

Z++ Language Syntax Chart

Early revisions take most of C++ subset for granted. In later revisions more Z++ rules will be presented. Eventually, the C++ subset will be included as well.

Preliminaries. Two meta-symbols are used. The meta-symbol `=#` separates the left and right hand sides of a rule. Meta-symbol `#` separates the choices on the right-hand-side of a rule. All other symbols appearing on the right-hand-side of a rule are part of the syntax.

Non-terminals are shown in *italic*. A ***bold italic*** item indicates a relative-terminal in that the reader will be able to think of possible terminals at that point. A numeric expression is an example of a relative-terminal item.

Keywords are shown in **bold**.

An underlined item indicates that the item is optional, meaning 0 or 1 instance of the item can appear at that location. It is also used in recursive rules.

As an aid to readability, the above conventions are enhanced with color-coding.

Keywords are colored blue.

A ***relative-terminal*** is colored green.

All symbols, including operators are colored red, like `{}`.

Links to the start of rules for each category

[Preprocessing](#)

[Reserved Words](#)

[Operators](#)

[Fundamental Types](#)

[Control Structures](#)

[Enumeration](#)

[Collection](#)

[Exceptions](#)

[Signaling](#)

[Tell-Hear Distributed Signals](#)

[Class](#)

[Task](#)

[Frame](#)

[Union](#)

[Method Prototype](#)

[Global Function Prototype](#)

[Template](#)

[Namespace](#)

[Travel statement for Strong Mobility](#)

[Database SQL Statements](#)

[Atomic Sequence](#)

[Debug Break and Block](#)

Preprocessing

Notes. Z++ preprocessor provides a simple mechanism for conditional compilation. Text substitution and macro expansion **transform programs prior to compilation** thereby distorting abstractions. **Z++ macros can be debugged just like functions.** Z++ preprocessor identifiers are **not processed in any way**. Instead, all preprocessor identifiers are simply removed prior to compilation.

All preprocessor commands are tagged with the symbol **#** as in C++. **There are no preprocessor operators for negation or other logical operations.**

preprocess **==** *inclusion* **#** *definition* **#** *conditional* **#** *macro-def*

inclusion **==** **include** *include-operand*

include-operand **==** **“file-path”** **#** **<file-path>**

Notes. When quotes are used, compiler will search for the file relative to current directory, unless full-path is given. The angle brackets are for paths known to the compiler, such as system header files.

definition **==** **define** *identifier* **#** **undef** *identifier*

Notes. The argument of a **define** tests true until **undef** is applied to it. All unseen identifiers will test false.

conditional **==** *conditional-head* *statements* *conditional-legs* **endif**

conditional-head **==** **if** *identifier* **#** **ifnot** *identifier*

Note. The preprocessor command **ifnot** is a single word.

conditional-legs **==** *preproc-elsif-sequence* **else** *statements*

preproc-elsif-sequence **==** *preproc-elsif-head* *statements* *preproc-elsif-sequence*

preproc-elsif-head **==** **elsif** *identifier* **#** **elsifnot** *identifier*

Note. The preprocessor commands **elsif** and **elsifnot** are single words.

macro-def **==** **macro** *macro-header* *macro-body* **endmacro**

Note 1. Reserved words **macro** and **endmacro** are preprocessor commands and must be preceded with the preprocessor symbol **#**. The command **#endmacro** must appear on a line by itself at end of macro definition.

Note 2. **The body of a macro in Z++ is defined like a function body using multiple lines, without the continuation character.**

Note 3. **Declaration of local objects is allowed in macros.**

Note 4. Except for inclusion, other preprocessor commands are allowed in the body of a macro. However, definition of macros cannot be nested just like function definitions.

Note 5. Macro and function calls can be nested arbitrarily.

Note 6. **Bodies of macros can be debugged just like functions.**

Language Reserved Words

The following are all the Z++ keywords in alphabetic order.

accepts, atomic, boolean, break, canvas, case, cast, char, class, collection, common, const, continue, databaseFetch, databaseFree, databaseInsert, databaseRemove, databaseSelect, databaseUpdate, debug, delete, destroy, do, double, else, elsif, end, endatomic, enddebug, endif, enddo, endfor, endlayer, endselect, endscope, endspace, endswitch, endusing, endwhile, entry, enum, extern, external, float, for, frame, friend, from, generated, handler, hear, hears, if, implementation, inline, invariant, int, layer, local, long, menu, menubar, mutex, namespace, new, operator, pattern, persistent, private, protected, public, raise, remote, repeat, resume, return, scope, self, select, set, setchar, shared, short, signal, size, string, struct, switch, substr, task, tell, template, thread, throws, travel, type, typedef, uchar, uint, ulong, union, ushort, using, visible, void, where, while.

Operators

Notes. All operators that make sense may be overloaded. The compiler generates diagnostic messages when an operator cannot be overloaded.

Operator overloading extends to arrays. That is, when an operator is overloaded for a class, it can be applied to an array of that class. **The Z++ compiler generates the code to apply the operator to each cell of the array.**

Specialized operators associated with specific types are illustrated at those types.

Scope Operator

:: resolution operator for namespace, enumeration and method definitions

Structure Operators

. member/method selector
-> pointer member/method selector
:: base (and base member/method) selector
->> pointer base selector
:= frame to GUI canvas association

Pointer Operators

***** pointer de-reference
& take address
() pointer casting

Note. Pointer arithmetic is the same as in C++.

Arithmetic Operators (binary and assignment)

=		assignment
+	+=	add
-	-=	subtract
*	*=	multiply
/	/=	divide
%	%=	modulus (remainder)
*^	*^=	exponent (raising to a power)
&	&=	bit-wise and
 	 =	bit-wise or
^	^=	bit-wise exclusive or

<< <<= shift-left
>> >>= shift-right

Unary Arithmetic Operators

+ no action
- change sign
~ invert bits
++ increment (enumeration successor)
-- decrement (enumeration predecessor)

Notes. Only for overloading purposes, the postscript versions of increment operator ++ and the decrement operator -- use the operator forms +@ and -@. The C++ solution is to assume an integer type argument. Note that these operators are only used in defining a class (and its methods). When applying the operators use the standard forms ++ and --.

Logical Operators

! negation
&& short and (same as C++)
|| short or (same as C++)
<> long and (both operands are evaluated)
>< long or (both operands are evaluated)
^^ exclusive or (both operands are evaluated)

Relational Operators

== equal
!= unequal
< less than
<= less than or equal
> greater than
>= greater than or equal

Signaling Operators

<- send a signal
? receive a signal

GUI Operators

\$ draw/erase toggle operator
? test whether GUI object is showing
~ clear operator (field contents etc)

- <- GUI event generator
- [] GUI list element selector
- << GUI output (same for sockets and streams)
- >> GUI input (same for sockets and streams)

Other Operators

- [] array index, enumeration numeric value, collection member
- .. range operator (as in ADA)
- ,
- ?: conditional operator (same as C++)
- | - conversion operator

Note. The conversion operator is binary. It converts the right operand to its left operand. One operand must be string, and the other a numeric type. Thus, the conversion operator simply converts numeric types to ASCII, and vice versa.

The conversion operator returns the result of conversion, so it can be used as argument to function calls.

Operator cast

cast operator is the Z++ general casting mechanism. The syntax is as follows.

cast (*result-type*, *object*)

The cast operator returns an object of type *result-type* without changing its argument. If the argument *object* is **const**, the above statement will drop the **const**. To maintain the **const**, or just make the result of **cast** a **const**, use the following.

cast (**const** *result-type*, *object*)

Notes. When casting pointers in a derivation, the **cast** operator will perform a downcast, and an up-cast, in that order, using depth-first search. In fact, in a multiple-inheritance, one can cast a pointer to one base to a pointer to another base of the same object.

The C++ casting () is supported for pointers only.

A constructor taking one argument is a cast, as in **double**(2).

Operator size

The **size** operator applied to a **string** object returns the size of the string. The return object is of type **int**. The following is an example.

```
string name = "Mozart";
```

```
int length = size(name); //sets length to 6
```

The **size** operator is also applicable to dynamic arrays. Here is an example.

```
int m = 5, n = 7;

// m and n may change before we reach the next line

double array[m][n]; // two-dimensional dynamic array
int first = size(array, 0); // size of first dimension -- m
int second = size(array, 1); // size of second dimension -- n
```

Operator **new**

The **new** operator is the same as in C++, with some correction and extension. Consider the following.

```
int * p = new int[m];
```

Unlike C++, the above is an error in that the type returned by the **new** operator is not a plain integer pointer. The actual type is a pointer to a dynamic array of integers. By dynamic we mean that the value of *m* is unknown at compile time.

First, we need to define a type for a degenerate dynamic array of integers. Note that the order of **typedef** is the reverse of that of C++. The **typedef** name below is `degenIntArray`.

```
typedef degenIntArray int[];
degenIntArray * p = new int[m]; // this is fine
```

When constants are used for array indices, the types on both sides must match exactly. That is, degenerate arrays cannot be used.

```
typedef IntArrayPointer int[20]*;
IntArrayPointer q = new int[20];
```

Z++ also extends the **new** operator so general constructors can be called on array cells at time of construction. The following example illustrates the simplicity of Z++ expressiveness.

```
new my_array_type[m][n](x, y, z);
```

Operator **delete**

The **delete** operator is the same as in C++, except Z++ does not allow the use of brackets, as in “delete []”. This is a consequence of correcting the semantics of **new** operator.

Fundamental Types

All C++ built-in types are supported. Z++ does not use the term unsigned. Instead, the letter u is used as a prefix, as in **ulong** for unsigned **long**.

In Z++ instances of any type are objects. However, unlike Eiffel in Z++ one cannot derive from the fundamental types, as there is no point in doing so. On the other hand, unlike C++ there are no l-value and r-value. **An object is an object regardless of the side on which it happens to be.**

The following are also Z++ fundamental types.

Boolean. The compiler predefines the objects **True** and **False** of type **boolean**. Among other things, they can be used for initializing other objects, as follows.

```
boolean young = False, old = True;
```

Objects of type **boolean** cannot be mixed with objects of numeric types. However, the logical negation operator can be applied to pointers. The result is **True** exactly when the pointer is null.

String. The type **string** is fundamental. The language provides direct support for common operations, and the string library extends the support.

Operators **+** and **+=** are **string** concatenation operators.

Operator **size** returns the length (number of characters) of an object of type **string**.

The Z++ string library uses the low-level operators **substr** and **setchar** to provide all the needed functions with familiar syntax.

Mutex. The type **mutex** is fundamental Z++ type. An instance of **mutex** can be defined as usual, and as pointer, as shown below.

```
mutex mtx;  
mutex* mtxptr = new mutex;  
delete mtxptr; // after use
```

Instances of type **mutex** can be passed to calls. Operator **++** locks a mutex while operator **--** unlocks it.

Constructs for Structured Programming

Notes. All Z++ control structures have a closing tag, and do not require the use of braces. However, braces are allowed for opening additional scopes.

The type **boolean** is a fundamental type. Therefore, a boolean expression cannot be mixed with a numeric expression.

The Z++ compiler predefines two instances of type **boolean**, **True** and **False**.

All C++ boolean operators retain their semantics in Z++. A boolean operator is short when its right-hand-side operand is not evaluated unless needed. A long boolean operator means that both operands of the operator are evaluated before evaluating the result of the operator. The C++ operators **||** and **&&** are short, as they are in Z++. However, Z++ provides the following additional boolean operators.

- ^^** long exclusive or (there is no short boolean exclusive or).
- <>** long and.
- ><** long or.

Z++ allows chaining of relational operators as it is done in mathematics. Thus, the expression $(v < x == y <= z)$ will be interpreted in its usual sense.

control-structure ::= *selection* # *iteration*

selection ::= *if-statement* # *switch-statement* # *conditional-statement*

if-statement ::= **if** (*boolean-expression*) *statements* *leg-sequence* **endif** ;

leg-sequence ::= *elsif-sequence* *else-leg*

elsif-sequence ::= **elsif** (*boolean-expression*) *statements* *elsif-sequence*

Note. The Z++ **elsif** keyword is similar to that of ADA.

else-leg ::= **else** *statements*

switch-statement ::= **switch** (*expression*) *switch-details* **endswitch** ;

switch-details ::= *statements* *case-else-segment*

Notes. Z++ switch construct allows statements before the first case leg. Furthermore, objects created here are available within the scope of cases and the else leg. However objects created in each leg are only available within the scope of that leg.

Since **string** is a built-in type, case labels can also be string literals.

case-else-segment ::= case-sequence *else-leg*

case-sequence ::= *case-leg* case-sequence

case-leg ::= **case** *case-labels* : statements

case-labels ::= *label-sequence* # *label-range*

label-sequence ::= **label-literal** , label-sequence

Note. A single literal is also a sequence.

label-range ::= **label-literal** .. **label-literal**

else-leg ::= **else** *statements*

Notes. Sequence and range are similar to that of ADA. Z++ does not use **break** for selections (only for iterations). However, one may assume an invisible break for each case. Thus, a case leg without any statements does not fall through to the next case leg.

conditional-statement ::= **boolean-expression** ? *expression* : *expression* ;

Notes. There are two differences between Z++ and C++. First, the Z++ compiler requires that the two expressions result in an object of the same type. Second, the result of the statement is an object in that it is possible to invoke methods on it (enclose the entire statement within parentheses).

iteration ::= *for-loop* # *while-loop* # *do-loop*

Notes. The keywords **break** and **continue** retain their C++ semantics.

for-loop ::= **for** *for-details* **endfor** ;

for-details ::= See remark below.

Notes. The syntax and semantics of the Z++ for-loop is identical to that of C++, except for the closing tag **endfor** and the lack of braces for the body of the loop.

while-loop ::= **while** (*boolean-expression*) statements **endwhile** ;

Notes. The boolean expression for a while-loop is evaluated prior to entering the loop body. This is the opposite of the do-loop below.

do-loop ::= **do** statements **enddo** (*boolean-expression*) ;

Notes. Note that the boolean expression and its surrounding parentheses are underlined, meaning that it is optional. When the boolean expression is left out the do-loop becomes a simple infinite loop.

The boolean expression, if present, is evaluated after all the statements in the loop have been executed. Thus, the do-loop will always execute at least once.

Note. A do-loop ends when its boolean expression evaluates true. This is the opposite of a while-loop, which iterates so long as its boolean expression evaluates true.

Enumeration

enum-type ::= **enum** *enum-identifier* *enum-details* ;

enum-details ::= *enum-extension* { *enum-value-list* }

enum-extension ::= : *enum-type-identifier*

Notes. The values of the enumeration type being defined will follow those of the type being extended, i.e. *enum-type-identifier*.

The type being extended is also called a base for the new (extended) type.

enum-value-list ::= *enum-literal* ≡ *int-literal* , *enum-value-list*

Note. *int-literal* is an integer numeric literal.

enum-literal ::= *identifier*

Notes. An enumeration literal must start with an underscore character ‘_’. Regular Z++ identifiers cannot start with underscore.

The literals of an enumeration type are private to the type in that the same literal can be used in defining other enumeration types. When compiler reports ambiguous for a particular literal, qualify the literal with its type, as in `type-name::literal`.

Exceptions and signals are of enumeration types defined in Z++ system header files. A user-defined exception type or signal type is simply an extension of these system types.

Enumeration operators. The type constructor enumeration provides operators for recurring operations.

Operator `::` is for ambiguity resolution of literals, as mentioned in the notes above.

Operator `[]` when applied to an enumeration object/literal evaluates to the integer value associated with that object/literal. For instance, `[some_literal]` returns an int value.

Operator `[]` when applied to an enumeration **type** evaluates to the least enumeration literal of the type. Note that, the operand is a type (not object or literal) and that the result is an enumeration literal (not an integer).

Operator `[]` evaluates to the largest enumeration literal of the type to which it is applied. That is, the operand is an enumeration type, such as `[[my_enum_type]]`, and the result is the largest enumeration literal value of `my_enum_type`.

Operator `++` is the successor function. When applied to an object of type enumeration it increments the objects values to the next enumeration literal for the type of object.

Operator `--` is the predecessor function and is analogous to `++`.

Collection

Note. Collection is a type constructor. It can be viewed as extending the type constructor [enumeration](#) where the values associated with enumeration literals are instances of classes instead of integers.

In addition, collection can have methods. In particular, shared methods of a collection invoke identical methods of each of the classes associated with the collection. This allows invoking methods on a set of objects whose classes are not related through inheritance.

collection-type ::= **collection** *collection-identifier* < *enum-type* > *collection-details* ;

Note. *enum-type* is name of a previously defined [enumeration](#) type.

collection-details ::= *collection-derivation* { *collection-body* } ;

collection-derivation ::= : *collection-identifier*

Note. *collection-identifier* in this context is name of a previously defined collection type from which the new collection is being derived. Derivation is only public and cannot be specified. Furthermore, derivation is linear (single inheritance).

Note. When deriving collections, the associated enumeration type of the derived collection must be an extension of the associated enumeration type of its base.

collection-body ::= *collection-values* *collection-member-section*

collection-values ::= *enum-literal* < *class-type* > , *enum-literal* < *class-type* >

Note. *enum-literal* is a literal of *enum-type* associated with collection, as in first rule above. The list is comma separated and must cover all literals of enumeration type. Each literal is associated with a previously defined class. At elaboration, each enumeration literal receives an instance of its associated class for its value.

collection-member-section ::= *access* # *collection-methods* # *shared-methods*

access ::= **private** : # **protected** : # **public** : # **shared** :

Note. Access kind **shared** can only be specified once as starting shared section. Its scope extends to end of definition of **collection**. The access of shared methods within the shared section, as **public** or **private**, will be whatever it was before the keyword **shared** was seen, and can also be changed within the shared section.

Note. Methods specified within shared section must be methods of each of the classes used for values of **collection**. When a new **collection** is derived from a **collection** that has shared method, the new classes of derived **collection** must have the shared methods of base **collection**. This is because when a shared method is invoked on an instance of **collection**, it is applied to all instances of classes, including the ones of the derived **collection**.

Note. When deriving new collections, the derived **collection** can introduce new **shared** methods. However, the new **shared** method will only be called on the new values (classes) of the derived **collection**.

Note. The body of a **shared** method is generated by, the compiler. Furthermore, a **shared** method of base does not need to be specified again in a derived collection.

Example. Consider the following enumeration and classes.

```
enum shapes {_square, _circle};

class Square
    // members, methods
public:
    // constructors etc.
    int length(void);
    double showArea(void);
end;

class Circle
    // members, methods
public:
    // constructors etc.
    double showArea(void);
end;
```

Now, we define a collection named MyShapes.

```
collection MySahpes<shapes> {
    _square<Square>,
    _circle<Circle>
    // constructors etc.

    void print(void); // default access is public
shared: // shared section extends to end of definition

    double showArea(void);
};
```

Note. The return object of a shared method is the object returned by invoking the method of the last value of the **collection**. In the above example, the shared method showArea() will return the double returned by the showArea() of instance of Circle.

Note. Inside the body of a method of a **collection**, the keyword **self** references the object itself, as it does for classes.

Note. An instance of a **collection** has an implicit tag. Initially the tag points to the first value (instance of class). The tag can be accessed, as well as set, as in the following example. In particular, the successor and predecessor functions work the same way as they do for [enumeration](#).

Note. The array brackets behave similarly for collections, also illustrated in the following example. This allows invoking methods on instances of classes that are values of a **collection**.

Example. Using previous example in this section.

```
[MyShapes] = _circle; // set the tag to point to Circle
MyShapes--; // now tag points to Square
MyShapes[_square].length(); // invoke a method of Square

void MyShapes::print(void) // definition of collection method

    switch([self]) // self refers to collection object

        case _square:
            output << "My tag is at Square\n";

        case _circle:
            output << "My tag is at Circle\n";

    endswitch;

end;
```

Exception

exception-statement ::= *layer-statement* # *raise-statement* # *resume-statement*

layer-statement ::= *exception-head* statements *handler-section* **endlayer** ;

Notes. Z++ does not use the C++ keywords try and catch.

exception-head ::= **layer** < *exception-type* >

Notes. The exception-literals that appear as case labels of the handler section must be of the type *exception-type* specified for the layer.

Exceptions are of enumeration type. New exception types are defined by extending system exception type.

handler-section ::= **handler** *handler-details*

Note. The *handler-details* is same as case-else-segment of switch statement except labels must be of an exception type.

raise-statement ::= **raise** *exception-literal* ;

Note. The **raise** statement is equivalent to C++ throw.

resume-statement ::= **resume** # **repeat** ;

Notes. The **resume** and **repeat** statements can only appear in *handler-section*. Repeat returns control to the start of the statement that caused the exception. Resume returns control to the statement following the one that caused the exception.

Generally, use **repeat** when the cause of exception can be repaired and the operation can start over. On the other hand, use **resume** when logging or posting messages about a failure, and continue the operation without performing the action that caused the exception.

Signaling

Notes. The keyword **signal** is used for generating a signal within an executing program, as well as catching the desired signals. In the following rules the *expression* must evaluate to a *signal-literal*.

User-defined signals are of [enumeration](#) type extending system signals.

signal-statement ::= *generate-signal* # *catch-signal*

generate-signal ::= **signal** <- *expression* ;

Note. The symbol <- is backwards form of C++ (and Z++) symbol for pointer dereferencing. This statement generates the signal resulting from expression. The Z++ virtual processor will keep the signal until it is delivered to its destination.

catch-signal ::= **signal** ? *expression*

Notes. *catch-signal* is a boolean expression.

A thread uses the *catch-signal* expression to wait until Z47 Processor delivers the signal to it. The expression *catch-signal* remains false until the expected signal is generated. At that time, Z47 Processor removes the signal from the context of the process and sets the *catch-signal* expression to true.

Processes and their threads can communicate to one another by sending different user-defined signals. Z++ has various categories of signals discussed below.

All categories of signals are defined in Z++ system header file. User-defined signals of each kind are defined by extending the relevant category in system header file.

signal-category ::= *process-signal* # *node-signal*

process-signal ::= *plain-process-signal* # *entire-process-signal*

node-signal ::= *plain-node-signal* # *entire-node-signal*

The above rules show four categories of signals.

A process-signal, or process-bounded signal is for signaling among threads of a single process. Only threads of the process that generates this kind of signal can catch it.

A node-signal or node-bounded signal is for signaling among process running on same Z47 Processor. For signaling among processes running on different Z47 Processors see [Tell-Hear](#) signals. All processes, including the process that generated the signal can catch the signal. This is useful for communication among threads of different processes.

A plain signal, either process-bounded or node-bounded can only be caught once, as first come first serve. Once caught the signal is removed from the Z47 Processor queue.

Catching an entire signal, process-bounded or node-bounded requires registration, which will be discussed shortly. An entire signal is delivered to all threads or processes that have registered to receive the signal prior to removing it from the Z47 Processor queue.

Note. A process-bounded entire signal can be caught by, all threads of the process that generated it, except the main startup thread of the process.

Catching an entire signal requires registration for accepting the signal. A thread or process registers the signals it desires to be delivered to it with Z47 Processor via its accepts-specification.

accepts-specification ::= **accepts** (*sig*)

A global thread registers its accept signal as shown below.

```
void MyGlobalThread(void) <thread> accepts (_sig);
```

For a process, use the accept specification at its **entry** point. Since a process can have any number of entry points, the signals registered will depend on the entry point invoked.

Registration of entire signals for task threads is discussed at [task-accepts](#).

Tell -Hear Distributed Signals

Note. Tell-Hear mechanism is an abstraction for a distributed model of Concurrent Communicating Processes. The signaling mechanism of Z++ is extended for asynchronous communication among distributed processes. These signals can also carry data along to their remote (or local) destinations.

Note. User-defined tell-hear signals are derived from Z++ base type the same way as other user-defined signals are derived (enumeration extension).

In the rules below, tell-side is for statements that the sender of a signal would execute, and hear-side are statements that the recipient would execute.

tell-hear ::= *tell-side* # *hear-side*

tell-side ::= **tell** <- *tell-details* ;

Note. These signals use the keyword **tell** instead of **signal** for generating a signal. However the operator is the same. The idea is that a **process is telling something to another process that it expects to hear it.**

tell-details ::= **sig** *tell-destination* : *tell-data*

Note. The none-terminal sig is a user-defined signal. Tell-destination and tell-data are optional. The separator colon is only needed when tell-data are present.

tell-destination ::= \$ *node-destination* , *process-destination*

Note. Separator \$ precedes node-destination and separator comma precedes process-destination. Since both destinations are optional their separators are present only when their expressions are. Both destinations are of type **string**.

Note. A node-destination is a URL. A process-destination is the name of process at target node. This is simply the name of the application without the extension ZXE.

Note. Without a URL, Z47 Processor assumes itself. That is, the signal is considered for local delivery.

Note. The name of a process, that is the argument process-destination, is only useful when there is a possibility that two different processes may have registered to catch the same signal. In that case, without specifying the name, the first process that catches the signal will serve it, and the other process will not hear the signal.

tell-data ::= **object** , *more-tell-data*

more-tell-data ::= **object** , *more-tell-data*

Note. Tell-data is a comma-separated list of objects. The types of these objects are specified in hears-specification, which we discuss next. That is, for each signal one specifies the types of objects to be received along with the signal so the compiler can verify objects of correct type, and in correct order are attached to that signal. This is the same as type checking the signature of a function call, in this case the check is done when the tell statement is parsed.

Now we discuss the hear-side. A tell-hear signal (or for short we may say tell signal or hear signal depending on context) requires registration. The registration of a hear signal is specified in the header or the prototype of a global thread, as illustrated for [accepts](#) specification, and will also be illustrated shortly. For **task** threads see [task-hears](#). The keyword for registering hear-signals is **hears**.

hear-side ::= *hears-specification* # *hear-catch*

hears-specification ::= **hears** (*sig* *hear-specification-details*)

hear-specification-details ::= < **object-type** , *more-hear-types* >

more-hear-types ::= **object-type** , *more-hear-types*

So, a hear-specification includes the signal, and optionally a comma-separated list of types enclosed between the symbols <>. The types, object-type, must have been defined previously.

After the following rules we will illustrate the semantics of tell-hear signals. Now we need to know how to catch a hear signal.

hear-catch ::= **hear** ? *hear-details*

The keyword for catching a hear-signal is **hear** instead of **signal** as for ordinary signals. However, the operator is the same, the symbol ?. Furthermore, the expression is boolean just as it was for ordinary signals.

Note. In the following rule the colon is needed only if hear-objects is present.

hear-details ::= *sig* : *hear-objects*

hear-objects ::= **object** , *hear-objects*

So, in addition to the signal to catch, a hear expression may include a list of comma-separated objects. These objects must have been defined earlier.

For illustration, assume `object_1` and `object_2` of types `type_1` and `type_2` have been defined. Furthermore, let `_sig` be a user-defined tell-hear signal.

An application named `MyHearApp` on a node at `URL` is started up. A thread of this application has the following specification.

```
void HearThread(void)<thread> hears(_sig < type_1, type_2 >);
```

When the thread `HearThread` begins, before doing anything else it registers the signal `sig` with `Z47 Processor`. The registration includes the types associated with this hear signal, in this case `type_1` and `type_2`.

Somewhere in the body of this thread we need to catch and serve the arrival of this signal. So, we may have a piece of code as follows.

```
if (hear ? _sig : object_1, object_2)
// do something
endif;
```

Assume at some point in time the signal `_sig` eventually arrives. All hear signals arrive at `Z47 Processor`. The Processor checks to see if such a signal with exact types has been registered. If not it will reject the signal and inform the sender. Otherwise, it will accept and record the signal along with objects sent with the signal.

Eventually the thread `HearThread` gets its time slice. If there is a signal waiting for the thread, and we reach the above code, `Z47` delivers the signal and the hear-expression becomes true. `Z47` also copies `object_1` and `object_2` to the context of the thread, and removes the signal from its queue. At this point `object_1` and `object_2` are exactly what the sender included in its tell statement, discussed next.

Somewhere in the universe, an application called `MyTellApp` begins execution on a `Z47 Processor`. At some point in this process we meet the following code.

```
tell <- _sig $ URL , MyHearApp : object_1, object_2;
```

`Z47 Processor` initially sends the signal `_sig`, the name of the process `MyHearApp` and the types of objects to the node at `URL`. If the receiving `Z47` at `URL` rejects the signal, it raises an exception. Otherwise, `Z47` packages `object_1` and `object_2` in binary and sends it to the receiving `Z47` at `URL`, and moves on to the next statement.

Note that `MyHearApp` at `URL` can later send a tell-hear signal to the node executing the application `MyTellApp`. Thus, two processes can engage in an asynchronous communication transferring complex data to one another in binary, without blocking.

Class

Note. In this revision we only present main features of Z++ as they differ from C++. Especially, the semantics of task for threading, and frame for GUI are not illustrated. Moreover, the semantics of various Z++ linkage mechanisms are only briefly discussed.

class-definition ::= *class-kind* **class-name** *class-specs* *derivation* *class-details* **end ;**

class-kind ::= **class** # **struct** # **task** # **frame**

Note. An instance of **task** is created in a new thread. The type constructor **frame** is for Graphical User Interface (GUI) implementation.

class-specs ::= *frame-canvas-specs* # *linkage-specs*

frame-canvas-specs ::= **:= canvas-identifier**

Notes. A **canvas** looks like a structure, and is generated by tools for creating GUI. However, a **frame** is a Z++ type that manipulates the **canvas** associated with it.

Graphic-related rules will be presented in future revisions.

linkage-specs ::= *language-linkage* # *module-linkage* **< execution-location >**

language-linkage ::= **“ language “ = CppLibrary-name**

Notes. The only language currently accepted is C++. **CppLibrary-name** is path to a C++ dynamic library (not a URL).

For all C++ libraries in a Z++ program, the compiler creates a C++ connector as a dynamic library that the virtual processor uses. The name of this connector library can be set by the rule: **extern** “C++” = “user-defined-name”.

module-linkage ::= **= ZppModule-name # = ZppModule-URL**

Note. **ZppModule-name** is the path to reach the program on the local machine.

ZppModule-URL is the URL to reach a Z++ program on a remote machine. The URL must start with “Zpp://”. The Z++ virtual processor currently uses port 1011.

execution-location ::= **local** # **remote**

Note. Default is **local**. When **local** is specified, the remote module is downloaded to the local machine and executed there. Specifying **remote** causes the remote module to execute on the remote machine. Z++ virtual processor performs the RPC transparently.

derivation ::= : *derivation-sequence*

derivation-sequence ::= *derivation-control* **derivation-item** , *derivation-sequence*

derivation-control ::= **public** # **private**

Note. Default is **public**.

The term virtual is not a Z++ keyword. Instead of virtual base, the Z++ compiler uses depth-first search to resolve ambiguity.

class-details ::= Z++ *Extension*

Notes. Since Z++ extends C++, for the first revision of this document, we will only present extensions. However, the following corrections are also done to C++.

Static. The term static is not used in Z++. However, static members of C++ are same as **common** members of Z++, with corrections to C++. There are no static methods in Z++. The methods to manipulate a **common** member are called authorized methods for that member.

A **common** member is initialized the same way as a static member is initialized in C++.

Virtual. The C++ keyword virtual is not used in Z++. All methods that can be virtual are in fact virtual without the need to use the keyword. However, the Z++ compiler determines when to make a polymorphic call, and when to perform static binding.

Reference. A member of a class cannot be declared as a reference to another object. Pointers are supported just as C++. However, the semantic of reference is not identical to that of a pointer.

Privacy. It is not possible to return the address, or a reference to a non-public member of a class. However, it is allowed to return a const reference to a non-public member. Using the casting mechanism to remove the const, one can finally achieve returning a plain reference to a non-public member. But how can several deliberate steps be the result of an accident? Nevertheless, in the future the Z++ compiler may be able to trap most of these abuses.

Invariants. A Z++ class can have invariants. Invariants for a class are tested at end of public methods of class (transparently).

The semantics is that, the specified action will be taken when the condition of invariant becomes false. The action could be raising an exception, or calling a method (trigger).

Z++ Extensions ::= common-member # class-invariants

class-invariants ::= exception-invariant # trigger-invariant

exception-invariant ::= invariant (boolean-expression) exception-literal

Note. The exception will be raised when the boolean-expression becomes false.

trigger-invariant ::= invariant (boolean-expression) function-call

Note. The call will be made when the boolean-expression becomes false. The call must be to a none-public method of the class.

common-member ::= common member-declarator : authorized-methods

Notes. Authorized methods of a member are the only methods that can modify the member. All methods can access a **common** member.

A **common** member cannot be **public**. A **common** member is initialized the same way as it is done in C++. However, unlike C++ accessing and manipulating a **common** member can only be done through instances of the class (as opposed to global static method).

For now we leave *member-declarator* vaguely to mean a C++ member declaration.

authorized-methods ::= method-prototype , authorized-methods

Note. The non-terminal *method-prototype* in this context is a simple form of prototype to identify the method. In particular, the prefix-specifications like **inline**, or tail specifications like **const** are not allowed.

Note. A **private** or **protected** member can be made **visible** for read access only. A **visible** member does not need a method to return its value.

class-member ::= member-declarator < visible >

Task

Note. Differences between a **task** and a **class** are as follows.

An instance of a **task** is created in a new thread. That is, first a new thread is created and then the instance of **task** is created in this thread. The thread is destroyed when the instance goes out of scope, or in case of a dynamic instance when the task object is destroyed.

Calls to public methods of a **task** object are queued (by the Z47 Processor). The methods are executed when the task thread receives its time slice.

Remark. A task object is destroyed after it services all requests in its queue.

Note. When a **task** is derived from other tasks, at elaboration each base task object is created in its own thread. Thus, an instance of a **task** could be multi-threaded.

Note. Each **task** data member of a **task** is created in its own thread. This also means an instance of a **task** could be multi-threaded.

Remark. In general, whenever a **class** could be used, one could also use **task** instead. In particular, the syntax for a **task template** is identical to a **class template**.

Task Idler

Task idler is a private method that cannot be called directly. The idler's prototype is similar to that of destructor, except the symbol ~ is replaced with @.

A task object executes its idler whenever there are no requests waiting in its queue, nor the task object is responding to signals.

Task Signal Handlers

Task signal handlers are private methods that cannot be called directly. Instead, task object invokes its handlers as their specified signals arrive. The syntax for a handler prototype is as follows.

```
void handler_name(void)<signal_name>;
```

That is, the return and the signature are void. The prototype name is followed by the signal for the handler, within brackets <>. **The definition of a handler does not need the signal specification.**

The prototype for a handler for a hear-signal is illustrated below.

Task accepts and hears specifications

Specifications of **accepts** for entire signals and **hears** for tell-hear signals are done in the body of the definition of a task type, just like **task** invariants. Here we illustrate the specifications. See rules for [accepts](#) and [hears](#).

Suppose the following specification is done in the definition of a **task**.

```
accepts (_sig1);
```

When an instance of this **task** is declared, `_sig1` is registered as an entire signal. This means, all instances of this task type will in fact receive that signal whenever it is generated. Therefore, all instances will invoke their associated signal handler to serve the signal.

Now suppose the following **hears** specification is done in the definition of a **task** type.

```
hears (_sig2 < type_1, type_2 >);
```

At declaration of an instance of this type the signal `_sig2` along with its types will be registered for this instance. However, note that tell-hear signals are not entire. That means if several instances of this type are created, only one instance at any one time will catch a generated hear-signal. Just which instance will serve the signal cannot be determined.

The specification of a handler for a hear-signal requires objects to receive the data sent along with the signal. This is only necessary if the signal actually carries data, of course.

The code below illustrates the prototype of a handler for `_sig2`.

```
void HearHandler(void)<_sig2 : object_1, object_2>;
```

The objects, `object_1` and `object_2` **must be members of the task**, of types `type_1` and `type_2`.

Frame

A **frame** is analogous to a **class**. However, **frame** is specifically for object-oriented graphical user interface (GUI). In order to illustrate the use of a **frame**, we begin with the notion of **canvas**.

A **canvas** is generated by, the GUI-Maker tool in a header file. Below is an example of a **canvas**.

```
canvas MyCanvas
    button Done;
    label Name;
    checkbox One;
    radiobutton First;
    combobox Choices;
    field Text;
end;
```

The strings, Done, Name etc. are strings that appear on the GUI objects, like button, whenever applicable. The same strings, though, are used in handling events associated with those objects, as illustrated later in this section.

The drawing of a canvas is the responsibility of Z47 Processor. In order to use the **canvas** we associate it to a **frame**, using the association operator **:=** as follows.

```
frame MyFrame := MyCanvas
    // members
public:
    // constructors, methods, etc.
    $MyFrame(interfaceEventType&); // instinct method
end;
```

A **frame** has an additional special method, namely its instinct method. The instinct method has the same name as the **frame**, but its name is preceded with the dollar sign. This is similar to a destructor except the symbol ~ is replaced with \$.

The type of argument to the instinct method is a structure defined in a system header file. Two of the members of this structure are of interest, namely the member 'entity' and the member 'Event'. We will see their use in defining the body of the instinct method.

It is the responsibility of the Z47 Processor to invoke the instinct method of a frame and to pass GUI events, such as mouse-click, to it. The engineer writes the body of the instinct method to deal with events, as illustrated below. Thus, the instinct method is object-oriented form of an input wait loop.

Below is an example of how the body of an instinct method is defined. In particular, observe the use of the members 'Event' and 'entity' of the argument passed to the instinct method. These are set by, the Z47 Processor when it invokes the instinct method.

```

MyFrame::~MyFrame(interfaceEventType& e)

    switch(e.Event) // do a switch on the type of event

        case _IES_Draw_Signal:
        case _IES_Erase_Signal:
        case _IES_Mouse_Click_Signal:

            select(e.entity) // for each event, do a select on GUI objects

                case Done:
                case One:
                case Choices:

            endselect;

        endswitch;
end;

```

Observe that the labels of select statement are the names of GUI entities as they appear in the **canvas**. The signals are defined in a system header file. The basic idea is to switch on the arrived event. Then, for each event of interest, do a select on names of GUI objects that you intend to manipulate. We have not shown any code except the skeletal form of the body of an instinct method.

Z++ GUI operations are platform independent, and are illustrated in other documents where GUI objects are discussed.

Union

Note. Z++ unions are same as C++ with following differences.

Data members of **union** can only be pointers or numeric, **char**, **int**, **double** etc.

Union is a type constructor and must therefore have a name. Unnamed unions hidden in a **class** are not supported.

Note. Just as in C++, an instance of a **union** allocates one slot for all its data members.

Note. As in C++, unions cannot be derived. All data members and methods of a **union** are **public**.

Method Prototype

method-prototype ::= prefix-specs *prototype* tail-specs ;

prefix-specs ::= **inline** # **external**

Notes. The keyword **inline** has same semantics as in C++.

An **external** method is Z++ specific. The Z++ compiler generates code for the body of an **external** method. The simple view is that an **external** method is the **entry** point of a module that the class being defined is representing in the context of current program.

The non-terminal *prototype* is to be taken for a simple C++ function prototype until future revisions. The underlining indicates that all items on the right of the following rule are optional, while all may be present simultaneously.

tail-specs ::= **const** **cast** exceptions constraints

Notes. A **const** method has the same semantics as in C++.

The semantics of **cast** is same as C++ explicit (which is not a Z++ keyword). The keyword **cast** is also used for all forms of casting in Z++, instead of multiple C++ forms of casts.

exceptions ::= **throws**(*exception-list*)

Note. The Z++ compiler uses the list of exceptions for **throws** in its verification that all raised exceptions are eventually handled.

exception-list ::= *exception-literal* , exception-list

constraints ::= { *constraint-sequence* }

Notes. The semantics for a constraint is that when the condition of the constraint becomes false, the specified action will be executed. Note that constraint statements are enclosed between braces. Definitions of method-bodies are not allowed within the definition of a **class**.

Constraints of a method are tested prior to executing the code of the method.

constraint-sequence ::= *constraint-item* ; constraint-sequence

constraint-item ::= *exception-constraint* # *trigger-constraint*

exception-constraint ::= (*boolean-expression*) *exception-literal*

Note. When boolean-expression becomes false, specified exception will be raised.

trigger-constraint ::= (*boolean-expression*) *function-call*

Note. When boolean-expression becomes false, specified call will be made. The call must be made to a **private** method of the **class**.

Global Function Prototype

function-prototype ::= *global-prefix-specs* **prototype** *global-tail-specs* ;

Note. The non-terminal **prototype** is to be taken for a simple C++ function prototype until future revisions.

global-prefix-specs ::= **inline** # **entry**

Notes. The keyword **entry** is Z++ specific. A function specified as **entry** is called a module entry-point.

A Z++ program can have any number of entry-points, but must have at least one.

A Z++ program is also a module (component) and can be used in composing larger programs. **Thus, there is no distinction between a program and a component.**

global-tail-specs ::= *global-thread* **exceptions**

Note. The **exceptions** item is the **throws** specification as shown for methods. The underline indicates that it is optional.

global-thread ::= < **thread** *disengage-list* > *thread-tail-specs*

Note. The Z++ virtual processor delivers the signals in a disengage list to the thread. For instance, when a server thread is blocked on a socket accept(), upon arrival of the signal the virtual processor will disengage (unblock) the thread.

disengage-list ::= : *disengage-signals*

disengage-signals ::= **signal-literal** , *disengage-signals*

A global thread can registers **entire** and **tell-hear** signals. The right side of the following rule gives two hyperlinks to where the rules are defined. Both specifications can be present for a thread.

thread-tail-specs ::= [accepts-specification](#) [hears-specification](#)

Note. An **entry** can have [accepts-specification](#) for registering entire signals.

Template

Notes. Z++ provides a pattern construct for specifying permissible types for instantiating a parameterized (template) type.

The C++ template construct is slightly extended to allow attaching a pattern to a type parameter.

template-construct ::= *template-tag* # *template-pattern*

template-tag ::= **template** < *template-parameters* >

Notes. A *template-tag* is followed by a class, method or function definition. The scope of a *template-tag* is the entity following it.

The definition of an entity following a *template-tag* may use a template parameter anywhere a type can appear.

When defining a template class, Z++ does not require the use of template parameters as part of name of class. For instance, when defining `My_Class < U, V >` the class itself is referred to as `My_Class` in the body of the definition.

template-parameters ::= **type identifier** : *pattern-identifier* , *template-parameters*

Notes. The term **type** is a Z++ keyword. The *identifier* following the keyword **type** is the name of type parameter.

A *pattern-identifier* is the name of a previously defined *template-pattern*.

template-pattern ::= **pattern template** < *parameter* > *pattern-details* **end** ;

parameter ::= *identifier*

Note. The *parameter* is used in pattern statements, below.

pattern-details ::= *pattern-identifier* *pattern-statements*

pattern-statements ::= *pattern-type-specs* # *pattern-method-specs*

Note. A **pattern** can either list the types that can be used to instantiate a **template** parameter, or the methods that the instantiating type must define.

pattern-type-specs ::= *parameter* : *type-sequence* ; *pattern-type-specs*

Note. The above rule lists the permissible types that can instantiate a template parameter.

type-sequence ::= **type-identifier** , *type-sequence*

pattern-method-specs ::= *plain-prototype* # *parameter-prototype* # *wild-prototype*

Note. The above rule specifies the **public** methods that a type must define so it can be used to instantiate a template parameter.

plain-prototype ::= See note, below.

Note. This is an ordinary function prototype, without the use of wild substitution mechanisms. The instantiating type must define a method with exact name and signature.

Note. A constructor prototype is specified with the parameter identifier used for name of constructor. For instance, T(**string**) is a constructor taking a string by value for its argument, where T is the template **pattern** parameter identifier.

parameter-prototype ::= See note, below.

Note. The *parameter* identifier, such as T acts like a wild type. The compiler will match the prototype with a **similarly named method prototype and exact signature** except for the types where the *parameter* identifier is seen.

wild-prototype ::= See note, below.

Note. When the name of a prototype is the symbol **?** (question mark), the compiler ignores the prototype name and only matches its signature. In addition, the signature can include the *parameter* identifier.

Namespace

Notes. Z++ namespaces can be derived from one another (multiple-inheritance).

A **namespace**, like a class, can have **private**, **protected** and **public** sections.

The **public** section of a **namespace** is what a **namespace** exports.

The definition and the implementation of a **namespace** can be separated.

namespace-statement ::= *space-definition* # *space-implementation* # *space-using*

space-definition ::= **protected** *namespace-head* *namespace-detail* **namespace** ;

Notes. A **protected namespace** can only be used as a base in a derivation. It cannot be referenced directly.

namespace-head ::= **namespace** *namespace-name*

namespace-detail ::= *derivation* *statements*

Note. Namespace derivation is identical to class [derivation](#).

space-implementation ::= **implementation** *implementation-details* **namespace** ;

Notes. The implementation statement allows separating the definition of a namespace from its implementation, in a manner similar to separating the implementation of a class from its definition.

implementation-details ::= *namespace-identifier* *statements*

Note. Statements inside the implementation of a namespace are usually the implementation (definition) of methods for classes defined in the **namespace**.

space-using ::= *open-namespace* # *close-namespace*

Note. Z++ allows explicit ending of the scope of a namespace.

open-namespace ::= **using namespace** *namespace-scopes* *namespace-entity* ;

Note. This statement is identical to that of C++ for starting the scope of a namespace.

close-namespace ::= **endusing namespace** *namespace-scopes* *namespace-entity* ;

Note. This statement is Z++ specific. It explicitly ends the scope of a namespace.

namespace-scopes ::= *namespace-identifier* :: *namespace-scopes*

namespace-entity ::= *namespace-identifier* # *namespace-identifier* :: *entity*

Note. An entity is an object or function in **public** section of a **namespace**.

Travel statement for Strong Mobility

Notes. Z++ **travel** statement is an abstraction for the notion of strong mobility. An agent is a Z++ component, which includes one or more **travel** statements.

The **travel** statement can appear in any context in an entry point of an agent.

The semantics is that, upon execution of a **travel** statement, the agent terminates itself at the local node. The agent then begins execution at the statement following the **travel** statement, at the destination node.

The state of the agent is transferred to the destination node.

The destination in the following rule is the IP address of the node to reach.

travel-statement ::= **travel** *destination* ;

Notes. The **travel** statement may raise exceptions while sending the agent to its destination. Should an exception occur the agent will not be sent to the designated destination. Instead, the execution of the agent will continue at the local node. This allows catching exceptions resulting from the execution of the **travel** statement.

The states of global objects are recovered to their initial startup state at destination. However, states of objects in an entry point that executed the travel statement are preserved. Thus, if you wish to preserve the state of a global object assign it to a local object before travel, and recover it after travel.

Database SQL Statements

Z++ database statements are confined to Data Manipulation Language (DML) of SQL for interaction with an existing database. Z++ statements resemble their SQL counterparts. However, Z++ statements are object-oriented in nature. Furthermore, Z++ statements handle database identifiers for tables and fields, as well as Z++ objects seamlessly.

Database operations require establishing a session, which includes connecting to the server and logging in. One then needs to end the session. These operations have been abstracted away through declaring a database object. At elaboration a session is established, which terminates when the object goes out of scope.

The Z++ type of database object is defined in the system include file `database.h`. All operations are implemented in the standard static library. However, beyond a plain declaration, there is no need to be aware of the details of the methods that the database system type defines. All methods are used internally by, the Z++ compiler.

In the statements presented in the next section, a database-object refers to an instance of database class in the include file `database.h`, that one declares before making a query. The declarations require standard arguments, as shown below.

```
databaseType Dbase(IP-address, port, database-name);
databaseUserType Duser(Dbase, user-name, password);
```

The first declaration provides the data needed for the second declaration. It is the second declaration of type `databaseUserType`, which the Z++ database statements use. A session begins with the second declaration, as well.

1. The Model

Generally, the purpose of executing the select statement is to receive a set of records from a database. An object-oriented approach to collecting a set of objects is to instantiate a container with the type of such records. Then, the records are inserted into the container using an instance of the record-type by which the container was instantiated. We refer to the container instantiator as catalyst.

The process of insertion into the container involves copying each database record to the catalyst. We refer to this process as mapping a database record to its object image. The mapping is defined by a comma-separated sequence of terms, as shown below.

```
Table-name.Field-name< catalyst-member >
```

The name of table is followed by a dot, and then the name of a field. This is followed by the name of a member, enclosed in $\langle \rangle$. The catalyst-member is the image of mapping. That is, the compiler will copy the field value to the specified member.

2. Database Statements

database-stmt ::= *command* *argument* *mapping* *body* *tail* ;

command ::= **databaseSelect** # **databaseInsert** # **databaseRemove** # **databaseUpdate**

arguments ::= < *database-object* , *catalyst -type* : *table-list* >

database-object ::= *instance of Z++ library class*

catalyst-type ::= *user-defined type for mapping*

table-list ::= *database-table-name* , *database-table-name*

Note. Only select statement can use a list of table names. The other three statements only use a single table, after the colon.

mapping ::= *database-field* < *image* > , *database-field* < *image* >

database-field ::= *table-name* . *field-name*

image ::= *catalyzer-member-name*

body ::= *set-expression* *where-expression*

Notes. Set-expression is permissible for update statement, only. On the other hand, insert statement cannot have a where-expression.

Set and where expressions are similar to their SQL syntax and semantics. In addition, these expressions can include literals, as well as Z++ objects and expressions.

A Z++ expression can be as complex as desired. The only syntactic requirement is to enclose the expression between curly brackets {}.

The logical and relational operators must be those of Z++, which are basically the same as the SQL operators.

tail ::= *select-tail* # *catalyst-object*

Note. Unlike select, the tail for insert, remove and update is just the catalyst.

select-tail ::= *container-object* , *container-method*

Note. Container object is an instance of a template container, such as a list, that the select statement will populate.

Container method is the identifier for the method that select must use in populating the container instance. For overloaded methods and operators, their prototype serves as an identifier.

3. Database Exceptions

The following exceptions could be raised when declaring a database object, i.e. an instance of `databaseUserType` (discussed earlier in the introductory part).

`_EXCEPTION_DATABASE_UnsupportedDatabaseKind`

`_EXCEPTION_DATABASE_LibraryInitializationFailed`
`_EXCEPTION_DATABASE_ConnectionToServerFailed`

The following exceptions can occur when carrying out an SQL statement. The first three exceptions are related to the select statement. The fetch exception could also occur when executing an explicit fetch-statement.

`_EXCEPTION_DATABASE_SelectQueryFailed`
`_EXCEPTION_DATABASE_InsufficientMemoryForQueryResult`
`_EXCEPTION_DATABASE_FetchFailed`

`_EXCEPTION_DATABASE_InsertRequestFailed`
`_EXCEPTION_DATABASE_UpdateRequestFailed`
`_EXCEPTION_DATABASE_RemoveRequestFailed`

Atomic Sequence

atomic-sequence ::= **atomic** *atomic-body* **endatomic** ;

Notes. The purpose of an atomic section is to ensure that a thread performs all the statements in the specified section, without interruption.

When a thread enters an atomic section, no other thread is allowed to run until the thread leaves the atomic section.

An atomic section is a scope, meaning that the objects created within the block will be destroyed when the thread's execution leaves the atomic section.

An atomic section is illustrated below.

```
atomic  
    // body of atomic section  
endatomic;
```

Note. As explained in fundamental types, [mutex](#) is a Z++ fundamental type for constructing critical sections for using shared resources.

Below is an example for quick reference.

```
mutex m; // create a mutex in this scope  
m++; // lock the mutex  
m--; // unlock the mutex
```

Debug Break and Block

Z++ debugging statements in this section are part of the language, not the preprocessor. However, the compiler ignores these statements in **Release** mode, and builds the application without them. There is no need for preprocessor conditional statements, which require setting and unsetting macros, frequently forgotten.

debug-statement ::= *debug-section* # *break-statement*

A Z++ debug section is a **scope** that is only compiled in debug mode. You can create objects in the debug section as you need, and they will be destroyed as execution leaves the debug section.

debug-section ::= **debug** *debug-body* **enddebug** ;

Below is an illustration.

```
debug  
    // debugging code in this scope  
enddebug;
```

The **break** statement does not terminate the execution of a program in debug mode. Instead, when its argument becomes false the execution stops at that point as if there was a breakpoint set by user.

break-statement ::= **break** (*boolean-expression*) ;