that can be used to sort bags or vectors.

Sort bags or vectors with a custom comparator function

This group of sort functions are useful to sort bags or vectors of either objects or tuples with a custom function supplied by the user. Either the function object or function named can be supplied. The comparator function must take two arguments with types compatible with the elements of the bag or the vector and return a boolean

value.

```
sort(vector v1, function compfno) -> vector

sort(vector v, charstring compfn) -> vector

sort(bag b, function compfno) -> vector

sort(bag b, charstring compfn) -> vector
```

Example:

```
create function younger(person p1, person p2) -> boolean as
select TRUE where age(p1) < age(p2);
/* Sort all persons sorted by their age */
 sort((select p from person p), 'younger');
```

Sort bags with a default comparator function

Whenever bags of literals (tuples of literals) are sorted for simplicity one may use the natural order of the literal types. Here the tuple positions are 1-based and *order* is one of 'inc' or 'dec'. Sort the bag *b* by the element at position *pos* in order specified by *order*.

```
sort(bag b, integer pos, charstring order) -> vector
```

Sort the bag of tuples *b* in default ascending order, by the element at position *pos*:

```
sort(bag b, integer pos) -> vector
```

Sort all elements in a bag *b* by comparing the whole elements in alphabetic order.

```
sort_tuples(bag b, charstring order) -> vector
```

Same as above, defaults to ascending order:

```
sort(bag b) -> vector
```

Note: Surrogate object will be sorted too, but by their OID which usually has no meaning.

Example:

```
sort((select i from integer i where i = iota(0,10)), 1, 'dec');
```

7.7 Accessing the type system

```
allfunctions() -> Bag of Function f
allfunctions(Type t)-> Bag of Function f
allobjects () -> Bag of Object o
alltypes() -> Bag of Type t
```
returns all functions, objects, and types, respectively, in the database.

```
allfunctions(Type t)-> Bag of Function
```
returns all functions where some argument or result is of type `t`.

```
subtypes(Type t) -> Bag of Type s
supertypes(Type t) -> Bag of Type s
```
returns the types immediately below/above type t in the type hierarchy.

```
allsupertypes(Type t) -> Bag of Type s
```
returns all types above t in the type hierarchy.

```
typesof(Object o) -> Bag of Object t
```
returns the type set of an object.

```
typeof(Object o) -> Type t
```
returns the most specific type of an object.

```
functionnamed(Charstring nm) -> Function fn
```
returns the function named nm. Notice that function names are in upper case.
```
kindoffunction(Function f) -> Charstring knd
```
returns the kind of the function f as a string. The result can be one of 'stored', 'derived', 'foreign' or 'overloaded'.

```
name(Function fn) -> Charstring nm
```
returns the name of the function fn.

```
typenamed(Charstring nm) -> Type t
```
returns the type named nm. Notice that type names are in upper case.

```
name(Type t) -> Charstring nm
```
returns the name of the type t.
```
objectname(Object o, Charstring nm) -> Boolean
```
returns TRUE if the object o has the name nm.
```
usedwhere(Function f) -> Function c
```
returns the functions calling the function f.
```
useswhich(Function f) -> Function c
```
returns the functions called from the function f.
```
generic(Function f) -> Function g
```
returns the generic function of a resolvent.

```
resolvents(Function g) -> Bag of Function r
```
returns the resolvents of an overloaded function `g`.

```
attributes(Type t) -> Bag of Function g
```
returns the generic functions having a single argument of type t and a single result.

```
methods(Type t) -> Bag of Function r
```
returns the resolvents having a single argument of type t and a single result.
```
resolventtype(Function fn) -> Bag of Type t
```
returns the types of the first arguments of the resolvents of function resolvent fn.

```
argrestypes(Function fn) -> Bag of <Integer pos,Type tp,Integer kind>
argrestypes(Charstring fname) -> Bag of <Integer pos,Type tp,Integer kind>
```
returns for each argument or result of a function resolvent:

`pos:` The position number. (1st is 1, etc.)
`type:` The type.
`kind:` A number indicating if it is an argument (kind = 0)or a result (kind = 1).

```
cardinality(Type t) -> Integer c
```
returns the number of object of type t and all its subtypes.

```
arity(Function f) -> Integer a
```
returns the number of arguments of function f.

```
width(Function f) -> Integer w
```
returns the width of the result tuple of function f.

7.8 Query optimizer tuning

```
costhint(Charstring r,Charstring bpat,Object q)->Boolean
costhint(Function r,Charstring bpat,Object q)->Boolean
```

Declare cost hint q for the AMOSQL resolvent function r and the binding pattern bpat. This cost hint feature is explained in AMOS II External Interfaces and in [LR92]. The cost hint can be a vector of two elements, {cost,fanout}, in case the cost to execute r is constant. It can also be an AMOSQL function returning the cost and the fanout. Strings denoting function names can be used instead of function objects.

```
optmethod(Charstring new) -> Charstring old
```
sets the optimization method used for cost-based optimization in Amos II to the method named new.
Three optimization modes for AMOSQL queries can be chosen:
`"ranksort":` (default) is fast but not always optimal.
`"exhaustive":` is optimal but it may slow down the optimizer considerably.
`"randomopt":` is a combination of two random optimization heuristics:
Iterative improvement and *sequence heuristics* [Nas93].

`optmethod` returns the old setting of the optimization method.

Random optimization can be tuned by using the function:

```
optlevel(Integer i,Integer j);
```

where i and j are integers specifying number of iterations in iterative improvement and sequence heuristics, respectively. Default settings is i=5 and j=5.

.

7.9 Miscellaneous

`eval(Charstring stmt) -> Object r`
evaluates the AMOSQL statement `stmt`.

`print(Object x) -> Object r`
prints x on the console.

`quit;`
quits AMOS II.

`exit;`
returns to the program that called AMOS II if the system is embedded in some other system.
Same as quit; for stand-alone AMOS II.

`goovi();`
starts the multi-database browser GOOVI[CR01].
This works only under JavaAMOS.

The *redirect* statement reads AMOSQL statements from a file:

```
redirect-stmt ::= '<' string-constant
```

For example

```
< 'person.amosql';
```

References

[FR95] S. Flodin, T. Risch, Processing Object-Oriented Queries with Invertible Late Bound Functions, *Proc. VLDB Conf.*, Zürich, Switzerland, 1995.

[CR01]K.Cassel, T.Risch: An Object-Oriented Multi-Mediator Browser. Presented at *2nd International Workshop on User Interfaces to Data Intensive Systems*, Zürich, Switzerland, May 31 - June 1, 2001

[LR92] W.Litwin, T.Risch: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992 (http://www.dis.uu.se/~udbl/publ/tkde92.pdf).

[Nas93] J.Näs: *Randomized optimization of object oriented queries in a main memory database management system*, MSc thesis, LiTH-IDA-Ex 9325 Linköping University 1993.