Kalimetrix **LOGISCOPE**

RuleChecker

Writing Ada, C++ and Java scriptable rules, metrics and contexts

Before using this information, be sure to read the general information under “Notices” section, on page **165**.

Table of content

Chapter 1 - Introduction 2
Chapter 2 Adding a Rule 4
1. A simple rule: forbidding an operator 4
2. One step further 5
Chapter 3 Adding a Metric 7
Chapter 4 Adding a Context 9
Chapter 5 Formal Descriptions 10
1. Formalism and Abstract Syntax 10
2. Specification of a Context Script 11
3. Specification of a Rule Script 12
4. Specification of a Metric Script 13
5. Interface procedures 15
5.1. Support procedures 15
5.2. Iteration procedures 15
5.3. Access procedures 16
6. Data models 18
6.1. Metal 18
6.1.1 Typographical conventions 18
6.1.2 Introduction 18
6.1.3 Abstract syntax 18
6.1.4 Concrete syntax 19
6.1.5 Tree building functions 19
6.1.6 Example 19
6.2. Ada 22
6.2.1 Abstract and concrete syntax 22
6.3. C++ 89
6.3.1 Abstract syntax 89
6.3.2 Concrete syntax 98
6.4. Java 133
6.4.1 Abstract syntax 133
6.4.2 Concrete syntax 139

Chapter 1 - Introduction

Kalimetrix Logiscope™ *RuleChecker and QualityChecker* allow adding new customized rules, metrics and contexts for verifying Ada, C++, and Java source code.

Logiscope builds a syntactic tree of each source file that is analyzed, based on the grammar of the language. Adding a new rule, metric or context mainly consists in writing a script to explore these trees and recognize structures or elements in the trees that correspond to what should be:

- detected as a violation for a rule,
- should be counted for a metric, or,
- should be stored for a context.

Rules check the source code compliance with respect to some programming standard and may issue violation notices.
Metrics compute numerical properties of the source code.
Contexts gather information from the overall application and may be used to store useful application information to be used in rules and metrics.
Contexts can be added to any type of projects, metrics only to *QualityChecker* projects, and rules only to *RuleChecker* projects.
As the contexts only exist to provide information to specific rules and metrics, their results do not appear as a given results in the Logiscope Studio.

Contexts are calculated over the entire application before rules and metrics are . The overall flow of control during the code analysis is as follow:

```
for all (contexts, standards, metrics)
      "reset": set parameters
end for
for all contexts
      "restart": initialize the context
end for
for all files to be analyzed
      for all contexts
            "compute": add information to the context
      end for
end for
for all contexts
      "free": free the data allocated by the context
end for
for all files to be analyzed
      for all metrics
            "compute": compute the value of the metric
      end for
      for all metrics
            "measureValue": retrieve the value of the metric
      end for
      for all standards
            "compute": check the standard
      end for
      for all (metrics, standards)
            "free": free the data allocated to compute the value or check
            the standard
end for
```

The new customized items: rules, metrics and contexts should be placed in the Logiscope Reference environnement: e.g. in the Ref directory of the Logiscope installation, under the Contexts,

`Metrics` or `Rules` folders, then the appropriate language directory: `C++, Ada, Java`, or `Common` for the items that do not contain any language specific references.
The file extension gives the type of the item:

- **".std"** for a rule,
- **".met"** for a metric,
- **".ctx"** for a context.

As an example, to add a new rule called myrule for checking Java source code, the file should be named myrule.std and placed in the Rules/Java folder.

When the Logiscope project is updated in order to activate the new rules, metrics or contexts, Logiscope checks the items placed in the project language directory and those in Common to check the additions referenced in the project.

The following chapters give simple examples of a rule, a metric and a context. These examples and other basic examples can be found in the standard Logiscope Reference: i.e. in the `Ref` directory of the Logiscope installation directory, under Rules, followed by the appropriate Language directory.

The complete formal descriptions of how the analyzers are designed, of what trees are, and what operators are available can be found under the chapter ***Formal descriptions***. The grammar for each language is also detailed in ***Formal descriptions***.

A `printTree` rule is provided in the `Rule/Common` folder as a utility to print out the syntactic tree in the build window, for an easier comprehension of what operators to use and how to navigate in the tree.The printTree rule prints the tree of the source code of each file, so the output can be quite long. It is recommended to create a project with a single test file containing the structures to be identified, to limit the output to the most useful information for the customization.

Chapter 2 Adding a Rule

1. A simple rule: forbidding an operator

This first rule is an Ada rule that can be found in the Logiscope standard installation, in the `Ref/RulesAda` directory: `noexit.std`.
It shall raise a violation for all exit instructions that are found in the source code. Example:

```
exit;  -- Violation
```

To activate the *noexit* rule checking, the following line should be added to the appropriate Rule Set file(s):

```
STANDARD noexit ON END STANDARD
```

The principal of the rule is to go through all operators of the tree, and to raise a violation when the operator is an `exit` operator.
The entry point is compute, which is the entry point to all rules; the compute parameter tree is the syntactic tree of the source file being parsed. It is necessary to recur on all sons of the initial tree to find all the operators in the code; this is done by filter tree.

```
.NAME No exit

.DESCRIPTION
No exit instructions are allowed

.SEVERITY Advisory

.COMMAND tcl

.CODE

proc compute {standardName tree} {

     set formalism [::logiscope::treeFormalism $tree]

     set opList {}
     ::logiscope::forobj op $formalism formalismOperator {
           lappend opList [::logiscope::opName $op]
     }
     # find the operators in the tree that are exit operators
     filterTree $tree [list opNameIs exit_statement]
}

#-------------------------------------------------------------------
# opNameIs adds a rule violation if the operator of tree is opName
#-------------------------------------------------------------------
proc opNameIs {opName tree} {
     set op [::logiscope::treeOperator $tree]
     if {[string equal [::logiscope::opName $op] $opName]} {
           # the operator has been identified: add a violation
           ::logiscope::addViolation $tree
           return 0
     }
     return 1
}
```

```
#-----------------------------------------------------------------
# filterTree applies callback to tree, and recurs on tree's sons
#-----------------------------------------------------------------
proc filterTree {tree callback} {
     if {[eval $callback [list $tree]]} {
           ::logiscope::forobj son $tree treeChild {
                 filterTree $son $callback
           }
     }
}
```

2. One step further

The following is an Ada rule that can be found in theLogiscope standard installation in the `Ref/RulesAda` directory: `exitlabel.std`.
It raises a violation for all exit instructions that are not followed by a label.

Example:
exit foo; -- Label: ok

```
exit;  -- No label: violation
```

This line should be added to the rule set or configuration file:

```
STANDARD exitlabel ON END STANDARD
```

This rule is based on the previous one: it first detects the exit operators, and then checks if it is followed by a label or not. This is what the interesting part of the syntactic tree of the previous two Ada statements looks like (using the printTree rule):

```
exit_statement
   identifier= " foo
   " void
exit_statement
   void
    voi
     d
```

When a label follows the exit operator, the first son of the exit_statement tree, is an identifier, whereas when there is no label, it is void.
Once the operator has been identified as an exit, the first son (obtained with logiscope::treeDown) is checked: a violation is raised if the operator of the son tree is void.

```
.NAME No exit

.DESCRIPTION
No exit instructions are allowed

.SEVERITY Advisory

.COMMAND tcl

.CODE

proc compute {standardName tree} {
```

```
     set formalism [::logiscope::treeFormalism $tree]

     set opList {}
     ::logiscope::forobj op $formalism formalismOperator {
           lappend opList [::logiscope::opName $op]
     }
     # find the operators in the tree that are exit operators
     # without a label (void operator as a first son of the exit
operator)
     filterTree $tree [list opNamesAre exit_statement void]
}

#-------------------------------------------------------------------
# opNamesAre adds a violation if the tree operator is opName
# and the first son of tree has the operator sonOpName
#-------------------------------------------------------------------
proc opNamesAre {opName sonOpName tree} {
     set op [::logiscope::treeOperator $tree]
     if {[string equal [::logiscope::opName $op] $opName]} {
           return [checkFirstSon $tree $sonOpName]
     }
     return 1
}

#-------------------------------------------------------------------
# opNameIs adds a rule violation if the operator of tree is opName
#-------------------------------------------------------------------
proc opNameIs {opName tree} {
     set op [::logiscope::treeOperator $tree]
     if {[string equal [::logiscope::opName $op] $opName]} {
           # the operator has been identified: add a violation
           ::logiscope::addViolation $tree
           return 0
     }
     return 1
}

#-------------------------------------------------------------------
# checkFirstSon gets the first son of tree and calls
# opNameIs to check for violations on the son
#-------------------------------------------------------------------
proc checkFirstSon {tree opName} {
     set son [logiscope::treeDown $tree 0]
     return [opNameIs $opName $son]
}

#-------------------------------------------------------------------
# filterTree applies callback to tree, and recurs on tree's sons
#-------------------------------------------------------------------
proc filterTree {tree callback} {
     if {[eval $callback [list $tree]]} {
           ::logiscope::forobj son $tree treeChild {
                 filterTree $son $callback
           }
     }
}
```

Chapter 3 Adding a Metric

As an example, this is a metric applicable to Java source code that can be found in the Logiscope standard installation, in the `Ref/Rules/java` directory: `ternary.met`.
It counts the number of use of the ternary operator within a method.

This line should be added to the Metrics Set ".mst" file:

```
METRIC methods ternary ON FORMAT "6" END METRIC
```

```
.DESCRIPTION
This metric computes the number of uses of the ternary operator
?: in the methods.

.COMMAND tcl

.CODE

# The result is stored in the namespace variable "count"
# by the proc "compute".

# vtpTree is the tree for the content of a method
proc compute {metricName vtpTree} {
     filterTree {} $vtpTree
}

# Walk down the abstract syntax tree, counting the
# number of "cond" operators (that is the operator name of
# the ternary operator in the Java data model).
proc filterTree {methodTree tree} {
     set op [::logiscope::treeOperator $tree]
     set opName [::logiscope::opName $op]
     if {"" != $methodTree && "cond" == $opName} {
           variable count
           if {![info exists count($methodTree)]} {
                 set count($methodTree) 0
           }
           incr count($methodTree)
     }
     # Set the tree of the method for the sub-tree, if applicable
     if {"method_decl" == $opName} {
           set currentMethodTree $tree
     } else {
           set currentMethodTree $methodTree
     }
     # Loop through the sub-trees
     ::logiscope::forobj son $tree treeChild {
           filterTree $currentMethodTree $son
     }
}

# Returns the value previously computed
proc measureValue {metricName vtpTree} {
     variable count
     if {[info exists count($vtpTree)]} {
           return $count($vtpTree)
} else {
           return 0
     }
```

```
}

proc metricType {metricName} {
     # This is the default value, anyway.
     return integer
}

proc metricLevel {metricName} {
     # Return the level of this metric.
     return methods
}
```

Chapter 4 Adding a Context

As an example, the following context can be found in the Logiscope standard installation , in the `Ref/Rules/C++` directory: `countFiles.ctx`.

The principal of this context is to increment the count variable at each execution of compute, which is to say for each new source file. The count variable is stored in memory (::logiscope::store) at the context calculation. It will be available for use in metrics and rules, which are all calculated after contexts.

```
.DESCRIPTION
This context computes the number of times it is executed.   The
result is stored as a decimal string in the fileCount index of the
context.

.COMMAND tcl

.CODE

# We only define the compute proc,, to count the number of files,
# The default behavior for the other procs fits our needs.

proc compute {contextName vtpTree} {
     set count 0
     catch {set count [::logiscope::get fileCount]}
     incr count
     ::logiscope::store fileCount $count
 }
```

The context has to be made available for each rule or metric that requires its computation by adding the following statement inside the rule or metric specification file : i.e. ".std" or ".met" files:

```
.REQUIRED <ContextName>
```

As an example, the "*filescope*" rule needs the countFiles context to be activated. So, the file filescope.std shall specify the .REQUIRED statement with the appropriate context:

```
.DESCRIPTION
Description of my rule...

.REQUIRED countFiles

.COMMAND tcl

.CODE
proc compute {standardName tree} {
    set counter [::logiscope::get countFiles fileCount]
    puts "counter is $counter"
}
```

Chapter 5 Formal Descriptions

1. Formalism and Abstract Syntax

A **formalism** is a structure that defines a set of trees. In its most elementary form, a formalism specifies only a set of **operators** which are used as labels to associate a type to each tree belonging to this formalism. A distinction is made between:

- **atomic operators** which are reserved for atomic trees, that is for one-node trees that cannot be broken down in sub-trees, and
- **non-atomic operators** which are reserved for non-atomic trees which can be broken down into sub-trees.

When the emphasis in on the relationship between a non-atomic tree and its sub-trees, the tree is also called a *parent tree* and its sub-trees are called *child trees* or simply *children*. The operator that is labeling a tree is called *head operator* when it is wishable to distinguish it from the operators of the children.
Atom trees have contents, called **atoms**.

The set of the trees of a given formalism is restricted by an **abstract syntax**, in other words a collection of rules that defines the number of children authorized for each tree in accordance with its head operator, and the operators authorized for its children. In addition, an **atom type** must be associated with each atomic operator to specify the class of the values that can be associated with the atomic nodes constituting this operator.

The abstract syntax of a formalism is characterized by the definition of the following elements:

- its *operators*,
- its *phyla*, and
- associations between operators and phyla.

Each *operator* of an abstract syntax is characterized by its *arity*, and the *phylum* to which the operators of its children must belong.
The **arity** of an operator specifies how many children are allowed for the trees labeled with that operator. The arity of atomic operators is 0. As regards non-atomic operators two arity-based categories are defined as
follows:

- **fixed-arity operators** impose a fixed number of children for the trees they are labeling;
- **list operators** do not impose a fixed number of children; they are classified in two sub-categories:
    - list operators that require at least one child;
    - list operators that accept zero or more children.

By extension, the same categories apply to the trees labeled with those operators.

A **phylum** is a set of operators pertaining to the formalism. For fixed-arity operators, one phylum is attached to each child position. For list operators, the same phylum affects all the children.
The position of a child of an operator (or of a tree) is specified by the **rank** of this child in the list of all the
children of that operator (or of that tree), that is by counting children from left to right, from zero. So, the rank of the leftmost child is 0, the rank of the immediately following child is 1, and so on. Negative ranks mean that children are counted from right to left from 1: for a tree with *n* children, both rank *0* and rank *–n* denote the leftmost child, and both rank *n-1* and *-1* denote the rightmost child.

Abstract syntax descriptions are usually created by compiling a description of the formalism written in
Metal Language (see the section Metal.)

2. Specification of a Context Script

The file containing the script of a context shall have the following structure:

```
.DESCRIPTION
Describes the goal of the context.

.COMMAND tcl

.CODE

# This part is verifier specific and optional.
# It extends to the end of the file and contains code
# to be interpreted by the verifier in order to
# accumulate data concerning the application being
# analyzed.

# The code is evaluated in the Tcl namespace bearing
# the name of the context The different proc are
# evaluated by jac et different times, as noted below.

# Except for the proc compute, a default procedure is
# provided that does something sensible when the
# corresponding proc is not defined.

#     for all contexts
#            reset: set parameters
#     end for
#     for all contexts
#            restart: initialize the context
#     end for
#     for all files to be analyzed
#            for all contexts
#                   compute: add information to the
#                                context
#            end for
#     end for
#     for all contexts
#            free: all files have been read:   free
#                   allocated data
#     end for

proc name {contextName} {
     # Returns the name of the context, possibly with the
parameters (see reset below).
     # The default proc returns the name of the context.
}

proc reset {contextName what parameterList} {
     # Set the context parameters, according to the content of the
metrics.cfg file.
     # Returns 0 if the parameters are correct, 1 otherwise.
     # Each context has two sets of parameters : the current one
and the default one.
     # The default procedure maintains two namespace variables
(DefaultParameterList
# and CurrentParameterList) containing the value to
$parameterList, and returns 0.
     # What may be:
     #      RESET_DEFAULT: set both default and current values to
$parameterList.
```

```
     #     RESET_RESET: set current value to default
($parameterList is an empty list)..
     #     RESET_PARAM: set current value to $parameterList.
}

proc restart {contextName} {
     # Initializes the data structures used by compute.
     # The default proc does nothing.
}

proc compute {contextName vtpTree} {
# Accumulate data about the content of vtpTree (syntax tree of one
file)
     # Storage of the data is made by evaluating the proc
::logiscope::store.
     # The result value is ignored.
     # The default proc throws an error : defining this proc is
mandatory.
}

proc free {contextName} {
     # Free the resources allocated during the evaluation of the
proc compute.
     # The default proc does nothing.
}
```

3. Specification of a Rule Script

The file containing the verification script of a rule shall have the following structure:

```
.NAME  Name of the rule

.DESCRIPTION
Describes the goal of the rule.

.PARAMETERS Parameter default values of the rule on a SINGLE line

.SEVERITY  Default level of severity of the rule

.REQUIRED  The context required to verify the rule

.COMMAND   tcl

.CODE

# This part is verifier specific and optional.
# It extends to the end of the file and contains code
# to be interpreted by the verifier in order to find
# violations of the standard.

# The code is evaluated in the Tcl namespace bearing
# the name of the standard The different proc are
# evaluated by jac et different times, as noted below.

# Except for the proc compute, a default procedure is
# provided that does something sensible when the
# corresponding proc is not defined.

# for all standards
#    reset: set parameters
```

```
# end for
# for all files to be analyzed
#     for all standards
#           compute: check the standard
#     end for
#     for all standards
#           free: free allocated data
#     end for
# end for

proc name {standardName} {
     # Returns the name of the standard, possibly with the
parameters (see reset below).
     # The default proc returns the name of the standard.
}

proc reset {standardName what parameterList} {
     # Set the standard parameters, according to the content of
the metrics.cfg file.
     # Returns 0 if the parameters are correct, 1 otherwise.
     # Each standard has two sets of parameters : the current one
and the default one.
     # The default procedure maintains two namespace variables
(DefaultParameterList
# and CurrentParameterList) containing the value to
$parameterList, and returns 0.
     # What may be:
     #     RESET_DEFAULT: set both default and current values to
$parameterList.
     #     RESET_RESET: set current value to default
($parameterList is an empty list)..
     #     RESET_PARAM: set current value to $parameterList.
}

proc compute {standardName vtpTree} {
     # Check the conformance of vtpTree (syntax tree of one file)
     # against the programming standard.
     # Non conformance is noted by evaluating the prog
::logiscope::addViolation.
     # The result value is ignored.
     # The default proc throws an error: defining this proc is
mandatory.
}

proc free {standardName} {
     # Free the resources allocated during the evaluation of the
proc compute.
     # The default proc does nothing.
}
```

4. Specification of a Metric Script

The file containing the script of a metric shall have the following structure:

```
.DESCRIPTION
Describes the goal of the metric.

.REQUIRED  The context required to calculate the metric

.COMMAND tcl

.CODE
```

```
# This part is verifier specific and optional.
# It extends to the end of the file and contains code
# to be interpreted by the verifier in order to compute
# a metric.

# The code is evaluated in the Tcl namespace bearing
# the name of the metric The different proc are
# evaluated by jac et different times, as noted below.

# Except for the proc compute, a default procedure is
# provided that does something sensible when the
# corresponding proc is not defined.

#    for all metrics
#           reset: set parameters
#    end for
#    for all files to be analyzed
#           for all metrics
#                  compute: get the value of the metrics
#           end for
#           for all metrics
#                  measureValue: write the value of the
#                                metrics
#           end for
#           for all metrics
#                  free: free the data allocated to
#                        compute the value
#           end for
#    end for

proc name {metricName} {
     # Returns the name of the metric, possibly with the
parameters (see reset below).
     # The default proc returns the name of the context.
}

proc reset {metricName what parameterList} {
     # Set the metric parameters, according to the content of the
metrics.cfg file.
     # Returns 0 if the parameters are correct, 1 otherwise.
     # Each metric has two sets of parameters : the current one
and the default one.
     # The default procedure maintains two namespace variables
(DefaultParameterList
# and CurrentParameterList) containing the value to
$parameterList, and returns 0.
     # What may be:
     #     RESET_DEFAULT: set both default and current values to
$parameterList.
     #     RESET_RESET: set current value to default
($parameterList is an empty list)..
     #     RESET_PARAM: set current value to $parameterList.
}

proc compute {metricName vtpTree} {
# Compute the value of the metric and stores it for later
retrieval
# by a call to measureValue or check by checkBounds.
     # The result value is ignored.
     # The default proc throws an error : defining this proc is
mandatory.
}
```

```
proc measureValue {metricName vtpTree} {
     # Return the value of the metric
     # The default proc throws an error : defining this proc is
mandatory.
}

proc free {metricName} {
     # Free the resources allocated during the evaluation of the
proc compute.
     # The default proc does nothing.
}

proc metricType {metricName} {
     # Return a string describing the type of the value of this
metric.
     # Allowable types are : integer, number, string.
     # The default proc returns integer.
}

proc metricLevel {metricName} {
     # Return the level of this metric.
     # Allowable levels are dependent on the language:
     # - C++: module, functions, classes, application.
     # - Ada:  module, functions.
     # - Java: module, methods, classes.
     # The default proc returns "module".
}
```

5. Interface procedures

A Tcl namespace is always available during rule checking. This namespaces is named "logiscope" and contains support procedures and access functions to query the abstract syntax tree representing the code in the source files.

Support procedures

- `addViolation standardName vtpTree`: adds a violation notification for the specified standard. The violation is located at the file and line specified by the `vtpTree` node of the syntax tree.
- `reset standardOrMetricOrContextName what parameterList`: manages the default behavior of the `reset` procs.
- `store index value`: stores `value` in the current context at index `index`; usable only in a CONTEXT script.
- `get ?contextName? index`: returns the value previously stored at index `index` in the context named `contextName` (the current context if the argument is not specified; in this case, it may only be used in a CONTEXT script). It is an error if the context or the index do not exist.
- `forobj varName container iteratorName script`: implements a loop where the loop variable `varName` takes on values from the list `iteratorName` of `container`. The `script` argument is a Tcl script that is evaluated for each element of the list.

Iteration procedures

Note: see the section Metal for an explanation of the different concepts used by these procedures (formalisms, operators, trees, etc).

Every iterator defines four procedures named after the name of the iterator:

- *`iterator`*`_Start containerInstance`: returns an iteractorObject to be used by the following procedure.
- *`iterator`*`_End iteratorObject`: returns true (1) if there remains elements to be returned by the `iteratorObject`.

- *iterator*`_Get iteratorObject`: returns the next element of the list, advancing the `iteratorObject` to the next position in the list.
- *iterator*`_Done iteratorObject`: frees the `iteratorObject`.

Iteration procedures are used in the following manner:

```
set iterator [logiscope::treeChild_Start $vtpTree]
while {![logiscope::treeChild_End $iterator] } {
  set child [logiscope::treeChild_Get $iterator]
  ...
}
logiscope::treeChild_Done $iterator
```

(see also the `forobj` utility procedure in section Support procedures).

Available iterators are:

- `treeChild: container: vtpTree, elements: Tree`: iterate over the child nodes of the tree, from left to right.
- `treeRChild: container: vtpTree, elements: Tree`: iterate over the child nodes of the tree, from right to left.
- `treeAnnot: container:  Tree, elements: Tree`: iterate over the annotation nodes of the tree.
- `phylumOperator: container Phylum, elements: Annot`: iterate over the operators of the phylum.
- `formalismPhylum: container: Formalism, elements: Phylum`: iterate over the phyla of the formalism.
- `formalismFrame: container: Formalism, elements: Frame`: iterate over the frames of the formalism.
- `formalismOperator: container: Formalism, elements: Operator`: iterate over the operators of the formalism.

Access procedures

Note: see the section Metal for an explanation of the different concepts used by these procedures (formalisms, operators, trees, etc).

- `treeUp: tree: Tree ► Tree`: returns the parent tree of `tree`.
- `treeRoot: tree: Tree ► Tree`: returns the highest tree above `tree`, i.e. the root of the tree to which `tree` belongs.
- `treeDown: tree: Tree, rank: int ► Tree`: returns the child tree of `tree` that is at position `rank` in the list of children (ranks are counted starting at 0).
- `treeRight: tree: Tree ► Tree`: returns the right sibling of `tree` in the list of children of the parent tree of `tree`.
- `treeLeft: tree: Tree ► Tree`: returns the left sibling of `tree` in the list of children of the parent tree of `tree`.
- `treeParent: tree: Tree ► Tree`: returns the parent tree of `tree`.
- `treeFormalism: tree: Tree ► Formalism`: returns the formalism to which `tree belongs`.
- `treeOperator: tree: Tree ► Operator`: returns the operator labeling `tree`.
- `treePhylum: tree: Tree ► Phylum`: returns the phylum associated with `tree`, if it is a child tree.
- `treeGetAnnotValue: tree: Tree, frame: Frame ► Atom`: returns the value of the annotation defined in `frame` and hung to `tree`.
- `treeGetAnnot: tree: Tree, frame: Frame ► Annot`: returns the annotation defined in `frame` and hung to `tree`.
- `treeLength: tree: Tree ► int`: returns the count of children of `tree` (0 if `tree` is atomic).
- `treeRank: tree: Tree ► int`: returns the rank of `tree` in the list of children of the parent tree of `tree`. Returns -1 if `tree` has no parent.
- `treeAtomType: tree: Tree ► AtomType`: returns the atom type that is associated with the operator of `tree`, or an empty string if `tree` is not atomic.
- `treeAtomValue: tree: Tree ► Atom`: returns the atom associated with `tree`.

- `treeEqual: tree1: Tree, tree2: Tree ► int:` Returns a true value when the trees `tree1` and `tree2` are equal, otherwise a false value. Two trees are considered equal either if they are the same tree or they are agreeing with each other in every details:
    - same head operators for both trees;
    - same count of children and same child operators at every level in the trees;
        - atomic children are at the same places and have equal atoms;
    - the annotations that are hung on the trees and pertain to a frame with the *equal* control set are the same: they are hung at the same places and have equal atoms.
- `treeIsParent: parent: Tree, son: Tree ► int`: returns true if `son` is a child of `parent`; false otherwise.
- `annotFrame: annot: Annot ► Frame`: returns the frame describing the legal values for `annot`.
- `annotValue: annot: Annot ► Atom`: returns the value of `annot`.
- `annotType: annot: Annot ► AtomType`: returns the atom type describing the legal values for `annot`.
- `atomTypeName: atomType: AtomType ► string`: returns the type of `atomType`.
- `atomEqual: atom1: Atom, atomType: AtomType, atom2: Atom ► int:` returns true if `atom1` and `atom2` are equal, as appropriate for `atomType`.
- `atomToString: atom: Atom, atomType: AtomType ► string`: returns a string representing the value of `atom`, as appropriate for `atomType`.
- `integerValue: atom: Atom ► int`: returns the value of `atom` as an integer (the atom type must be `integer`).
- `stringValue: atom: Atom ► string`: returns the value of `atom` as a string (the atom type must be `string`).
- `nameString: atom: Atom ► string`: returns the value of `atom` as a string (the atom type must be `name`).
- `nameValue: atom: Atom ► string`: returns the value of `atom` as an identifier (the atom type must be `name`).
- `atomTypeByName: name: string ► AtomType:` returns the atom type identified by `name`.
- `formName: form: Formalism ► string`: returns the name of `form`.
- `formVersion: form: Formalism ► int`: returns the version of `form`.
- `phylumName: phyl: Phylum ► string`: returns the name of `phyl`.
- `phylumFormalism: phyl: Phylum ► Formalism`: returns the formalism to which `phyl` belongs.
- `phylumByName: name: string, form: Formalism ► Phylum`: returns the phylum identified by `name` in the formalism `form`.
- `phylumHasOperator: phyl: Phylum, op: Operator ► int`: returns true (1) if `op` is a member of `phyl`; false (0) otherwise.
- `opName: op: Operator ► string`: returns the name of the operator `op`.
- `opFormalism: op: Operator ► Formalism`: returns the formalism to which `op` belongs.
- `opArity: op: Operator ► int:` returns the arity of the operator `op`. The operator must be a fixed arity operator or an atomic operator.
- `opIsAtom: op: Operator ► int`: returns true (1) if `op` is an atomic operator; false (0) otherwise.
- `opIsList0: op: Operator ► int`: returns true (1) if `op` is an operator with a varying number of arguments, 0 or more; false (0) otherwise.
- `opIsList1: op: Operator ► int`: returns true (1) if `op` is an operator with a varying number of arguments, 1 or more; false (0) otherwise.
- `opIsList: op: Operator ► int`: returns true (1) if `op` is an operator with a fixed number of arguments; false (0) otherwise.
- `opIsFixArity: op: Operator ► int`: returns true (1) if `op` is an atomic operator; false (0) otherwise.
- `opAtomType: op: Operator ► AtomType`: returns the atom type associated with `op`.
- `opSonsPhylum: op: Operator ► Phylum:` returns the phylum describing the legal operators for every child of `op` (`op` must be a list operator).
- `opNthPhylum: op: Operator, n: int ► Phylum`: returns the phylum describing the legal operators for the `n`th child of the fixed arity operator `op` (`n` is counted starting from 0).
- `operatorByName: name: string, form: Formalism ► Operator`: returns the operator identified by `name` in the formalism `form`.
- `frameName: frame: Frame ► string:` returns the name of the frame.
- `framePhylum: frame: Frame ► Phylum:` returns the phylum associated with `frame`

- frameByName: name: string, form: Formalism ► Frame: returns the frame identified by name in the formalism form.
- frameAtomType: frame: Frame ► AtomType: returns the atom type that describes the legal annotations for frame.

Some helper access functions:

- fileName ► string: returns the name of the current file, without the path.
- FileFullName ► string: returns the name of the current file, with the path.
- Application ► string: returns the name of the application directory, with the path.
- FirstFile ► string: returns the name of the first file of the application to be analyzed, without the path.
- LastFile ► string: returns the name of the last file of the application to be analyzed, without the path.
- treeGetStartLine tree: Tree ► int: returns the line that the tree starts on
- treeGetEndLine tree: Tree ► int: returns the line that the tree ends on
- treeGetStartChar tree: Tree ► int: returns the number of the character (starting from the beginning of the file) that the tree starts on
- treeGetEndChar tree: Tree ► int: returns the number of the character (starting from the beginning of the file) that the tree ends on

6. Data models

1. Metal

Typographical conventions

•The description of Metal grammatical constructs uses a syntax notation that is similar to Extended Backus-Naur Format *(EBN*F). The symbols used in this format and their meaning are indicated in the table below.

Symbol	**Meaning**
::=	is defined to be
|	alternatively
<text>	non-terminal
"text"	literal
<<text>>	textual description
*	the preceding syntactic unit can be repeated zero or more times
+	the preceding syntactic unit can be repeated one or more times
{}	the enclosed syntactic units are grouped as a single syntactic unit
[]	the enclosed syntactic unit is optional -may occur zero or one time

Introduction

Metal is a language to specify formalisms. A typical Metal specification for a formalism is characterized by the definition of

- an abstract syntax ,
- a concrete syntax, ,
- and, to link them, tree building functions.

With the Metal language, programmers have a convenient way to define the abstract syntax.
It is possible either to group the definition of the abstract syntax and the definition of the concrete syntax and of the tree building functions, in one Metal program, or to separate them in two Metal programs.

Abstract syntax

A document in a formalism is represented by a tree. Each node of this tree corresponds to a syntactic construction of the formalism.
The abstract syntax specifies the set of trees that are considered as abstract representations of well-formed documents in the formalism.
Abstract syntaxes are characterized by the definition of operators and phyla, and possibly of frames.

Operators are used as labels on the nodes of trees that are pertaining to the formalism. They denote the type of the nodes.
A phylum is a set of operators pertaining to the formalism.
An operator can be:

- atomic: nodes labeled with such an operator do not have children but contain a value of a given type;
- or non-atomic: nodes labeled with such an operator may have children nodes.

Each non-atomic operator is characterized:

- by its arity, that is the number of children nodes permitted to the nodes it labels, and
- by the phyla to which the operators of their children must belong.

Frames are used to hang ancillary data onto the nodes of the abstract trees. Frames are characterized by the nature of the data and by controls on the usage of these data.

Concrete syntax

The concrete syntax is specified by rules written in a modified BNF (*Backus-Naur Forma*t) format where non-terminals are written between angle brackets and terminals are enclosed in double quotes or in single quotes according to their types.

Tree building functions

The tree building functions specify the translation from concrete syntax to abstract syntax.
A tree building function is attached to each concrete syntax rule.

Example

To start with an example, consider the following excerpts of dictionary articles:

programme, n. & v.t.
1. Descriptive notice of series of events; definite plan of intended proceedings;
(colloq.) "what is the ~ for today?" what are we going to do today? ;
"~ picture" ....
2. v.t. Make a ~or definite plan of.
[f. LL f. Gk programma ...]
analyze (-z), v.t. Examine minutely the constitution of;
...
(gram.) resolve (sentence)into its grammatical elements.
Hence ~ABLE a.
[f. F analyzer (analyze, as foll.) ...]
user[1], n. See USE[2].
user[2] (-z-), n. (law). Continued use ...

Such entries can be thought of as consisting of:

- the word being defined;
- aphonetic respelling, if needed;
- one or more grammatical types;
- one or more definitions, possibly split in sub-definitions;
- one or more derivatives of the word;
- etymology if needed.

When there are several articles for the same word, a reference number is set on the word to distinguish them.
The example below illustrates how this specification can be turned into a formalism by the way of the Metal language.

```
definition of DICTIONARY version 1 is
abstract syntax
article -> ENTRY DEFINITIONS DERIVATIVES ETYMOLOGY ; --(O1)
ENTRY := entry ;
entry -> WORD PHONETIC TYPES ; --(O1)
WORD := word ; --(P1)
PHONETIC := string void; --(P1)
TYPES := types; --(P1)
types -> TYPE+; --(O2a)
TYPE := type; --(P1)
```

```
DEFINITIONS := definitions; --(P1)
definitions -> DEFINITION+; --(O2a)
DEFINITION := string subdefinitions; --
(P2) subdefinitions -> SUBDEFINITION+; --
(O2a) SUBDEFINITION := string;
DERIVATIVES := derivatives; --
(P1) derivatives -> DERIVATIVE*;
--(O2b) DERIVATIVE := string; --
(P1)
ETYMOLOGY := string void; --(P1)
word -> implemented as name; --
(O3)
string -> implemented as string; --(O3)
type -> implemented as number; --(O3)
void -> implemented as void; --(O3)
frames
article_nr -> implemented as integer; --
(F1) end definition
```

This is a complete Metal program that defines the version 1 of a VTP formalism named *DICTIONARY* as stated in the beginning of the program:

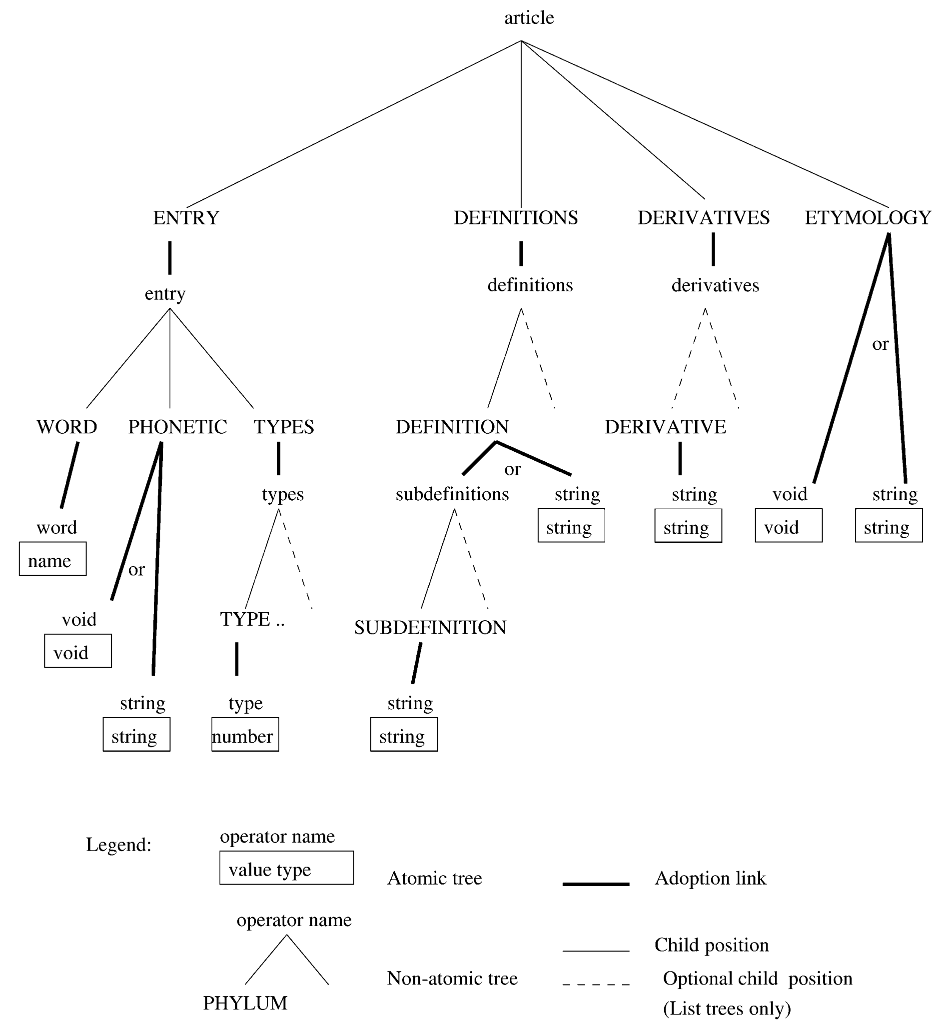
```
definition of DICTIONARY version 1 is
```

This definition is made of:

- comments which start with "--" (double minus) and terminate at the end of the line;
- several kinds of statements with "->" in a section starting with `abstract syntax`: they are operator definitions;
- two kinds of statements with ":=" in the *abstract syntax* section : they are phylum definitions;
- and a statement with "->" in a section starting with *frame*s: it is a frame definition.

Note: in the example, phylum names are in uppercase whereas operator names and frame names are in lowercase. This is done only to help readers make a distinction between operators and phyla.

An abstract syntax for a formalism defines a complete and consistent set of abstract trees that reflects the semantics of that formalism.
The abstract syntax for *DICTIONARY* does not differentiate numbered definitions from single unnumbered definitions because the difference lays in presentation — it is not desirable to number a unique element —, not in semantics.


article
ENTRY
DEFINITIONS
DERIVATIVES
ETYMOLOGY
entry
definitions
derivatives
or
WORD
PHONETIC
TYPES
DEFINITION
DERIVATIVE
or
word
name
types
subdefinitions
string
string
string
string
void
void
string
string
or
void
void
TYPE ..
SUBDEFINITION
string
string
type
number
string
string
Legend:
operator name
value type
Atomic tree
Adoption link
operator name
PHYLUM
Non-atomic tree
Child position
Optional child position
(List trees only)

2. Ada

Abstract and concrete syntax

```
definition of ADA version 4 is

%[LEFT 'AND', 'OR', 'XOR']%
%[LEFT "=", "/=", "<", "<=", ">", ">=", 'IN']%
%[LEFT "+", "-", "&"]%
%[RIGHT %UNARY]%
%[LEFT "*", "/", 'MOD', 'REM']%
%[LEFT "**", 'ABS', 'NOT']%

chapter AXIOME

rules

    program_in_the_ada_language := compilation_unit_list;
        compilation_unit_list;

    program_in_the_ada_language := phylum;
        phylum;

end chapter;

chapter TOKENS

rules

    numeric_literal := %NUMLITERAL;
        :numeric_literal[%NUMLITERAL];

    character_literal := %CHARLITERAL;
        :character_literal[%CHARLITERAL];

    character_string := %CHARSTRING;
        :string_literal[%CHARSTRING];

    identifier := %IDENTIFIER;
        :identifier[%IDENTIFIER];

    identifier := meta;
        meta;

    label := "<<" designator ">>";
        designator;

    operator := %OPERATOR;
        :operator[%OPERATOR];

    c_designator := compound_name;
        compound_name;

    c_designator := operator;
        operator;

    designator := identifier;
```

```
        identifier;

    designator := operator;
        operator;

    meta := %META;
        :meta[%META];

end chapter;

chapter ' 3 : DECLARATIONS AND TYPES '

chapter ' 3.1 DECLARATIONS '

rules

    declaration := object_declaration;
        object_declaration;

    declaration := number_declaration;
        number_declaration;

    declaration := type_declaration;
        type_declaration;

    declaration := subtype_declaration;
        subtype_declaration;

    declaration :=
        subprogram_declaration;
        subprogram_declaration;

    declaration := package_declaration;
        package_declaration;

    declaration := task_declaration;
        task_declaration;

    declaration := protected_declaration;
        protected_declaration;

    declaration := generic_declaration;
        generic_declaration;

    declaration := exception_declaration;
        exception_declaration;

    declaration :=
        generic_instantiation_decl;
        generic_instantiation_decl;

    declaration := renaming_declaration;
        renaming_declaration;

    declaration := unit_renaming_declaration;
        unit_renaming_declaration;
```

```
declaration :=
    pragma; pragma;
```

```
end chapter;

chapter ' 3.2 TYPES AND SUBTYPES '

chapter ' 3.2.1 TYPE DECLARATIONS '

rules

    type_declaration :=
        full_type_declaration;
        full_type_declaration;

    type_declaration := incomplete_type_declaration;
        incomplete_type_declaration;

    type_declaration := private_type_declaration;
        private_type_declaration;

    type_declaration :=
        private_extension_declaration;
        private_extension_declaration;

    full_type_declaration :=
        'TYPE' identifier known_discriminant_part_option_is
type_definition;
        :type_declaration<identifier,
        known_discriminant_part_option_is,
                             type_definition>;

    known_discriminant_part_option_is := 'IS';
        :discriminant_part<>;

    known_discriminant_part_option_is := discriminant_part
        'IS'; discriminant_part;

    discriminant_part_option_is := 'IS';
        :discriminant_part<>;

    discriminant_part_option_is := "(" "<>" ")" 'IS';
        :unknown_discriminant;

    discriminant_part_option_is := discriminant_part 'IS';
        discriminant_part;

    type_definition := enumeration_type_definition;
        enumeration_type_definition;

    type_definition := integer_type_definition;
        integer_type_definition;

    type_definition := real_type_definition;
        real_type_definition;

    type_definition :=
        array_type_definition;
```

```
    array_type_definition;

type_definition :=
    record_type_definition;
    record_type_definition;

type_definition := access_type_definition;
    access_type_definition;
```

```
    type_definition :=
        derived_type_definition;
        derived_type_definition;

end chapter;

chapter ' 3.2.2 SUBTYPE DECLARATIONS '

rules

    subtype_declaration := 'SUBTYPE' identifier 'IS'
    subtype_indication;
        :subtype_declaration<identifier, subtype_indication>;

    subtype_indication :=
        ambig_subtype_entry_subprogram_call_statement;
        ambig_subtype_entry_subprogram_call_statement;

    ambig_subtype_entry_subprogram_call_statement :=
        ambig_entry_subprogram_call_statement;
        %{
      VTP_TreeP tree = Parser_Pop();
      VTP_TreeP tr = VTP_TreeDown(tree,
      0);
      if (VTP_TreeLength(VTP_TreeDown(tree, 1)) >
          0) { tr =
           TreeMake2(subtype_indication, Ada_Disown(tree, 0),
               TreeRename(Ada_Disown(tree, 1),
                                   record_or_array_aggregate));
      } else if (CheckOper(tr, slice))
          { tr =
           TreeMake2(subtype_indication, Ada_Disown(tr, 0),
               TreeMake1(index_constraint, Ada_Disown(tr, 1)));
      } else {
          tr =
           TreeMake2(subtype_indication, Ada_Disown(tree, 0),
               TreeMake0(void, tree));
        }
        Parser_SetCoordNN(tr, tree, tree);
      Parser_Push(tr);
      VTP_TreeDestroy(tree);
      }%;

    subtype_indication := name constraint;
        :subtype_indication<name, constraint>;

    constraint := range_constraint;
        range_constraint;

    constraint := floating_point_constraint;
        floating_point_constraint;

    constraint :=
        fixed_point_constraint;
```

```
        fixed_point_constraint;

    constraint := index_constraint;
        index_constraint;

end chapter;

end chapter;
```

```
chapter ' 3.3 OBJECT AND NAMED NUMBERS '

rules

    object_declaration :=
        identifier ":" qualifier_option subtype_indication
                                initialization_option;
        :object_declaration<:identifiers_list<identifi
er>, qualifier_option,
                        subtype_indication, initialization_option>;

    object_declaration :=
        two_identifier_list ":" qualifier_option
        subtype_indication
                                initialization_option;
        :object_declaration<two_identifier_list,
                       qualifier_option, subtype_indication,
                       initialization_option>;

    object_declaration :=
        identifier ":" qualifier_option array_type_definition
                                initialization_option;

:object_declaration<:identifiers_list<identifier>,qualifier_option
,
                        array_type_definition,initialization_optio
                        n>;

    object_declaration :=
        two_identifier_list ":" qualifier_option
            array_type_definition initialization_option;
        :object_declaration<two_identifier_list,
            qualifier_option,
            array_type_definition,initialization_option>;

    qualifier_option :=;
        :void;

    qualifier_option := 'ALIASED';
        :aliased;

    qualifier_option := 'CONSTANT';
        :constant;

    qualifier_option := 'ALIASED' 'CONSTANT';
        :aliased_constant;

    initialization_option :=;
        :void;

    initialization_option := ":="
        expression; expression;

    number_declaration := identifier ":" 'CONSTANT' ":="
```

```
    expression;
        :number_declaration<:identifiers_list<identifier>,
        expression>;

    number_declaration := two_identifier_list ":" 'CONSTANT'
":=" expression;
        :number_declaration<two_identifier_list, expression>;

end chapter;
```

```
chapter ' 3.4 DERIVED TYPES AND CLASSES '

rules

    derived_type_definition := abstract_option 'NEW'
            subtype_indication record_extension_option;
        :derived_type<abstract_option,
            subtype_indication,
            record_extension_option>;

    abstract_option := ;
        :void;

    abstract_option := 'ABSTRACT';
        :'abstract';

    record_extension_option := ;
        :void;

    record_extension_option := 'WITH' record_definition;
        record_definition;

end chapter;

chapter ' 3.5 SCALAR TYPES '

rules

    range_constraint_option :=;
        :void;

    range_constraint_option :=
        range_constraint; range_constraint;

    range_constraint := 'RANGE'
        range; range;

    range := range1;
        range1;

    range := attribute;
        attribute;

    range1 := simple_expr ".." simple_expr;
        :range<simple_expr.0, simple_expr.1>;

chapter ' 3.5.1 ENUMERATION TYPES '

rules

    enumeration_type_definition :=
        "(" enumeration_literal_specification_list ")";
        enumeration_literal_specification_list;

    enumeration_literal_specification_list :=
enumeration_literal_specification;
```

:enumeration_type_definition<enumeration_literal_specification>; enumeration_literal_specification_list :=

```
        enumeration_literal_specification_list ","
            enumeration_literal_specification;
        enumeration_literal_specification_list:<..,
            enumeration_literal_specification>;


    enumeration_literal_specification := identifier;
        identifier;

    enumeration_literal_specification := character_literal;
        character_literal;

end chapter;

chapter ' 3.5.4 INTEGER TYPES '

rules

    integer_type_definition := range_constraint;
        :integer_type<range_constraint>;

    integer_type_definition := 'MOD' expression;
        :integer_type<expression

>; end chapter;

chapter ' 3.5.6 REAL TYPES

' rules

    real_type_definition :=
    floating_point_constraint;
        floating_point_constraint;

    real_type_definition := fixed_point_constraint;
        fixed_point_constraint;

    real_type_definition := decimal_fixed_point_constraint;
        decimal_fixed_point_constraint;

    floating_point_constraint := 'DIGITS' expression
range_constraint_option;
        :floating_point_constraint<expression,
        range_constraint_option>;

    fixed_point_constraint := 'DELTA' expression
    range_constraint_option;
        :fixed_point_constraint<expression,
        range_constraint_option>;

    decimal_fixed_point_constraint := 'DELTA' expression
'DIGITS' expression
                range_constraint_option;
        :decimal_fixed_point_constraint<expression.0, expression.1,
                range_constraint_option>;
```

```
end chapter;

end chapter;

chapter ' 3.6 ARRAY TYPES '
```

```
rules

    array_type_definition :=
        unconstrained_array_definition;
        unconstrained_array_definition;

    array_type_definition := constrained_array_definition;
        constrained_array_definition;

    unconstrained_array_definition :=
        'ARRAY' "(" index_subtype_definition_list ")" 'OF'
subtype_indication;
        :array<index_subtype_definition_list, :void,
        subtype_indication>;

    unconstrained_array_definition :=
        'ARRAY' "(" index_subtype_definition_list ")" 'OF' 'ALIASED'
                 subtype_indication;
        :array<index_subtype_definition_list, :aliased,
subtype_indication>;

    constrained_array_definition :=
        'ARRAY' "(" gen_discrete_range_list ")" 'OF'
        subtype_indication;
        :array<gen_discrete_range_list, :void, subtype_indication>;

    constrained_array_definition :=
        'ARRAY' "(" gen_discrete_range_list ")" 'OF' 'ALIASED'
                 subtype_indication;
        :array<gen_discrete_range_list, :aliased,
        subtype_indication>;

    gen_discrete_range_list := discrete_range;
        :index_definitions_list<discrete_range>;

    gen_discrete_range_list := gen_discrete_range_list ","
discrete_range;
        gen_discrete_range_list:<.., discrete_range>;

    index_subtype_definition_list := index;
        :index_definitions_list<index>;

    index_subtype_definition_list :=
index_subtype_definition_list "," index;
        index_subtype_definition_list:<.., index>;

    index := name 'RANGE' "<>";
        :index_subtype_definition<name>;

    discrete_range := name;
        %{
     VTP_TreeP tree = Parser_Pop();
     VTP_TreeP tr;
     if (CheckOper(tree, identifier) ||
         CheckOper(tree, selected_component) ||
```

```
    CheckOper(tree, indexed_component)) {
  Parser_Push(TreeMake2(subtype_indication, tree,
                 TreeMake0(void, tree)));
} else if (CheckOper(tree, slice)) {
  tr = (TreeMake2(subtype_indication, Ada_Disown(tree, 0),
             Ada_Disown(tree, 1)));
    Parser_SetCoordNN(tr, tree, tree);
```

```
        Parser_Push(tr);
      VTP_TreeDestroy(tree);
    } else {
      Parser_Push(tree);
    }
    }%;

  discrete_range := range_denotation;
      range_denotation;

  range_denotation := name range_constraint;
      :subtype_indication<name, range_constraint>;

  range_denotation := range1;
      range1;

  index_constraint := "(" discrete_range_list ")";
      discrete_range_list;

  discrete_range_list := range_denotation "," range_denotation;
      :index_constraint<range_denotation.0, range_denotation.1>;

  discrete_range_list := range_denotation "," name;
      %{
      VTP_TreeP tr_name = Parser_Pop();
      VTP_TreeP tr_range = Parser_Pop();
      VTP_TreeP tr;
      if (CheckOper(tr_name,identifier) ||
        CheckOper(tr_name, selected_component)
        || CheckOper(tr_name,
        indexed_component)) { tr =
        TreeMake2(index_constraint,tr_range,
                          TreeMake2(subtype_indication,tr_name,
                                       TreeMake0(void,tr_name)));
      } else if (CheckOper(tr_name,slice)) {
        tr = TreeMake2(index_constraint,tr_range,
                TreeMake2(subtype_indication,
                Ada_Disown(tr_name, 0),
                    Ada_Disown(tr_name, 1)));
      } else {
        tr = TreeMake2(index_constraint,tr_range,tr_name);
      }
      Parser_PopUntilToken($2);
    Parser_Push(tr);
    }%;

  discrete_range_list := expression_list "," range_denotation;
      %{
      VTP_TreeP tr_range = Parser_Pop();
      VTP_TreeP tr_list = Parser_Pop();
      VTP_TreeP tr;
      tr = PostRename(tr_range, tr_list, index_constraint);
      Parser_PopUntilToken($2);
    Parser_Push(tr);
    }% ;
```

```
    discrete_range_list := discrete_range_list "," discrete_range;
        discrete_range_list:<.., discrete_range>;

end chapter;
```

```
chapter ' 3.8 RECORD TYPES

' rules

    discriminant_part := "(" discriminant_specification_list ")";
        discriminant_specification_list;

    discriminant_specification_list := discriminant_specification;
        :discriminant_part<discriminant_specification>;

    discriminant_specification_list :=
        discriminant_specification_list ";"
        discriminant_specification;
        discriminant_specification_list:<..,
        discriminant_specification>;

    discriminant_specification :=
        identifier_list ":" name initialization_option;
        :object_declaration<identifier_list, :void,
              :subtype_indication<name,
              :void>,initialization_option>;

    discriminant_specification :=
        identifier_list ":" 'ACCESS' name initialization_option;
        :object_declaration<identifier_list, :access,
              :subtype_indication<name,
              :void>,initialization_option>;

rules

    record_type_definition := tagged_option record_definition;
        :record_type<tagged_option, :void, record_definition>;

    record_type_definition := tagged_option 'LIMITED'
    record_definition;
        :record_type<tagged_option, :limited, record_definition>;

    record_definition := 'NULL' 'RECORD';
        :null_record;

    record_definition := 'RECORD' component_list 'END' 'RECORD';
        component_list;

    component_list := component;
        :components_list<component>;

    component_list := pragma ";";
        :components_list<pragma>;

    component_list := component_list pragma ";";
        component_list:<.., pragma>;

    component_list := component_list
        component; component_list:<..,
```

```
    component>;

component := variant_part;
    variant_part;

component := representation_clause ";";
    representation_clause;

component := 'NULL' ";";
```

```
        :null_component;

    component :=
        identifier ":" subtype_indication initialization_option ";";
        :object_declaration<:identifiers_list<identifier>, :void,
              subtype_indication,initialization_option>;

    component :=
        two_identifier_list ":" subtype_indication
";"     initialization_option
;
        :object_declaration<two_identifier_list, :void,
subtype_indicatio
n,                              initialization_option>;

    component :=
        identifier ":" 'ALIASED' subtype_indication
";"     initialization_option
;
        :object_declaration<:identifiers_list<identifier>, :aliased,
              subtype_indication,initialization_option>;

    component :=
        two_identifier_list ":" 'ALIASED' subtype_indication
                                initialization_option ";";
        :object_declaration<two_identifier_list, :aliased,
subtype_indication,
                                initialization_option>;

    variant_part := 'CASE' designator 'IS'
pragma_option_list variant_list
                   'END' 'CASE' ";";
        :variant_part<designator, pragma_option_list, variant_list>;

    variant_list := variant;
        :variants_list<variant>;

    variant_list := variant_list variant;
        variant_list:<.., variant>;

    variant := 'WHEN' choice_list "=>"
    component_list;
        :variant<choice_list, component_list>;

    choice_list := choice;
        :choices_list<choice>;

    choice_list := choice_list "|" choice;
        choice_list:<.., choice>;

    choice := expression;
        expression;

    choice := range_denotation;
        range_denotation;
```

```
    choice := 'OTHERS';
        :others;

end chapter;
```

```
chapter ' 3.10 ACCESS TYPES '

rules

    access_type_definition := 'ACCESS' access_to_object_option
            subtype_indication;
        :access_to_object_type<access_to_object_opt
ion, subtype_indication>;

    access_type_definition := 'ACCESS' access_to_subprogram_option
'PROCEDURE'
            formal_part_option;

        :access_to_subprogram_type<access_to_subprogram_option,
            formal_part_option, :void>;

    access_type_definition := 'ACCESS' access_to_subprogram_option
'FUNCTION'
            fct_formal_part_option 'RETURN' name;
        :access_to_subprogram_type<access_to_subprogram_option,
            fct_formal_part_option, name>;

    access_to_object_option := ;
        :void;

    access_to_object_option := 'ALL';
        :all;

    access_to_object_option := 'CONSTANT';
        :constant;

    access_to_subprogram_option := ;
        :void;

    access_to_subprogram_option := 'PROTECTED';
        :protected;

    incomplete_type_declaration := 'TYPE' identifier
    discriminant_part;
        :type_declaration<identifier, discriminant_part, :void>;

    incomplete_type_declaration := 'TYPE' identifier "(" "<>" ")";
        :type_declaration<identifier, :unknown_discriminant,
        :void>;

    incomplete_type_declaration := 'TYPE' identifier;
        :type_declaration<identifier, :discriminant_part<>,

:void>; end chapter;

rules

    private_type_definition := tagged_option 'PRIVATE';
        :private_type<tagged_option, :void>;

    private_type_definition := tagged_option 'LIMITED' 'PRIVATE';
```

```
    :private_type<tagged_option, :limited>;

tagged_option := ;
    :void;
```

```
    tagged_option := 'TAGGED';
        :tagged;

    tagged_option := 'ABSTRACT' 'TAGGED';
        :abstract_tagged;

    private_extension_definition := abstract_option
'NEW' subtype_indication
            'WITH' 'PRIVATE';
        :derived_type<abstract_option, subtype_indication,
:with_private>;

end chapter;

chapter ' 4 : NAMES AND EXPRESSIONS '

chapter ' 4.1 NAMES '

rules

    identifier_option :=;
        :void;

    identifier_option := identifier;
        :void;

    two_identifier_list := identifier "," identifier;
        :identifiers_list<identifier.0, identifier.1>;

    two_identifier_list := two_identifier_list "," identifier;
        two_identifier_list:<.., identifier>;

    identifier_list := identifier;
        :identifiers_list<identifier>;

    identifier_list := identifier_list "," identifier;
        identifier_list:<.., identifier>;

    name := identifier;
        identifier;

    name :=
        attribute;
        attribute;

    name :=
        ambig_slice_subtype_indication;
        ambig_slice_subtype_indication;

    name := selected_component;
        selected_component;

    name := indexed_component;
        indexed_component;
```

```
c_name_option :=;
    :void;

c_name_option := compound_name;
```

```
        :void;

    compound_name := identifier;
        :compound_name<identifier>;

    compound_name := compound_name "."
        identifier; compound_name:<..,
        identifier>;

    ambig_slice_subtype_indication := name "(" range_denotation
    ")";
        :slice<name, range_denotation>;

    ambig_slice_subtype_indication := function_call "("
")" range_denotation
;
        :slice<function_call,

    range_denotation>; selected_component :=

    operator "." selector;
        :selected_component<operator, selector>;

    selected_component := name "." selector;
        :selected_component<name, selector>;

    selected_component := function_call "." selector;
        :selected_component<function_call, selector>;

    selector :=
        designator;
        designator;

    selector :=
        character_literal;
        character_literal;

    selector := 'ALL';
        :all;

    indexed_component := function_call "(" expression_list ")";
        :indexed_component<function_call, expression_list>;

    attribute :=
        ambig_expression_subtype_indication;
        ambig_expression_subtype_indication;

    attribute :=
        simple_attribute;
        simple_attribute;

    ambig_expression_subtype_indication := name "(" expression_list
    ")";
        %{
     VTP_TreeP tr_expr = Parser_Pop();
```

```
V    ame = Parser_Pop();  if
T    (CheckOper(tr_name, attribute) &&
P        CheckOper(VTP_TreeDown(tr_name, 2), void) &&
_        CheckOper(tr_expr, expressions_list) &&
T        VTP_TreeLength(tr_expr) == 1) {
r        VTP_TreeDestroySetChild(tr_name, Ada_Disown(tr_expr,
e        0), 2); VTP_TreeDestroy(tr_expr);
e    } else {
P        tr_name =
                      (TreeMake2(indexed_component_or_functio
t                     n_call, tr_name,
r                     TreeRename(tr_expr, actual_parameter_part)));
_
n
     }
       Parser_SetCoordNT(tr_name, tr_name, $4);
```

```
        Parser_PopUntilToken($2);
      Parser_Push(tr_name);
      }%;

    simple_attribute := operator "'" identifier;
        :attribute<operator, identifier, :void>;

    simple_attribute := name "'" identifier;
        :attribute<name, identifier, :void>;

    simple_attribute := function_call "'" identifier;
        :attribute<function_call, identifier, :void>;

    simple_attribute := operator "'" 'DELTA';
        :attribute<operator, :identifier["DELTA"], :void>;

    simple_attribute := name "'" 'DELTA';
        :attribute<name, :identifier["DELTA"], :void>;

    simple_attribute := function_call "'" 'DELTA';
        :attribute<function_call, :identifier["DELTA"], :void>;

    simple_attribute := operator "'" 'DIGITS';
        :attribute<operator, :identifier["DIGITS"], :void>;

    simple_attribute := name "'" 'DIGITS';
        :attribute<name, :identifier["DIGITS"], :void>;

    simple_attribute := function_call "'" 'DIGITS';
        :attribute<function_call, :identifier["DIGITS"], :void>;

    simple_attribute := operator "'" 'RANGE';
        :attribute<operator, :identifier["RANGE"], :void>;

    simple_attribute := name "'" 'RANGE';
        :attribute<name, :identifier["RANGE"], :void>;

    simple_attribute := function_call "'" 'RANGE';
        :attribute<function_call, :identifier["RANGE"], :void>;

    simple_attribute := operator "'" 'ACCESS';
        :attribute<operator, :identifier["ACCESS"], :void>;

    simple_attribute := name "'" 'ACCESS';
        :attribute<name, :identifier["ACCESS"], :void>;

    simple_attribute := function_call "'" 'ACCESS';
        :attribute<function_call, :identifier["ACCESS"],

:void>; end chapter;

chapter ' 4.2 LITERALS

' rules
```

```
literal := numeric_literal;
    numeric_literal;
```

```
    literal :=
        character_string;
        character_string;

    literal := 'NULL';
        :null_access_valu
        e;

end chapter;

chapter ' 4.3 AGGREGATES '

rules

    aggregate := "(" 'NULL' 'RECORD' ")";
        :null_record_aggregat
        e;

    aggregate := "(" aggregate_list ")";
        aggregate_list;

    aggregate := "(" expression_list "," expression ")";
        %
        {
        VTP_TreeP       tr_expr       =
        Parser_Pop();        VTP_TreeP
        tr_list     =     Parser_Pop();
        VTP_TreeP tr;
        tr        =         PostRename(tr_expr,         tr_list,
        record_or_array_aggregate);
        Parser_SetCoordTT(tr, $1, $5);
        Parser_PopUntilToken($1);
        Parser_Push(tr);
      }%;

    aggregate_list := named_component;
        :record_or_array_aggregate<named_compone
        nt>;

    aggregate_list := expression_list "," named_component;
        %
        {
        VTP_TreeP       tr_comp       =
        Parser_Pop();        VTP_TreeP
        tr_list     =     Parser_Pop();
        VTP_TreeP tr;
        tr = PostRename(tr_comp, tr_list,
        record_or_array_aggregate); Parser_PopUntilToken($2);
        Parser_Push(tr
        );
      }%;

    aggregate_list := aggregate_list "," named_component;
        aggregate_list:<.., named_component>;
```

```
    named_component := choice_list "=>" expression;
        :named_association<choice_list,
        expression>;

    aggregate := "(" expression 'WITH' expression ")";
        :extension_aggregate<expression
        .0,
             :record_or_array_aggregate<expression.1>>;

    aggregate := "(" expression 'WITH' expression_list ","
")" expression
;
        %{
        VTP_TreeP tr_expr2 =
        Parser_Pop(); VTP_TreeP tr_list
        = Parser_Pop(); VTP_TreeP
        tr_expr1 = Parser_Pop();
```

```
        VTP_TreeP tr;
        tr = PostRename(tr_expr2, tr_list,
        record_or_array_aggregate); tr_expr1 =
        TreeMake2(extension_aggregate, tr_expr1, tr);
        Parser_SetCoordTT(tr_expr1, $1, $7);
        Parser_PopUntilToken($1);
        Parser_Push(tr_expr1);
      }%;

    aggregate := "(" expression 'WITH' aggregate_list ")";
        :extension_aggregate<expression, aggregate_list>;

    aggregate := "(" expression 'WITH' 'NULL' 'RECORD' ")";
        :extension_aggregate<expression,

:null_record_aggregate>; end chapter;

chapter ' 4.4 EXPRESSIONS '

rules

    expression_list := expression;
        :expressions_list<expression>;

    expression_list := expression_list "," expression;
        expression_list:<.., expression>;

    expression := expression 'AND' expression;
        :and<expression.0, expression.1>;

    expression := expression 'OR' expression;
        :or<expression.0, expression.1>;

    expression := expression 'XOR' expression;
        :xor<expression.0, expression.1>;

    expression := expression 'AND' 'THEN' expression;
        %[PREC 'AND' ]%
        :and_then<expression.0, expression.1>;

    expression := expression 'OR' 'ELSE' expression;
        %[PREC 'AND' ]%
        :or_else<expression.0, expression.1>;

    expression := simple_expr;
        simple_expr;

    expression := simple_expr "=" simple_expr;
        :equal<simple_expr.0, simple_expr.1>;

    expression := simple_expr "/=" simple_expr;
        :different<simple_expr.0, simple_expr.1>;

    expression := simple_expr "<" simple_expr;
        :less<simple_expr.0, simple_expr.1>;
```

```
expression := simple_expr ">" simple_expr;
    :greater<simple_expr.0, simple_expr.1>;
```

```
expression := simple_expr "<=" simple_expr;
    :less_equal<simple_expr.0, simple_expr.1>;

expression := simple_expr ">=" simple_expr;
    :greater_equal<simple_expr.0, simple_expr.1>;

expression := simple_expr 'IN' name;
    :member<simple_expr, name>;

expression := simple_expr 'IN' range1;
    :member<simple_expr, range1>;

expression := simple_expr 'NOT' 'IN' name;
    %[PREC 'IN' ]%
    :not_member<simple_expr, name>;

expression := simple_expr 'NOT' 'IN' range1;
    %[PREC 'IN' ]%
    :not_member<simple_expr, range1>;

simple_expr := simple_expr "+" simple_expr;
    :addition<simple_expr.0, simple_expr.1>;

simple_expr := simple_expr "-" simple_expr;
    :substraction<simple_expr.0, simple_expr.1>;

simple_expr := simple_expr "&" simple_expr;
    :catenation<simple_expr.0, simple_expr.1>;

simple_expr := "+" simple_expr;
    %[PREC %UNARY ]%
    :unary_plus<simple_expr>;

simple_expr := "-" simple_expr;
    %[PREC %UNARY ]%
    :unary_minus<simple_expr>;

simple_expr := simple_expr "*" simple_expr;
    :multiplication<simple_expr.0, simple_expr.1>;

simple_expr := simple_expr "/" simple_expr;
    :division<simple_expr.0, simple_expr.1>;

simple_expr := simple_expr 'MOD' simple_expr;
    :modulus<simple_expr.0, simple_expr.1>;

simple_expr := simple_expr 'REM' simple_expr;
    :remainder<simple_expr.0, simple_expr.1>;

simple_expr := simple_expr "**" simple_expr;
    :exponentiation<simple_expr.0, simple_expr.1>;

simple_expr := 'ABS' simple_expr;
    :abs<simple_expr>;
```

```
simple_expr := 'NOT' simple_expr;
```

```
:not<simple_expr>;
```

```
    simple_expr :=
        literal; literal;

    simple_expr := character_literal;
        character_literal;

    simple_expr := aggregate;
        aggregate;

    simple_expr := operator;
        %{
        VTP_TreeP tr_oper = Parser_Pop();
        Parser_Push(TreeRename(tr_oper, string_literal));
        }%;

    simple_expr := name;
        name;

    simple_expr := function_call;
        function_call;

    simple_expr := allocator;
        allocator;

    simple_expr := qualified_expression;
        qualified_expression;

    simple_expr := "(" expression ")";
        :parenthesis<expression

>; end chapter;

chapter ' 4.7 QUALIFIED EXPRESSIONS '

rules

    qualified_expression := name "'" "(" expression ")";
        :qualified_expression<name, expression>;

    qualified_expression := name "'" aggregate;
        :qualified_expression<name,

aggregate>; end chapter;

chapter ' 4.8 ALLOCATORS '

rules

    allocator := 'NEW' qualified_expression;
        :allocator<qualified_expression>;

    allocator := 'NEW'
    ambig_subtype_entry_subprogram_call_statement;
        :allocator<ambig_subtype_entry_subprogram_call_statement>
        ;
```

```
    allocator := 'NEW' name index_constraint;
        :allocator<:subtype_indication<name,

index_constraint>>; end chapter;
```

```
end chapter;

chapter ' 5 : STATEMENTS

' rules

    statement_option_list :=;
        :void;

    statement_option_list := statement_list;
        statement_list;

    statement_list := statement ";";
        :statements_list<statement>;

    statement_list := statement_list statement ";";
        statement_list:<.., statement>;

    statement := label statement;
        :labeled_statement<label, statement>;

    statement := unlabeled_statement;
        unlabeled_statement;

    statement := pragma;
        pragma;

    unlabeled_statement :=
        simple_statement; simple_statement;

    unlabeled_statement := compound_statement;
        compound_statement;

    simple_statement := assignment_statement;
        assignment_statement;

    simple_statement := ambig_entry_subprogram_call_statement;
        ambig_entry_subprogram_call_statement;

    simple_statement := exit_statement;
        exit_statement;

    simple_statement := return_statement;
        return_statement;

    simple_statement := goto_statement;
        goto_statement;

    simple_statement := raise_statement;
        raise_statement;

    simple_statement := abort_statement;
        abort_statement;

    simple_statement := requeue_statement;
```

```
requeue_statement;
```

```
    simple_statement := delay_statement;
        delay_statement;

    simple_statement := code_statement;
        code_statement;

    simple_statement := 'NULL';
        :null_statement;

    compound_statement := if_statement 'END' 'IF';
        if_statement;

    compound_statement := case_statement 'END'
        'CASE'; case_statement;

    compound_statement := loop_statement;
        loop_statement;

    compound_statement :=
        accept_statement;
        accept_statement;

    compound_statement := 'SELECT' select_statement 'END' 'SELECT';
        select_statement;

    compound_statement := block_statement;
        block_statement;

chapter ' 5.2 ASSIGNMENT STATEMENTS '

rules

    assignment_statement := name ":=" expression;
        %{
     VTP_TreeP tr_expr =
     Parser_Pop(); VTP_TreeP
     tr_name = Parser_Pop();
     if (CheckOper(tr_name, indexed_component_or_function_call)) {
         tr_expr = TreeMake2(assignment_statement,
                        TreeMake2(indexed_component,
                    Ada_Disown(tr_name, 0),
                    TreeRename(Ada_Disown(tr_name, 1),
                         expressions_list)),
                       tr_expr);
           Parser_SetCoordNN(tr_expr, tr_name, tr_expr);
           Parser_Push(tr_expr);
         VTP_TreeDestroy(tr_name);
     } else {
         Parser_Push(TreeMake2(assignment_statement, tr_name,
         tr_expr));
     }
        Parser_PopUntilToken($2);

     }%;

end chapter;
```

```
chapter ' 5.3 IF STATEMENTS '

rules
```

```
    if_statement := true_part_list else_part;
        true_part_list:<.., else_part>;

    if_statement := true_part_list;
        true_part_list;

    true_part_list := if_then_part;
        :if_statement<if_then_part>;

    true_part_list := true_part_list elsif_part;
        true_part_list:<.., elsif_part>;

    if_then_part := 'IF' condition 'THEN' statement_list;
        :conditional_clause<condition, statement_list>;

    elsif_part := 'ELSIF' condition 'THEN' statement_list;
        :conditional_clause<condition, statement_list>;

    else_part := 'ELSE' statement_list;
        :conditional_clause<:void, statement_list>;

    condition :=
        expression;
        expression;

end chapter;

chapter ' 5.4 CASE STATEMENTS '

rules

    case_statement := 'CASE' expression
'IS' case_statement_alternative_list;
        :case_statement<expression,
        case_statement_alternative_list>;

    case_statement_alternative_list :=
        pragma_option_list
        case_statement_alternative;
        %{
        VTP_TreeP tr_alt =
        Parser_Pop(); VTP_TreeP tr_list
        = Parser_Pop(); VTP_TreeP tr;
        tr = PostRename(tr_alt, tr_list, alternatives_list);
      Parser_Push(tr);
        }%;

    case_statement_alternative_list :=
        case_statement_alternative_list
        case_statement_alternative;
        case_statement_alternative_list:<..,
        case_statement_alternative>;

    case_statement_alternative := 'WHEN' choice_list "=>"
    statement_list;
        :alternative<choice_list,
```

```
statement_list>; end chapter;

chapter ' 5.5 LOOP STATEMENTS '

rules

    loop_statement := named_loop_statement;
```

```
        named_loop_statement;

    loop_statement := unnamed_loop_statement;
        unnamed_loop_statement;

    named_loop_statement := designator ":"
unnamed_loop_statement designator;
        :named_statement<designator.0, unnamed_loop_statement>;

    unnamed_loop_statement := iteration_sheme basic_loop;
        :loop_statement<iteration_sheme, basic_loop>;

    basic_loop := 'LOOP' statement_list 'END'
        'LOOP'; statement_list;

    iteration_sheme :=;
        :void;

    iteration_sheme := 'FOR' identifier 'IN' discrete_range;
        :for<identifier, discrete_range>;

    iteration_sheme := 'FOR' identifier 'IN' 'REVERSE'
    discrete_range;
        :reverse<identifier, discrete_range>;

    iteration_sheme := 'WHILE' condition;
        :while<condition

>; end chapter;

chapter ' 5.6 BLOCK STATEMENTS '

rules

    block_statement := named_block_statement;
        named_block_statement;

    block_statement :=
        unnamed_block_statement;
        unnamed_block_statement;

    named_block_statement := designator ":"
unnamed_block_statement designator;
        :named_statement<designator.0, unnamed_block_statement>;

    unnamed_block_statement :=
        declare_part_option 'BEGIN' %[CBLOCK_BEGIN]%
        statement_list
          %[CBLOCK_END]% exception_option 'END';
        :block_statement<declare_part_option, statement_list,
exception_option>;

    declare_part_option :=;
        :declarative_part<>;
```

```
declare_part_option := 'DECLARE'
    declarative_part; declarative_part;

exception_option :=;
    :void;
```

```
    exception_option :=
        'EXCEPTION' %[CBLOCK_BEGIN]% exception_handler_list
%[CBLOCK_END]%;
        exception_handler_list;

end chapter;

chapter ' 5.7 EXIT STATEMENTS '

rules

    exit_statement := 'EXIT' dot_name_option when_condition_option;
        :exit_statement<dot_name_option, when_condition_option>;

    dot_name_option :=;
        :void;

    dot_name_option := name;
        name;

    when_condition_option :=;
        :void;

    when_condition_option := 'WHEN'
        condition; condition;

end chapter;

chapter ' 5.8 GOTO STATEMENTS '

rules

    goto_statement := 'GOTO' name;
        :goto_statement<name

>; end chapter;

end chapter;

chapter ' 6 : SUBPROGRAMS '

rules

    declarative_part := %[CBLOCK_BEGIN]% declarative_part_list
%[CBLOCK_END]%;
        declarative_part_list;

    declarative_part_list :=;
        :declarative_part<>;

    declarative_part_list := declarative_part_list
";"
;       declarative_part_item declarative_part_list:<..,

        declarative_part_item>;
```

```
decl ative_part_item := body;
arat
ive_
part
_ite
m :=
decl
arat
ive_
item
;
    d

e

c

l

a

r

a

t

i

v

e

_

i

t

e

m

;



d

e

c

l

a

r
```

```
        body;

    declarative_item := declaration;
        declaration;

    declarative_item :=
        use_clause; use_clause;

    declarative_item :=
        representation_clause;
        representation_clause;

    body := subprogram_body;
        subprogram_body;

    body := package_body;
        package_body;

    body := task_body;
        task_body;

    body := protected_body;
        protected_body;

    body := body_stub;
        body_stub;

chapter ' 6.1 SUBPROGRAM DECLARATIONS '

rules

    subprogram_declaration := 'FUNCTION' function_header
is_abstract_option;
        :subprogram_declaration<function_header,
        is_abstract_option>;

    subprogram_declaration := 'PROCEDURE' compound_name
is_abstract_option;
        :subprogram_declaration<:procedure_header<compound_name,

:procedure_formal_part<>
>,                              is_abstract_option>;

    subprogram_declaration := 'PROCEDURE'
procedure_header is_abstract_option;
        :subprogram_declaration<procedure_header,
        is_abstract_option>;

    is_abstract_option := ;
        :void;

    is_abstract_option := 'IS' 'ABSTRACT';
        :is_abstract;

    function_header := c_designator fct_formal_part_option 'RETURN'
```

```
name;
    :function_header<c_designator, fct_formal_part_option,
    name>;


procedure_header := compound_name formal_part;
    :procedure_header<compound_name, formal_part>;
```

```
    c_designator_option :=;
        :void;

    c_designator_option := c_designator;
        :void;

    proc_ident_is := 'PROCEDURE' compound_name 'IS';
        compound_name;

    fct_formal_part_option :=;
        :function_formal_part<>;

    fct_formal_part_option := "(" fct_parameter_list ")";
        fct_parameter_list;

    fct_parameter_list := in_parameter_declaration;
        :function_formal_part<in_parameter_declaration>;

    fct_parameter_list := access_parameter_declaration;
        :function_formal_part<access_parameter_declaration>;

    fct_parameter_list := fct_parameter_list
";" in_parameter_declaration;
        fct_parameter_list:<.., in_parameter_declaration>;

    fct_parameter_list := fct_parameter_list ";"
access_parameter_declaration;
        fct_parameter_list:<.., access_parameter_declaration>;

    formal_part_option :=;
        :procedure_formal_part<>;

    formal_part_option := formal_part;
        formal_part;

    formal_part := "(" parameter_specification_list ")";
        parameter_specification_list;

    parameter_specification_list := any_parameter_declaration;
        :procedure_formal_part<any_parameter_declaration>;

    parameter_specification_list :=
        parameter_specification_list ";"
        any_parameter_declaration;
        parameter_specification_list:<..,
        any_parameter_declaration>;

    any_parameter_declaration := in_parameter_declaration;
        in_parameter_declaration;

    any_parameter_declaration := access_parameter_declaration;
        access_parameter_declaration;

    any_parameter_declaration := in_out_parameter_declaration;
        in_out_parameter_declaration;
```

```
any_parameter_declaration :=
    out_parameter_declaration;
    out_parameter_declaration;
```

```
    in_parameter_declaration := identifier_list ":"
name initialization_option;
        :parameter<identifier_list, name, initialization_option>;

    in_parameter_declaration :=
        identifier_list ":" 'IN' name initialization_option;
        :in_parameter<identifier_list, name,
        initialization_option>;

    access_parameter_declaration :=
        identifier_list ":" 'ACCESS' name initialization_option;
        :access_parameter<identifier_list, name,
        initialization_option>;

    in_out_parameter_declaration := identifier_list ":" 'IN' 'OUT'
    name;
        :in_out_parameter<identifier_list, name>;

    out_parameter_declaration := identifier_list ":" 'OUT' name;
        :out_parameter<identifier_list, name>;

end chapter;

chapter ' 6.3 SUBPROGRAM BODIES '

rules

    subprogram_body :=
        'FUNCTION' function_header 'IS' block 'END'
        c_designator_option;
        :subprogram_body<function_header, block>;

    subprogram_body := proc_ident_is block 'END' c_name_option;
        :subprogram_body<
            :procedure_header<proc_ident_is,:procedure_formal_part<>
            >,
block>;

    subprogram_body :=
        'PROCEDURE' procedure_header 'IS' block 'END'
        c_name_option;
        :subprogram_body<procedure_header, block>;

    block :=
        declarative_part 'BEGIN' %[CBLOCK_BEGIN]% statement_list
%[CBLOCK_END]%
            exception_option;
        :block_statement<declarative_part, statement_list,
exception_option

>; end chapter;

chapter ' 6.4 SUBPROGRAM CALLS '

rules
```

```
    ambig_entry_subprogram_call_statement := name;
        %{
      VTP_TreeP tr_name = Parser_Pop();
      if (CheckOper(tr_name, indexed_component_or_function_call)
          || CheckOper(tr_name, function_call)) {
        Parser_Push(TreeRename(tr_name,
procedure_or_entry_call_statement));
      } else {
```

```
       Parser_Push(TreeMake2(procedure_or_entry_call_statement,
          tr_name, TreeMake0(actual_parameter_part,
          TreeCopy(tr_name))));
     }
     }%
     ;

    ambig_entry_subprogram_call_statement :=
       name "(" parameter_association_list
       ")";
       :procedure_or_entry_call_statement<name,
parameter_association_list>;

    function_call := operator "(" expression_list ")";
       %{
     VTP_TreeP       tr_expr       =
     Parser_Pop();         VTP_TreeP
     tr_oper     =     Parser_Pop();
     tr_oper =
         TreeMake2(function_call, tr_oper,
          TreeRename(tr_expr, actual_parameter_part));
       Parser_SetCoordNT(tr_oper, tr_oper, $4);

       Parser_PopUntilToken($2);
     Parser_Push(tr_oper);
     }%
     ;

    function_call := operator actual_parameter_part_option;
       :function_call<operator, actual_parameter_part_option>;

    function_call := name actual_parameter_part_option;
       :function_call<name, actual_parameter_part_option>;

    actual_parameter_part_option := "(" parameter_association_list
       ")"; parameter_association_list;

    parameter_association_list := parameter_association;
       :actual_parameter_part<parameter_association>;

    parameter_association_list := expression_list ","
parameter_association;
       %{
       VTP_TreeP tr_param = Parser_Pop();
       VTP_TreeP tr_list = Parser_Pop();
     VTP_TreeP
     tr;
       tr = PostRename(tr_param, tr_list, actual_parameter_part);
       Parser_PopUntilToken($2);
     Parser_Push(tr);
     }%
     ;

    parameter_association_list :=
       parameter_association_list ","
```

```
    parameter_association;
    parameter_association_list:<..,
    parameter_association>;

parameter_association := parameter_name_list "=>" expression;
    :named_association<parameter_name_list, expression>;

parameter_name_list := designator;
    :choices_list<designator>;

parameter_name_list := character_literal;
    :choices_list<character_literal>;
```

```
    parameter_name_list := parameter_name_list "|" designator;
        parameter_name_list:<.., designator>;

    parameter_name_list := parameter_name_list "|"
        character_literal; parameter_name_list:<..,
        character_literal>;

end chapter;

chapter ' 6.5 RETURN STATEMENTS '

rules

    return_statement := 'RETURN';
        :return_statement<:void>;

    return_statement := 'RETURN' expression;
        :return_statement<expression>;

end chapter;

end chapter;

chapter ' 7 : PACKAGES '

chapter ' 7.1 PACKAGE SPECIFICATIONS AND DECLARATIONS  '

rules

    package_declaration := package_specification;
        package_specification;

    generic_instantiation_decl :=
        'PACKAGE' compound_name 'IS' generic_instantiation;
        :package_declaration<compound_name,
        generic_instantiation>;

    package_specification :=
        'PACKAGE' compound_name 'IS' package_specif_body 'END'
c_name_option;
        :package_declaration<compound_name, package_specif_body>;

    package_specif_body := declarative_item_option_list
private_part_option;
        :package_specification<declarative_item_option_list,
                                 private_part_option>;

    private_part_option :=;
        :basic_declarative_part<>;

    private_part_option := 'PRIVATE'
        declarative_item_option_list;
        declarative_item_option_list;
```

```
declarative_item_option_list :=;
    :basic_declarative_part<>;

declarative_item_option_list :=
    declarative_item_option_list declarative_item ";";
    declarative_item_option_list:<..,
    declarative_item>;
```

```
    package_body :=
        'PACKAGE' 'BODY' compound_name 'IS' package_block
'END' c_name_option;
        :package_body<compound_name, package_block>;

    package_block := declarative_part;
        :block_statement<declarative_part,
:statements_list<:null_statement>,
                            :void>;

    package_block :=
        declarative_part 'BEGIN' %[CBLOCK_BEGIN]% statement_list
%[CBLOCK_END]%
           exception_option;
        :block_statement<declarative_part, statement_list,
exception_option>;

end chapter;

chapter ' 7.3 PRIVATE TYPES AND PRIVATE EXTENSIONS '

rules

    private_type_declaration :=
        'TYPE' identifier known_discriminant_part_option_is
private_type_definition;
        :type_declaration<identifier,
        known_discriminant_part_option_is,
                             private_type_definition>;

    private_type_declaration :=
        'TYPE' identifier "(" "<>" ")" 'IS' private_type_definition;
        :type_declaration<identifier, :unknown_discriminant,
                             private_type_definition>;

    private_extension_declaration :=
        'TYPE' identifier known_discriminant_part_option_is
                             private_extension_definition;
        :type_declaration<identifier,
        known_discriminant_part_option_is,
                             private_extension_definition>;


    private_extension_declaration :=
        'TYPE' identifier "(" "<>" ")" 'IS'
        private_extension_definition;
        :type_declaration<identifier, :unknown_discriminant,
                             private_extension_definition>;

end chapter;


end chapter;
```

chapter ' 8 : VISIBILITY RULES '

chapter ' 8.4 USE CLAUSES '

rules

```
    use_clause := 'USE' name;
        :use_package_clause<name>;

    use_clause := 'USE' 'TYPE' name;
        :use_type_clause<name>;

    use_clause := use_clause ","
        name; use_clause:<.., name>;

end chapter;

chapter ' 8.5 RENAMING DECLARATIONS '

rules

    renaming_declaration := identifier ":" qualifier_option
             name 'RENAMES' rename;
        :object_declaration<:identifiers_list<identifier.0>, :void,
                                :subtype_indication<name, :void>,
rename>;

    renaming_declaration := identifier ":" 'EXCEPTION' 'RENAMES'
    rename;
        :exception_declaration<:identifiers_list<identifier>,
        rename>;

    unit_renaming_declaration := 'FUNCTION' function_header
'RENAMES' rename;
        :subprogram_declaration<function_header, rename>;

    unit_renaming_declaration := 'PROCEDURE' compound_name
'RENAMES' rename;
        :subprogram_declaration<:procedure_header<compound_name,

:procedure_formal_part<>
>,                                rename>;

    unit_renaming_declaration := 'PROCEDURE' procedure_header
'RENAMES' rename;
        :subprogram_declaration<procedure_header, rename>;

    unit_renaming_declaration := 'PACKAGE' compound_name 'RENAMES'
rename;
        :package_declaration<compound_name, rename>;

    unit_renaming_declaration := 'GENERIC'
            generic_parameter_list_option 'FUNCTION' compound_name
            'RENAMES' rename;
        :generic_declaration<:generic_formal_part<>,
            :subprogram_declaration<:function_header<compound_name,
                                     :function_formal_part<>,:void>
                                , rename>>;

    unit_renaming_declaration := 'GENERIC'
            generic_parameter_list_option 'PROCEDURE' compound_name
            'RENAMES' rename;
```

```
    :generic_declaration<:generic_formal_part<>,
      :subprogram_declaration<:procedure_header<compound_name,
                                  :procedure_formal_part<>>,
                                rename>>;

unit_renaming_declaration := 'GENERIC'
generic_parameter_list_option
```

```
            'PACKAGE' compound_name 'RENAMES' rename;
        :generic_declaration<:generic_formal_part<>,
            :package_declaration<compound_name, rename>>;

    rename := operator;
        :renaming<operator>;

    rename := name;
        :renaming<name>;


    rename := character_literal;
        :renaming<character_literal>;

end chapter;

end chapter;

chapter ' 9 : TASKS AND SYNCHRONIZATION '

chapter ' 9.1 TASK UNITS AND TASK OBJECTS '

rules

    task_declaration := task_specification;
        task_specification;

    task_specification := 'TASK' 'TYPE' identifier
            discriminant_part task_definition_option;

:type_declaration<identifier,discriminant_part,task_definition_opti

    on>; task_specification := 'TASK' 'TYPE' identifier
task_definition_option;
        :type_declaration<identifier, :discriminant_part<>,
                              task_definition_option>;

    task_specification := 'TASK' identifier task_definition_option;
        :task_declaration<identifier, task_definition_option>;

    task_definition := 'IS' task_items_option_list
            task_private_option 'END' identifier_option;
        :task_specification<task_items_option_list,task_private_opt
        ion>;

    task_definition_option :=;
        :task_specification<:task_items_list<>,:task_items_list<>>;

    task_definition_option := task_definition;
        task_definition;

    task_items_option_list :=;
        :task_items_list<>;

    task_items_option_list := task_items_option_list task_item
        ";"; task_items_option_list:<.., task_item>;
```

```
task_private_option :=;
    :task_items_list<>;
```

```
    task_private_option := 'PRIVATE'
        task_items_option_list; task_items_option_list;

    task_item:= entry_declaration;
        entry_declaration;

    task_item:= representation_clause;
        representation_clause;

    task_body := 'TASK' 'BODY' identifier 'IS' block
'END' identifier_option;
        :task_body<identifier, block>;

end chapter;


chapter ' 9.4 PROTECTED UNITS AND PROTECTED OBJECTS '

rules

    protected_declaration :=
        protected_specification;
        protected_specification;

    protected_specification := 'PROTECTED' 'TYPE' identifier
              discriminant_part protected_definition;
        :type_declaration<identifier, discriminant_part,
              protected_definition>;

    protected_specification := 'PROTECTED' 'TYPE' identifier
              protected_definition;
        :type_declaration<identifier, :discriminant_part<>,
              protected_definition>;

    protected_specification := 'PROTECTED' identifier
              protected_definition;
        :protected_declaration<identifier, protected_definition>;

    protected_definition := 'IS'
protected_operation_declaration_option_
list
              protected_private_option 'END' identifier_option;

:protected_specification<protected_operation_declaration_option
              _list, protected_private_option>;

    protected_operation_declaration_option_list :=;
        :protected_operation_declarations_list<>;

    protected_operation_declaration_option_list :=
              protected_operation_declaration_option_
              list protected_operation_declaration;
        protected_operation_declaration_option_list:<..,
              protected_operation_declaration>;
```

```
protected_private_option :=;
    :protected_element_declarations_list<>;

protected_private_option := 'PRIVATE'
```

```
              protected_element_declaration_option_list;
        protected_element_declaration_option_list;

    protected_element_declaration_option_list :=;
        :protected_element_declarations_list<>;

    protected_element_declaration_option_list :=
              protected_element_declaration_option_list
              protected_element_declaration;
        protected_element_declaration_option_list:
              <.., protected_element_declaration>;

    protected_operation_declaration := subprogram_declaration ";";
        subprogram_declaration;

    protected_operation_declaration := entry_declaration
        ";"; entry_declaration;

    protected_operation_declaration := representation_clause ";";
        representation_clause;

    protected_element_declaration := subprogram_declaration
        ";"; subprogram_declaration;

    protected_element_declaration := entry_declaration ";";
        entry_declaration;

    protected_element_declaration :=
        component; component;

    protected_body := 'PROTECTED' 'BODY' identifier 'IS'
           protected_operation_item_option_list 'END'
           identifier_option;
        :protected_body<identifier,protected_operation_item_option_l
        ist>;

    protected_operation_item_option_list :=;
        :protected_operation_items_list<>;

    protected_operation_item_option_list :=
              protected_operation_item_option_list
protected_operation_item;

protected_operation_item_option_list:<..,protected_operation_item>;

    protected_operation_item := pragma ";";
        pragma;

    protected_operation_item := subprogram_declaration ";";
        subprogram_declaration;

    protected_operation_item := subprogram_body ";";
        subprogram_body;

    protected_operation_item := entry_body
```

```
    ";"; entry_body;

protected_operation_item := representation_clause ";";
    representation_clause;
```

```
end chapter;

chapter ' 9.5 INTERTASK COMMUNICATION '

rules

    entry_declaration :=
        pragma; pragma;

    entry_declaration := 'ENTRY' entry_header;
        entry_header;

    entry_header := identifier "(" discrete_range ")"
    formal_part_option;
        :entry_declaration<identifier, discrete_range,
formal_part_option>;

    entry_header := identifier formal_part_option;
        :entry_declaration<identifier, :void, formal_part_option>;

    accept_statement := 'ACCEPT' identifier formal_part_option;
        :accept_statement<identifier, formal_part_option, :void>;

    accept_statement :=
        'ACCEPT' identifier formal_part_option 'DO' statement_list
        'END'
            identifier_option;
        :accept_statement<identifier, formal_part_option,
statement_list>;

    accept_statement := 'ACCEPT' indexed_entry_name
    formal_part_option;
        :accept_statement<indexed_entry_name, formal_part_option,
        :void>;

    accept_statement :=
        'ACCEPT' indexed_entry_name formal_part_option 'DO'
statement_list
            'END' identifier_option;
        :accept_statement<indexed_entry_name, formal_part_option,
                         statement_list>;

    indexed_entry_name := identifier "(" expression
    ")";
        :indexed_component<identifier,
        :expressions_list<expression>>;

    entry_body := 'ENTRY' identifier formal_part_option 'WHEN'
                  expression 'IS' entry_block 'END'
                  identifier_option;
        :entry_body<identifier, :void,
             formal_part_option, expression,
             entry_block>;

    entry_body := 'ENTRY' identifier "(" entry_index
```

```
    ")"
                   formal_part_option 'WHEN' expression 'IS'
                   entry_block 'END' identifier_option;
        :entry_body<identifier, entry_index,
             formal_part_option, expression, entry_block>;

    entry_index := 'FOR' identifier 'IN'
    discrete_range;
        :entry_index<identifier, discrete_range>;

    entry_block :=
        declarative_part 'BEGIN' %[CBLOCK_BEGIN]% statement_list
%[CBLOCK_END]%
```

```
            exception_option;
        :block_statement<declarative_part, statement_list,
exception_option>;

    requeue_statement := 'REQUEUE' name;
        :requeue_statement <name, :void>;

    requeue_statement := 'REQUEUE' name with_abort;
        :requeue_statement <name, with_abort>;

    with_abort := 'WITH' 'ABORT';
        :with_abort

; end chapter;

chapter ' 9.6 DELAY STATEMENTS '

rules

    delay_statement := 'DELAY' expression;
        :delay_relative_statement<expression>;

    delay_statement := 'DELAY' 'UNTIL' expression;
        :delay_until_statement<expression>;

end chapter;

chapter ' 9.7 SELECT STATEMENTS '

rules

    select_statement := selective_accept;
        selective_accept;

    select_statement := timed_entry_call;
        timed_entry_call;

    select_statement :=
        conditional_entry_call;
        conditional_entry_call;

    select_statement :=
        asynchronous_select;
        asynchronous_select;

    selective_accept := select_alternative_list 'ELSE'
    statement_list;
        :selective_accept<select_alternative_list,
        statement_list>;

    selective_accept := select_alternative_list;
        :selective_accept<select_alternative_list, :void>;

    conditional_entry_call :=
        entry_call ";" statement_option_list 'ELSE'
        statement_list;
```

```
    :conditional_entry_call<entry_call, statement_option_list,
                            statement_list>;

timed_entry_call :=
    entry_call ";" statement_option_list 'OR'
    delay_alternative;
    :timed_entry_call<entry_call, statement_option_list,
                delay_alternative>;
```

```
    delay_alternative := delay_statement ";"
    statement_option_list;
   :delay_alternative<delay_statement, statement_option_list>;

    entry_call := ambig_entry_subprogram_call_statement;
        ambig_entry_subprogram_call_statement;

    select_alternative_list := guarded_select_alternative;
        :select_clauses_list<guarded_select_alternative>;

    select_alternative_list := select_alternative_list
            'OR' guarded_select_alternative;
        select_alternative_list:<.., guarded_select_alternative>;

    guarded_select_alternative := select_alternative;
        :select_clause<:void, select_alternative>;

    guarded_select_alternative := 'WHEN' condition "=>"
select_alternative;
        :select_clause<condition, select_alternative>;

    select_alternative := accept_statement ";"
    statement_option_list;
        :accept_alternative<accept_statement,
        statement_option_list>;

    select_alternative := delay_statement ";"
    statement_option_list;
        :delay_alternative<delay_statement,
        statement_option_list>;

    select_alternative := terminate_alternative ";";
        terminate_alternative;

    terminate_alternative := 'TERMINATE';
        :terminate_alternative;

    asynchronous_select := entry_call ";"
             statement_option_list 'THEN' 'ABORT'
             statement_list;
        :asynchronous_select<entry_call, statement_option_list,
             statement_list>;

    asynchronous_select := delay_statement ";"
             statement_option_list 'THEN' 'ABORT' statement_list;
        :asynchronous_select<delay_statement,
             statement_option_list, statement_list>;

end chapter;

chapter ' 9.8 ABORT STATEMENTS '

rules
```

```
abort_statement := 'ABORT' name_list;

    name_list;

name_list := name;
    :abort_statement<name>;

name_list := name_list "," name;
    name_list:<.., name>;
```

```
end chapter;

end chapter;

chapter ' 10 : PROGRAM STRUCTURE AND COMPILATION ISSUES '

chapter ' 10.1.1 COMPILATION UNITS '

rules

    compilation_unit_list := compilation_unit ";";
        :compilation<compilation_unit>;

    compilation_unit_list := compilation_unit_list compilation_unit
        ";"; compilation_unit_list:<.., compilation_unit>;

    compilation_unit := pragma;
        pragma;

    compilation_unit := context_specif_option
             private_unit_option unit_declaration;
        :comp_unit<context_specif_option, private_unit_option,
             unit_declaration>;

    compilation_unit := context_specif_option private_unit_option
unit_body;
        :comp_unit<context_specif_option, :void, unit_body>;

    context_specif_option :=;
        :context_clause<>;

    context_specif_option := with_use_list;
        with_use_list;

    with_use_list := with_clause;
        :context_clause<with_clause>;

    with_use_list := with_use_list
        with_clause; with_use_list:<..,
        with_clause>;

    with_use_list := with_use_list use_clause ";";
        with_use_list:<.., use_clause>;

    with_use_list := with_use_list pragma
        ";"; with_use_list:<.., pragma>;

    private_unit_option:=;
        :void;

    private_unit_option:= 'PRIVATE';
        :private_unit;

    unit_declaration := subprogram_declaration;
        subprogram_declaration;
```

```
unit_declaration := unit_renaming_declaration;
    unit_renaming_declaration;
```

```
unit_declaration := generic_declaration;
    generic_declaration;

unit_declaration :=
    generic_instantiation_decl;
    generic_instantiation_decl;

unit_declaration := package_declaration;
    package_declaration;

unit_body := subprogram_body;
    subprogram_body;

unit_body := package_body;
    package_body;


unit_body := sub_unit;
    sub_unit;

sub_unit := 'SEPARATE' "(" name ")" proper_body;
    :subunit<name, proper_body>;

pragma_option_list :=;
    :pragmas<>;

pragma_option_list := pragma_option_list pragma ";";
    pragma_option_list:<.., pragma>;

pragma := 'PRAGMA' identifier;
    :pragma<identifier, :actual_parameter_part<>>;

pragma := 'PRAGMA' identifier "(" expression_list ")";
    %{
  VTP_TreeP tr_expr =
  Parser_Pop(); VTP_TreeP tr_id =
  Parser_Pop(); tr_id =
     (TreeMake2(pragma, tr_id,
       TreeRename(tr_expr, actual_parameter_part)));
    Parser_SetCoordTT(tr_id, $1, $5);
    Parser_PopUntilToken($1);
  Parser_Push(tr_id);
  }%;

pragma := 'PRAGMA' identifier "(" parameter_association_list
")";
    :pragma<identifier, parameter_association_list>;

with_clause := 'WITH' c_name_list ";";
    c_name_list;

c_name_list := compound_name;
    :with_clause<compound_name>;

c_name_list := c_name_list "," compound_name;
```

```
        c_name_list:<.., compound_name>;
end chapter;
```

```
chapter ' 10.1.3 SUBUNITS OF COMPILATION UNITS '

rules

    proper_body := subprogram_body;
        subprogram_body;

    proper_body := package_body;
        package_body;

    proper_body := task_body;
        task_body;

    proper_body := protected_body;
        protected_body;

    body_stub := 'FUNCTION' function_header stub;
        :subprogram_body<function_header, stub>;

    body_stub := 'PROCEDURE' compound_name stub;
        :subprogram_body<:procedure_header<compound_name,
:procedure_formal_part<>>,
                            stub>;

    body_stub := 'PROCEDURE' procedure_header stub;
        :subprogram_body<procedure_header, stub>;

    body_stub := 'PACKAGE' 'BODY' compound_name stub;
        :package_body<compound_name, stub>;

    body_stub := 'TASK' 'BODY' identifier stub;
        :task_body<identifier, stub>;

    body_stub := 'PROTECTED' 'BODY' identifier stub;
        :protected_body<identifier, stub>;

    stub := 'IS' 'SEPARATE';
        :stub;

end chapter;

end chapter;

chapter ' 11 : EXCEPTIONS '

chapter ' 11.1 EXCEPTION DECLARATIONS '

rules

    exception_declaration := identifier ":" 'EXCEPTION';
        :exception_declaration<:identifiers_list<identifier>,
        :void>;

    exception_declaration := two_identifier_list ":"
    'EXCEPTION';
```

```
        :exception_declaration<two_identifier_list,
:void>; end chapter;
chapter ' 11.2 EXCEPTION HANDLERS '
```

```
rules

    exception_handler_list := pragma_option_list exception_handler;
        %{
        VTP_TreeP tr_handl =
        Parser_Pop(); VTP_TreeP tr_list
        = Parser_Pop(); VTP_TreeP tr;
        tr = PostRename(tr_handl, tr_list,
      exception_alternatives_list); Parser_Push(tr);
        }%;

    exception_handler_list := exception_handler_list
        exception_handler; exception_handler_list:<..,
        exception_handler>;

    exception_handler := 'WHEN' exception_choice_list "=>"
statement_list;
        :exception_alternative<:void, exception_choice_list,
statement_list>;

    exception_handler := 'WHEN' identifier ":" exception_choice_list
             "=>" statement_list;
        :exception_alternative<identifier,
             exception_choice_list,
             statement_list>;

    exception_choice_list := exception_choice;
        :choices_list<exception_choice>;

    exception_choice_list := exception_choice_list "|"
        exception_choice; exception_choice_list:<..,
        exception_choice>;

    exception_choice := name;
        name;

    exception_choice := 'OTHERS';
        :others;

end chapter;

chapter ' 11.3 RAISE STATEMENTS '

rules

    raise_statement := 'RAISE' name;
        :raise_statement<name>;

    raise_statement := 'RAISE';
        :raise_statement<:void>;

end chapter;

end chapter;
```

chapter ' 12 : GENERIC UNITS '

chapter ' 12.1 GENERIC

DECLARATIONS '

rules

```
    generic_declaration := 'GENERIC'
            generic_parameter_list_option subprogram_declaration;
        :generic_declaration<generic_parameter_list_option,
            subprogram_declaration>;

    generic_declaration := 'GENERIC'
            generic_parameter_list_option package_specification;
        :generic_declaration<generic_parameter_list_option,
            package_specification>;

    generic_parameter_list_option :=;
        :generic_formal_part<>;

    generic_parameter_list_option := generic_parameter_list;
        generic_parameter_list;

    generic_parameter_list := generic_parameter ";";
        :generic_formal_part<generic_parameter>;

    generic_parameter_list := generic_parameter_list
";"
;       generic_parameter generic_parameter_list:<..,

        generic_parameter>;

    generic_parameter := in_parameter_declaration;
        in_parameter_declaration;

    generic_parameter :=
        in_out_parameter_declaration;
        in_out_parameter_declaration;

    generic_parameter :=
        generic_type_declaration;
        generic_type_declaration;

    generic_parameter := 'WITH' 'FUNCTION'
function_header default_subprogram;
        :subprogram_declaration<function_header,
        default_subprogram>;

    generic_parameter := 'WITH' 'PROCEDURE'
compound_name default_subprogram;
        :subprogram_declaration<:procedure_header<compound_name,

:procedure_formal_part<>
>,                               default_subprogram>;

    generic_parameter := 'WITH' 'PROCEDURE'
procedure_header default_subprogram;
        :subprogram_declaration<procedure_header,
        default_subprogram>;

    generic_parameter := 'WITH' 'PACKAGE' identifier
```

```
            'IS' formal_package_specification;
    :package_declaration<identifier,formal_package_specificati
    on>;

default_subprogram :=;
    :void;

default_subprogram := 'IS'
    operator; operator;
```

```
    default_subprogram := 'IS'
        name; name;

    default_subprogram := 'IS' character_literal;
        character_literal;

    default_subprogram := 'IS' "<>";
        :box;

    formal_package_specification := 'NEW' name "(" "<>" ")";
        :formal_package_box<name>;

    formal_package_specification :=
        generic_instantiation; generic_instantiation;

end chapter;

chapter ' GENERIC TYPE DECLARATIONS '

rules

    generic_type_declaration := private_type_declaration;
        %{
        VTP_TreeP tr = Parser_Pop();
      tr = TreeRename(tr, generic_type);
      Parser_Push(tr);
      }%;

    generic_type_declaration := 'TYPE' identifier
          known_discriminant_part_option_is
formal_derived_type_definition;
        :generic_type<identifier,
        known_discriminant_part_option_is,
                        formal_derived_type_definition>;

    generic_type_declaration := 'TYPE' identifier "(" "<>" ")"
    'IS'
                        formal_derived_type_definition;
        :generic_type<identifier, :unknown_discriminant,
                        formal_derived_type_definition>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                       "(" "<>" ")";
        :generic_type<identifier, discriminant_part_option_is,
                        :generic_formal_discrete_type>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                           'RANGE'
                            "<>";
        :generic_type<identifier, discriminant_part_option_is,
                        :generic_formal_integer_type>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
```

```
                    'MOD' "<>";
:generic_type<identifier, discriminant_part_option_is,
                    :generic_formal_modular_type>;
```

```
    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                        'DELTA' "<>";
        :generic_type<identifier, discriminant_part_option_is,
                        :generic_formal_fixed_point_type>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                        'DELTA' "<>" 'DIGITS' "<>";
        :generic_type<identifier, discriminant_part_option_is,
                        :generic_formal_decimal_fixed_point_type>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                           'DIGITS'
                            "<>";
        :generic_type<identifier, discriminant_part_option_is,
                        :generic_formal_floating_point_type>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                       array_type_definition;
        :generic_type<identifier, discriminant_part_option_is,
                        array_type_definition>;

    generic_type_declaration := 'TYPE' identifier
discriminant_part_option_is
                        access_type_definition;
        :generic_type<identifier, discriminant_part_option_is,
                        access_type_definition>;

    formal_derived_type_definition := abstract_option 'NEW'
            name private_extension_option;
        :derived_type<abstract_option, :subtype_indication<name,
            :void>, private_extension_option>;

    private_extension_option := ;
        :void;

    private_extension_option := 'WITH' 'PRIVATE';
        :with_private;

end chapter;

chapter ' 12.3 GENERIC INSTANTIATION
                 '

rules

    generic_instantiation_decl :=
        'FUNCTION' c_designator 'IS' generic_instantiation;
        :subprogram_declaration<:function_header<c_designator,

:function_formal_part<
>,                                                :void>
```

```
,                                  stantiation>;
generic_in

    generic_instantiation_decl := proc_ident_is
    generic_instantiation;
        :subprogram_declaration<:procedure_header<proc_ident_is,
```

```
:procedure_formal_part<>
>,                              generic_instantiation>;

    generic_instantiation := 'NEW'
    ambig_entry_subprogram_call_statement;
        %{
     VTP_TreeP tree = Parser_Pop();
     VTP_TreeP tr;
     tr = TreeMake2(instantiation, Ada_Disown(tree, 0),
                TreeRename(Ada_Disown(tree, 1),
actual_parameter_part));
     Parser_SetCoordTN(tr, $1, tree);
     VTP_TreeDestroy(tree);
        Parser_PopUntilToken($1
     ); Parser_Push(tr);
     }%   ;

    generic_instantiation := 'NEW' operator
    actual_parameter_part_option;
        :instantiation<operator,

actual_parameter_part_option>; end chapter;

end chapter;

chapter ' 13 : REPRESENTATION ISSUES '

chapter ' 13.1 REPRESENTATION ITEMS '

rules

    representation_clause := attribute_definition_clause;
        attribute_definition_clause;

    representation_clause := record_representation_clause;
        record_representation_clause;

    representation_clause := at_clause;
        at_clause;

end chapter;

chapter
' 13.3 REPRESENTATION ATTRIBUTES & 13.4 ENUMERATION REPRESENATION
CLAUSES '

rules

    attribute_definition_clause := 'FOR' name 'USE' expression;
        :attribute_definition_clause<name, expression>;

end chapter;

chapter ' 13.5 RECORD LAYOUT '
```

```
rules

    record_representation_clause :=
        'FOR' name 'USE' 'RECORD' mod_clause_option
```

```
            component_clause_list 'END' 'RECORD';
        :record_representation_clause<name, mod_clause_option,
                                       component_clause_list>;

    component_clause_list :=;
        :component_clauses_list<>;

    component_clause_list :=
component_clause_list component_name_location
";";
        component_clause_list:<.., component_name_location>;

    component_name_location := name 'AT' expression
    range_constraint;
        :component_clause<name, expression,

range_constraint>; end chapter;

chapter ' 13.8 MACHINE CODE INSERTIONS '

rules

    code_statement := qualified_expression;
        %{
        VTP_TreeP tr = Parser_Pop();
     tr = TreeRename(tr, assembly_code);
     Parser_Push(tr);
     }%;

end chapter;


end chapter;

chapter ' J.7 AT CLAUSES '

rules

    at_clause := 'FOR' name 'USE' 'AT' expression;
        :at_clause<name, expression>;

end chapter;

chapter ' J.8 MOD CLAUSES '

rules

    mod_clause_option :=;
        :void;

    mod_clause_option := 'AT' 'MOD' expression ";";
        expression;

end chapter;
```

```
chapter

ENTRY_POINTS rules

    phylum := '[ABSTRACT]' abstract_option;
```

```
    abstract_option;

phylum := '[ACCEPT]' accept_statement;
    accept_statement;

phylum := '[ACTUAL]' expression;
    expression;

phylum := '[AGGREGATE]' meta;
    meta;

phylum := '[AGGREGATE]' aggregate;
    aggregate;

phylum := '[AGGREGATE]' "(" expression ")";
    :record_or_array_aggregate<expression>;

phylum := '[ALIASED]';
    :void;

phylum := '[ALIASED]' 'ALIASED';
    :aliased;

phylum := '[ALLOCATOR]' allocator;
    allocator;

phylum := '[ALTERNATIVE]' meta;
    meta;

phylum := '[ALTERNATIVE]' case_statement_alternative;
    case_statement_alternative;

phylum := '[ALTERNATIVE]'
    pragma; pragma;

phylum := '[ALTERNATIVE_S]'
    meta; meta;

phylum := '[ALTERNATIVE_S]'
    case_statement_alternative_list;
    case_statement_alternative_list;

phylum := '[ATTRIBUTE]'
    attribute; attribute;

phylum := '[BLOCK_LOOP]' unnamed_block_statement;
    unnamed_block_statement;

phylum := '[BLOCK_LOOP]' unnamed_loop_statement;
    unnamed_loop_statement;

phylum := '[BLOCK_STUB]' package_block
    'END'; package_block;

phylum := '[BLOCK_STUB]' package_block 'END' ";";
    package_block;
```

```
phylum := '[BLOCK_STUB]' 'SEPARATE';
    :stub;
```

```
phylum := '[CALL]' ambig_entry_subprogram_call_statement
    ";"; ambig_entry_subprogram_call_statement;

phylum := '[CHOICE]' choice;
    choice;

phylum := '[CHOICE_S]'
    choice_list; choice_list;

phylum := '[COMPILATION]' meta;
    meta;

phylum := '[COMPILATION]' compilation_unit_list;
    compilation_unit_list;

phylum := '[COMPOUND_DESIGNATOR]' c_designator;
    c_designator;

phylum := '[COMPOUND_NAME]' compound_name;
    compound_name;

phylum := '[COMP]' meta;
    meta;

phylum := '[COMP]' component;
    component;

phylum := '[COMP]' pragma ";";
    pragma;

phylum := '[COMP_ASSOC]'
    expression; expression;

phylum := '[COMP_ASSOC]' named_component;
    named_component;

phylum := '[COMP_REP]'
    meta; meta;

phylum := '[COMP_REP]' component_name_location ";";
    component_name_location;

phylum := '[COMP_REP_S]' component_clause_list;
    component_clause_list;

phylum := '[COMP_UNIT]' meta;
    meta;

phylum := '[COMP_UNIT]' compilation_unit ";";
    compilation_unit;

phylum := '[COND_CLAUSE]' meta;
    meta;

phylum := '[COND_CLAUSE]' if_then_part;
```

```
if_then_part;
```

```
    phylum := '[COND_CLAUSE]' elsif_part;
        elsif_part;

    phylum := '[COND_CLAUSE]' 'ELSE' statement_list;
        :conditional_clause<:void, statement_list>;

    phylum := '[CONSTRAINED]'
        subtype_indication;
        subtype_indication;

    phylum := '[CONSTRNT]' meta;
        meta;

    phylum := '[CONSTRNT]';
        :void;

    phylum := '[CONSTRNT]'
        constraint; constraint;

    phylum := '[CONTEXT]'
        meta; meta;

    phylum := '[CONTEXT]' context_specif_option;
        context_specif_option;

    phylum := '[CONT_ELEM]' meta;
        meta;

    phylum := '[CONT_ELEM]' use_clause
        ";"; use_clause;

    phylum := '[CONT_ELEM]'
        with_clause; with_clause;

    phylum := '[CONT_ELEM]' pragma ";";
        pragma;

    phylum := '[DECL]' meta;
        meta;

    phylum := '[DECL]' declarative_item
        ";"; declarative_item;

    phylum := '[DECL_S]' declarative_item_option_list;
        declarative_item_option_list;

    phylum := '[DELAY]' delay_statement;
        delay_statement;

    phylum := '[DELAY_ALTERNATIVE]' delay_statement ";"
statement_option_list;
        :delay_alternative<delay_statement,
        statement_option_list>;

    phylum := '[DESIGNATOR]' designator;
        designator;
```

```
phylum := '[DSCRT_RANGE]' discrete_range;
    discrete_range;
```

```
phylum := '[ENTRY_BLOCK]' entry_block;
    entry_block;

phylum := '[ENTRY_INDEX_VOID]';
    :void;

phylum := '[ENTRY_INDEX_VOID]' "(" entry_index ")";
    entry_index;

phylum := '[ENTRY_NAME]' identifier;
    identifier;

phylum := '[ENTRY_NAME]'
    indexed_entry_name;
    indexed_entry_name;

phylum := '[ENUM_LITERAL]' character_literal;
    character_literal;

phylum := '[ENUM_LITERAL]'
    identifier; identifier;

phylum := '[EXC_ALTERNATIVE_S_VOID]' meta;
    meta;

phylum := '[EXC_ALTERNATIVE_S_VOID]';
    :void;

phylum := '[EXC_ALTERNATIVE_S_VOID]'
    case_statement_alternative_list;
    case_statement_alternative_list;

phylum := '[EXCEPTION_DEF]' meta;
    meta;

phylum := '[EXCEPTION_DEF]' 'RENAMES' rename;
    rename;

phylum := '[EXCEPTION_DEF]';
    :void;

phylum := '[EXP]' expression;
    expression;

phylum := '[EXP_S]' "(" expression_list ")";
    expression_list;

phylum := '[EXP_RANGE]' 'MOD' expression;
    expression;

phylum := '[EXP_RANGE]'
    range_constraint;
    range_constraint;

phylum := '[EXP_VOID]' expression;
    expression;
```

```
phylum := '[EXP_VOID]';
    :void;

phylum := '[EXTENSION_VOID]' ;
```

```
        :void;

    phylum := '[EXTENSION_VOID]' 'WITH' 'PRIVATE';
        :with_private;

    phylum := '[EXTENSION_VOID]' 'WITH' record_definition;
        record_definition;

    phylum := '[FCT_NAME]' operator;
        operator;

    phylum := '[FCT_NAME]'
        name; name;

    phylum := '[FCT_NAME]' character_literal;
        character_literal;

    phylum := '[FCT_PARAM_S]' meta;
        meta;

    phylum := '[FCT_PARAM_S]' fct_formal_part_option;
        %{
        VTP_TreeP tr = Parser_Pop();
    tr = TreeRename(tr, function_formal_part);
        Parser_PopUntilToken($1);
      Parser_Push(tr);
      }%;

    phylum := '[FORM_TYPE_SPEC]' meta;
        meta;

    phylum := '[FORM_TYPE_SPEC]' private_type_definition;
        private_type_definition;

    phylum := '[FORM_TYPE_SPEC]' array_type_definition;
        array_type_definition;

    phylum := '[FORM_TYPE_SPEC]' access_type_definition;
        access_type_definition;

    phylum := '[FORM_TYPE_SPEC]' formal_type_spec;
        formal_type_spec;

    formal_type_spec := "(" "<>" ")";
        :generic_formal_discrete_type;

    formal_type_spec := 'DELTA' "<>";
        :generic_formal_fixed_point_type;

    formal_type_spec := 'DELTA' "<>" 'DIGITS' "<>";
        :generic_formal_decimal_fixed_point_type;

    formal_type_spec := 'DIGITS' "<>";
        :generic_formal_floating_point_type;

    formal_type_spec := 'RANGE' "<>";
```

```
:generic_formal_integer_type;
```

```
formal_type_spec := 'MOD' "<>";
    :generic_formal_modular_type;

phylum := '[GENERIC_HEADER]' subprogram_declaration;
    subprogram_declaration;

phylum := '[GENERIC_HEADER]' package_specification;
    package_specification;

phylum := '[GENERIC_PARAM]'
    meta; meta;

phylum := '[GENERIC_PARAM]' generic_parameter ";";
    generic_parameter;

phylum := '[GENERIC_PARAM_S]'
    meta; meta;

phylum := '[GENERIC_PARAM_S]'
    generic_parameter_list_option;
    generic_parameter_list_option;

phylum := '[HEADER]' meta;
    meta;

phylum := '[HEADER]' 'FUNCTION'
    function_header; function_header;

phylum := '[HEADER]' 'PROCEDURE' compound_name;
    :procedure_header<compound_name,
    :procedure_formal_part<>>;

phylum := '[HEADER]' 'PROCEDURE' procedure_header;
    procedure_header;

phylum := '[ID]' identifier;
    identifier;

phylum := '[ID_S]' identifier_list;
    identifier_list;

phylum := '[ID_VOID]' ;
    :void;

phylum := '[ID_VOID]' identifier ":";
    identifier;

phylum := '[IN]' meta;
    meta;

phylum := '[IN]' in_parameter_declaration;
    in_parameter_declaration;

phylum := '[INDEX_DEF]' discrete_range;
    discrete_range;
```

```
phylum := '[INDEX_DEF]' index;
    index;

phylum := '[INDEX_DEF_S]' "(" gen_discrete_range_list ")";
```

```
    gen_discrete_range_list;

phylum := '[INDEX_DEF_S]' "(" index_subtype_definition_list
    ")"; index_subtype_definition_list;

phylum := '[ITEM]' meta;
    meta;

phylum := '[ITEM]' declarative_part_item ";";
    declarative_part_item;

phylum := '[ITEM]' entry_declaration;
    entry_declaration;

phylum := '[ITEM_S]' declarative_part_list;
    declarative_part_list;

phylum := '[ITERATION]' meta;
    meta;

phylum := '[ITERATION]' iteration_sheme;
    iteration_sheme;

phylum := '[LIMITED]' ;
    :void;

phylum := '[LIMITED]' 'LIMITED';
    :limited;


phylum := '[MODIFIER]' access_to_object_option;
    access_to_object_option;

phylum := '[NAME]' name;
    name;

phylum := '[NAME_RANGE]' name;
    name;

phylum := '[NAME_RANGE]'
    range1; range1;

phylum := '[NAME_VOID]' name;
    name;

phylum := '[NAME_VOID]';
    :void;

phylum := '[OBJECT_DEF]'
    expression; expression;

phylum := '[OBJECT_DEF]';
    :void;

phylum := '[OBJECT_DEF]' 'RENAMES' rename;
```

```
    rename;

phylum := '[OBJECT_QUALIFIER]' 'ACCESS';
```

```
    :access;

phylum := '[OBJECT_QUALIFIER]' qualifier_option;
    qualifier_option;

phylum := '[OBJECT_TYPE]' array_type_definition;
    array_type_definition;

phylum := '[OBJECT_TYPE]' subtype_indication;
    subtype_indication;

phylum := '[OBJ]' meta;
    meta;

phylum := '[OBJ]' object_declaration;
    object_declaration;

phylum := '[OBJ_S]' meta;
    meta;

phylum := '[OBJ_S]' "(" meta ")";
    :discriminant_part<meta>;

phylum := '[OBJ_S]' "(" "<>" ")";
    :unknown_discriminant;

phylum := '[OBJ_S]';
    :discriminant_part<>;

phylum := '[OBJ_S]' "(" discriminant_specification_list
    ")"; discriminant_specification_list;

phylum := '[PACK_DEF]' generic_instantiation;
    generic_instantiation;

phylum := '[PACK_DEF]' 'NEW' name "(" "<>" ")";
    :formal_package_box<name>;

phylum := '[PACK_DEF]' package_specif_body;
    package_specif_body;

phylum := '[PACK_DEF]' 'RENAMES' rename;
    rename;

phylum := '[PACK_SPEC]' package_specif_body;
    package_specif_body;

phylum := '[PARAM]' meta;
    meta;

phylum := '[PARAM]' any_parameter_declaration;
    any_parameter_declaration;

phylum := '[PARAM_ASSOC]'
    expression; expression;
```

```
phylum := '[PARAM_ASSOC]' parameter_association;
    parameter_association;
```

```
phylum := '[PARAM_ASSOC_S]';
    :actual_parameter_part<>;

phylum := '[PARAM_ASSOC_S]' "(" parameter_association_list ")";
    parameter_association_list;

phylum := '[PARAM_ASSOC_S]' "(" expression_list ")";
    %{
    VTP_TreeP tr = Parser_Pop();
  tr = TreeRename(tr, actual_parameter_part);
    Parser_PopUntilToken($1);
  Parser_Push(tr);
  }%;

phylum := '[PARAM_S]' meta;
    meta;

phylum := '[PARAM_S]' "(" meta ")";
    :procedure_formal_part<meta>;

phylum := '[PARAM_S]';
    :procedure_formal_part<>;

phylum := '[PARAM_S]' "(" parameter_specification_list ")";
    parameter_specification_list;

phylum := '[PRAGMA]' pragma ";";
    pragma;

phylum := '[PRAGMAS]' pragma_option_list;
    pragma_option_list;

phylum := '[PREFIX]' operator;
    operator;

phylum := '[PREFIX]' name;
    name;

phylum := '[PREFIX]'
    function_call;
    function_call;

phylum := '[PRIVATE_VOID]' private_unit_option;
    private_unit_option;

phylum := '[PROT_DEF]'
    protected_definition;
    protected_definition;

phylum := '[PROT_ELEM]'
    protected_element_declaration;
    protected_element_declaration;

phylum := '[PROT_ITEM]'
```

```
    protected_operation_item;
    protected_operation_item;

phylum := '[PROT_OPER]'
    protected_operation_declaration;
    protected_operation_declaration;

phylum := '[PROT_ELEM_S]'
protected_element_declaration_option_list;
```

```
        protected_element_declaration_option_list;

    phylum := '[PROT_ITEM_S_STUB]'
        protected_operation_item_option_list;
        protected_operation_item_option_list;

    phylum := '[PROT_ITEM_S_STUB]' 'SEPARATE';
        :stub;

    phylum := '[PROT_OPER_S]'
protected_operation_declaration_option_l
ist;
        protected_operation_declaration_option_list;

    phylum := '[PROTECTED]' access_to_subprogram_option;
        access_to_subprogram_option;

    phylum := '[VOID_DSCRT_RANGE]' discrete_range;
        discrete_range;

    phylum := '[VOID_DSCRT_RANGE]';
        :void;

    phylum := '[VOID_WITH_ABORT]' with_abort;
        with_abort;

    phylum := '[VOID_WITH_ABORT]';
        :void;

    phylum := '[RANGE]' meta;
        meta;

    phylum := '[RANGE]' range;
        range;

    phylum := '[RANGE]' range_constraint;
        range_constraint;

    phylum := '[RANGE_VOID]' meta;
        meta;

    phylum := '[RANGE_VOID]' range;
        range;

    phylum := '[RANGE_VOID]';
        :void;

    phylum := '[RECORD_OR_ARRAY_AGGREGATE]' "(" aggregate_list ")"
        ; aggregate_list;

    phylum := '[RECORD_OR_ARRAY_AGGREGATE]' "(" 'NULL' 'RECORD' ")"
    ;
        :null_record_aggregate;

    phylum := '[RECORD_OR_ARRAY_AGGREGATE]' aggregate_list ;
```

```
    aggregate_list;

phylum := '[RECORD_OR_ARRAY_AGGREGATE]' 'NULL' 'RECORD' ;
    :null_record_aggregate;

phylum := '[RECORD_DEF]' component_list;
```

```
    component_list;

phylum := '[RECORD_DEF]' record_definition;
    record_definition;

phylum := '[REP]' meta;
    meta;

phylum := '[REP]' representation_clause;
    representation_clause;

phylum := '[SELECTOR]' selector;
    selector;

phylum := '[SELECT_ALTERNATIVE]'
    select_alternative; select_alternative;

phylum := '[SELECT_CLAUSE]'
    meta; meta;

phylum := '[SELECT_CLAUSE]' guarded_select_alternative;
    guarded_select_alternative;

phylum := '[SELECT_CLAUSE_S]' meta;
    meta;

phylum := '[SELECT_CLAUSE_S]' select_alternative_list;
    select_alternative_list;

phylum := '[STM]' meta;
    meta;

phylum := '[STM]' statement ";";
    statement;

phylum := '[STM_S_VOID]' meta;
    meta;

phylum := '[STM_S_VOID]';
    :void;

phylum := '[STM_S_VOID]' statement_list;
    statement_list;

phylum := '[STM_S]' meta;
    meta;

phylum := '[STM_S]' statement_list;
    statement_list;

phylum := '[SUBPROGRAM_DEF]'
    operator; operator;

phylum := '[SUBPROGRAM_DEF]' name;
    name;
```

```
phylum := '[SUBPROGRAM_DEF]' character_literal;
    character_literal;
```

```
phylum := '[SUBPROGRAM_DEF]';
    :void;

phylum := '[SUBPROGRAM_DEF]' 'IS' 'ABSTRACT';
    :is_abstract;

phylum := '[SUBPROGRAM_DEF]' "<>";
    :box;

phylum := '[SUBPROGRAM_DEF]' generic_instantiation;
    generic_instantiation;

phylum := '[SUBPROGRAM_DEF]' 'RENAMES'
    rename; rename;

phylum := '[SUBUNIT_BODY]'
    meta; meta;

phylum := '[SUBUNIT_BODY]' proper_body ";";
    proper_body;

phylum := '[SUB_PARAM_S]' 'PROCEDURE'
    formal_part_option; formal_part_option;

phylum := '[SUB_PARAM_S]' 'FUNCTION'
    fct_formal_part_option; fct_formal_part_option;

phylum := '[TAGGED]' tagged_option;
    tagged_option;

phylum := '[TASK_DEF]'
    meta; meta;

phylum := '[TASK_DEF]' 'RENAMES' rename;
    rename;

phylum := '[TASK_DEF]' task_definition_option;
    task_definition_option;

phylum := '[TASK_ITEM]' meta;
    meta;

phylum := '[TASK_ITEM]' representation_clause;
    representation_clause;

phylum := '[TASK_ITEM]' entry_declaration ";";
    entry_declaration;

phylum := '[TASK_ITEM_S]' task_items_option_list;
    task_items_option_list;

phylum := '[TRIG_STM]' entry_call;
    entry_call;

phylum := '[TRIG_STM]' delay_statement;
```

```
delay_statement;
```

```
phylum := '[TYPE_RANGE]' discrete_range;
    discrete_range;

phylum := '[TYPE_SPEC]' private_type_definition;
    private_type_definition;

phylum := '[TYPE_SPEC]' type_definition;
    type_definition;

phylum :=
    '[TYPE_SPEC]' 'IS' task_items_option_list 'PRIVATE'
        task_items_option_list 'END' identifier_option;
    %{
    VTP_TreeP opt_tr =
  Parser_Pop(); VTP_TreeP
  tr_item1 = Parser_Pop();
  VTP_TreeP tr_item2 =
  Parser_Pop();
  if (CheckOper(tr_item1, task_items_list) &&
      (VTP_TreeLength(tr_item1) == 0) &&
      CheckOper(tr_item2,
      task_items_list) &&
      (VTP_TreeLength(tr_item2) == 0)) {
      Parser_Push(TreeMake0(void,
      tr_item1));
  } else {
      Parser_Push(TreeMake2(task_specification, tr_item1,
      tr_item2));
  }
    Parser_PopUntilToken($1);
  }%;

phylum := '[UNIT_ITEM]' meta;
    meta;

phylum := '[UNIT_ITEM]' unit_declaration

    ";"; unit_declaration;

phylum := '[UNIT_ITEM]' unit_body ";";
    unit_body;

phylum := '[VARIANT]' meta;
    meta;

phylum := '[VARIANT]' variant;
    variant;

phylum := '[VARIANT_S]' meta;
    meta;

phylum := '[VOID_VARIANT_PART]';
    :void;

phylum := '[VOID_VARIANT_PART]'
```

```
        variant_part; variant_part;

    phylum := '[VARIANT_S]' variant_list;
        variant_list;

rules

    phylum := '[IF]' meta;
```

```
        meta;

    phylum := '[IF]' meta 'END' 'IF'
        ";"; meta;

    phylum := '[IF]' if_statement 'END' 'IF' ";";
        if_statement;

    phylum := '[ENUM_LITERAL_S]' enumeration_type_definition;
        enumeration_type_definition;

    phylum := '[USE]' use_clause;
        use_clause;

    phylum := '[WITH]' with_clause;
        with_clause;

    phylum := '[ABORT]'
        abort_statement;
        abort_statement;

end chapter;

chapter

PROPERTIES

abstract syntax

    TOPOP := compilation comp_unit ITEM EXP NAME
    STM; VOID :=
        void others null_statement stub
        component_clauses_list components_list
        context_clause
        basic_declarative_part task_items_list
        function_formal_part generic_formal_part declarative_part
        actual_parameter_part procedure_formal_part
        discriminant_part;

frames
    prefix -> implemented as tree;
        controls copy save;
    postfix -> implemented as
    tree;
        controls copy save;
    focus -> implemented as integer;
        controls copy;

end chapter;

chapter ' Atomic Constructors '

abstract syntax

    identifier -> implemented as string;
```

```
'abstract' -> implemented as void;
abstract_tagged -> implemented as void;
access -> implemented as void;
aliased -> implemented as void;
aliased_constant -> implemented as
void; all -> implemented as void;
box -> implemented as void;
character_literal -> implemented as
string; constant -> implemented as
void;
```

```
    generic_formal_discrete_type -> implemented as void;
    generic_formal_fixed_point_type -> implemented as void;
    generic_formal_decimal_fixed_point_type -> implemented as
    void; generic_formal_floating_point_type -> implemented
    as void; generic_formal_integer_type -> implemented as
    void; generic_formal_modular_type -> implemented as void;
    is_abstract -> implemented as void;
    limited -> implemented as void;
    null_access_value -> implemented as void;
    null_component -> implemented as void;
    null_record -> implemented as void;
    null_record_aggregate -> implemented as
    void; null_statement -> implemented as
    void; numeric_literal -> implemented as
    string; operator -> implemented as string;
    others -> implemented as void;
    private_unit -> implemented as void;
    protected -> implemented as void;
    string_literal -> implemented as
    string; stub -> implemented as void;
    tagged -> implemented as void;
    terminate_alternative -> implemented as
    void; unknown_discriminant -> implemented
    as void; void -> implemented as void;
    with_abort -> implemented as void;
    with_private -> implemented as void;

end chapter;

chapter ' Unary Constructors '

abstract syntax

    abs -> EXP;
    allocator -> ALLOCATOR;
    delay_relative_statement -> EXP;
    delay_until_statement -> EXP;
    formal_package_box -> NAME;
    goto_statement -> NAME;
    index_subtype_definition -> NAME;
    integer_type -> EXP_RANGE;
    not -> EXP;
    parenthesis -> EXP;
    raise_statement ->
    NAME_VOID; renaming ->
    FCT_NAME; return_statement
    -> EXP_VOID; unary_minus -
    > EXP; unary_plus -> EXP;
    while -> EXP;

end chapter;

chapter ' Binary Constructors '
```

```
abstract syntax

    accept_alternative -> ACCEPT STM_S_VOID;
    access_to_object_type -> MODIFIER CONSTRAINED;
```

```
addition -> EXP EXP;
at_clause -> NAME EXP;
alternative -> CHOICE_S
STM_S; and -> EXP EXP;
and_then -> EXP EXP;
indexed_component_or_function_call -> FCT_NAME
PARAM_ASSOC_S; assignment_statement -> NAME EXP;
named_association -> CHOICE_S ACTUAL;
procedure_or_entry_call_statement -> NAME PARAM_ASSOC_S;
case_statement -> EXP ALTERNATIVE_S;
catenation -> EXP EXP;
assembly_code -> NAME AGGREGATE;
comp_unit -> CONTEXT PRIVATE_VOID UNIT_ITEM;
conditional_clause -> EXP_VOID STM_S;
subtype_indication -> NAME CONSTRNT;
delay_alternative -> DELAY STM_S_VOID;
different -> EXP EXP;
division -> EXP EXP;
entry_index -> ID
DSCRT_RANGE; equal -> EXP
EXP; exponentiation -> EXP
EXP;
exception_declaration -> ID_S EXCEPTION_DEF;
exit_statement -> NAME_VOID EXP_VOID;
extension_aggregate -> EXP
RECORD_OR_ARRAY_AGGREGATE; fixed_point_constraint -
> EXP RANGE_VOID; floating_point_constraint -> EXP
RANGE_VOID;
for -> ID DSCRT_RANGE;
function_call -> FCT_NAME PARAM_ASSOC_S;
generic_declaration -> GENERIC_PARAM_S
GENERIC_HEADER; greater -> EXP EXP;
greater_equal -> EXP EXP;
indexed_component -> PREFIX EXP_S;
instantiation -> FCT_NAME
PARAM_ASSOC_S; in_out_parameter ->
ID_S NAME; labeled_statement ->
DESIGNATOR STM; less -> EXP EXP;
less_equal -> EXP EXP;
loop_statement -> ITERATION
STM_S; member -> EXP NAME_RANGE;
modulus -> EXP EXP;
multiplication -> EXP EXP;
named_statement -> DESIGNATOR BLOCK_LOOP;
not_member -> EXP NAME_RANGE;
number_declaration -> ID_S
EXP; or -> EXP EXP;
or_else -> EXP EXP;
out_parameter -> ID_S NAME;
package_body -> COMPOUND_NAME BLOCK_STUB;
package_declaration -> COMPOUND_NAME
PACK_DEF; package_specification -> DECL_S
DECL_S;  pragma -> ID PARAM_ASSOC_S;
private_type -> TAGGED LIMITED;
```

```
protected_body -> ID
PROT_ITEM_S_STUB;
protected_declaration -> ID
PROT_DEF;
protected_specification -> PROT_OPER_S PROT_ELEM_S;
subprogram_declaration -> HEADER SUBPROGRAM_DEF;
subprogram_body -> HEADER BLOCK_STUB;
procedure_header -> COMPOUND_NAME PARAM_S;
```

```
    qualified_expression -> NAME
    EXP; range -> EXP EXP;
    remainder -> EXP EXP;
    requeue_statement -> NAME
    VOID_WITH_ABORT; reverse -> ID
    DSCRT_RANGE; selected_component ->
    PREFIX SELECTOR;
    select_clause -> EXP_VOID
    SELECT_ALTERNATIVE; selective_accept ->
    SELECT_CLAUSE_S STM_S_VOID; slice -> PREFIX
    DSCRT_RANGE;
    substraction -> EXP EXP;
    subtype_declaration -> ID CONSTRAINED;
    subunit -> FCT_NAME SUBUNIT_BODY;
    task_body -> ID BLOCK_STUB;
    task_declaration -> ID TASK_DEF;
    task_specification -> TASK_ITEM_S TASK_ITEM_S;
    variant -> CHOICE_S RECORD_DEF;
    xor -> EXP EXP;

end chapter;

chapter ' Ternary Constructors

' abstract syntax

    accept_statement -> ENTRY_NAME PARAM_S STM_S_VOID;
    access_parameter -> ID_S NAME EXP_VOID;
    access_to_subprogram_type -> PROTECTED SUB_PARAM_S
    NAME_VOID; array -> INDEX_DEF_S ALIASED CONSTRAINED;
    asynchronous_select -> TRIG_STM STM_S_VOID STM_S;
    attribute -> PREFIX ID EXP_VOID;
    attribute_definition_clause -> ATTRIBUTE EXP;
    block_statement -> ITEM_S STM_S
    EXC_ALTERNATIVE_S_VOID; component_clause -> NAME EXP
    RANGE; conditional_entry_call -> CALL STM_S_VOID
    STM_S; decimal_fixed_point_constraint -> EXP EXP
    RANGE_VOID; derived_type -> ABSTRACT CONSTRAINED
    EXTENSION_VOID;
    entry_declaration -> ID VOID_DSCRT_RANGE PARAM_S;
    exception_alternative -> ID_VOID CHOICE_S STM_S;
    function_header -> COMPOUND_DESIGNATOR FCT_PARAM_S
    NAME_VOID; generic_type -> ID OBJ_S FORM_TYPE_SPEC;
    parameter -> ID_S NAME EXP_VOID;
    in_parameter -> ID_S NAME EXP_VOID;
    record_representation_clause -> NAME EXP_VOID
    COMP_REP_S; record_type -> TAGGED LIMITED
    RECORD_DEF; timed_entry_call -> CALL STM_S_VOID
    DELAY_ALTERNATIVE; type_declaration -> ID OBJ_S
    TYPE_SPEC;
    variant_part -> DESIGNATOR PRAGMAS

VARIANT_S; end chapter;
```

```
chapter ' Other Constructors '

abstract syntax

    entry_body -> ID ENTRY_INDEX_VOID PARAM_S EXP ENTRY_BLOCK;
    object_declaration -> ID_S OBJECT_QUALIFIER OBJECT_TYPE

OBJECT_DEF; end chapter;
```

```
chapter ' List Constructors '

abstract syntax

    abort_statement -> NAME +;
    record_or_array_aggregate -> COMP_ASSOC +;
    alternatives_list -> ALTERNATIVE +;
    exception_alternatives_list -> EXC_ALTERNATIVE
    +; choices_list -> CHOICE +;
    compilation -> COMP_UNIT +;
    component_clauses_list -> COMP_REP
    *; components_list -> COMP *;
    compound_name -> ID +;
    context_clause -> CONT_ELEM *;
    basic_declarative_part -> DECL *;
    index_constraint -> DSCRT_RANGE +;
    enumeration_type_definition -> ENUM_LITERAL +;
    expressions_list -> EXP +;

    function_formal_part -> IN *;
    generic_formal_part -> GENERIC_PARAM
    *; identifiers_list -> ID +;
    if_statement -> COND_CLAUSE +;
    index_definitions_list -> INDEX_DEF
    +; declarative_part -> ITEM *;
    actual_parameter_part -> PARAM_ASSOC
    *; pragmas -> PRAGMA *;
    procedure_formal_part -> PARAM *;
    protected_element_declarations_list -> PROT_ELEM
    *; protected_operation_declarations_list ->
    PROT_OPER *; protected_operation_items_list ->
    PROT_ITEM *; select_clauses_list -> SELECT_CLAUSE
    +; statements_list -> STM +;
    task_items_list -> TASK_ITEM *;
    use_package_clause -> NAME +;
    use_type_clause -> NAME +;
    variants_list -> VARIANT +;
    discriminant_part -> OBJ *;
    with_clause -> COMPOUND_NAME +;

end chapter;

chapter ' Phyla '

abstract syntax

    ABSTRACT := 'abstract';
    ACCEPT := accept_statement;
    ACTUAL := EXP operator;
    AGGREGATE := null_record_aggregate
        record_or_array_aggregate extension_aggregate;
    ALIASED := aliased void;
```

```
ALLOCATOR := qualified_expression subtype_indication;
ALTERNATIVE := alternative pragma;
ALTERNATIVE_S := alternatives_list;
ATTRIBUTE := attribute;
```

```
    BLOCK_LOOP := block_statement loop_statement;
    BLOCK_STUB := block_statement stub;
    BOOLEAN :=
        not and and_then or or_else xor less greater
less_equal greater_equal
        equal different member not_member attribute
        function_call identifier indexed_component
        qualified_expression selected_component;
    CALL                                :=
    procedure_or_entry_call_statement;
    CHOICE  :=  DSCRT_RANGE  EXP  operator
    others; CHOICE_S := choices_list;
    COMP := null_component object_declaration REP variant_part
    pragma; COMPILATION := compilation;
    COMPOUND_DESIGNATOR := compound_name operator;
    COMPOUND_NAME := compound_name;
    COMP_ASSOC := EXP
    named_association; COMP_REP :=
    component_clause; COMP_REP_S :=
    component_clauses_list; COMP_UNIT
    := comp_unit pragma; COND_CLAUSE
    := conditional_clause; CONSTRAINED
    := subtype_indication; CONSTRNT :=
        AGGREGATE attribute index_constraint
        fixed_point_constraint floating_point_constraint range
        void;
    CONTEXT := context_clause;
    CONT_ELEM := use_package_clause use_type_clause with_clause
    pragma; DECL :=
        REP object_declaration exception_declaration
        generic_declaration number_declaration package_declaration
        pragma
subprogram_declaration
        subtype_declaration task_declaration type_declaration
use_type_clause
        use_package_clause protected_declaration;
    DECL_S := basic_declarative_part;
    DELAY := delay_relative_statement delay_until_statement;
    DELAY_ALTERNATIVE := delay_alternative;
    DESIGNATOR := identifier operator;
    DSCRT_RANGE := subtype_indication range attribute;
    VOID_DSCRT_RANGE := DSCRT_RANGE void;
    ENTRY_BLOCK := block_statement;
    ENTRY_INDEX_VOID := entry_index void;
    ENTRY_NAME := identifier
    indexed_component; ENUM_LITERAL :=
    character_literal identifier;
    EXCEPTION_DEF := renaming void;
    EXC_ALTERNATIVE := exception_alternative pragma;
    EXC_ALTERNATIVE_S_VOID := exception_alternatives_list
    void; EXP :=
        parenthesis indexed_component_or_function_call BOOLEAN
        NUMERIC AGGREGATE allocator catenation character_literal
null_access_value
```

```
    slice string_literal;
EXP_RANGE := EXP range;
EXP_VOID := EXP void;
EXP_S := expressions_list;
EXTENSION_VOID := RECORD_DEF with_private void;
FCT_NAME :=
    indexed_component_or_function_call DESIGNATOR
    character_literal attribute indexed_component
    selected_component slice;
```

```
    FCT_PARAM_S := function_formal_part;
    FORM_TYPE_SPEC :=
        generic_formal_discrete_type
        generic_formal_fixed_point_type
        generic_formal_floating_point_type
        generic_formal_integer_type generic_formal_modular_type
generic_formal_decimal_fixed_point_type
        access_to_object_type access_to_subprogram_type
        array
private_type;
    GENERIC_PARAM := parameter in_parameter in_out_parameter
        subprogram_declaration package_declaration generic_type;
    GENERIC_HEADER := subprogram_declaration package_declaration;
    GENERIC_PARAM_S := generic_formal_part;
    HEADER := function_header
    procedure_header; ID := identifier;
    ID_S := identifiers_list;
    ID_VOID := identifier
    void; IN := parameter
    in_parameter;
    INDEX_DEF := DSCRT_RANGE index_subtype_definition;
    INDEX_DEF_S := index_definitions_list;
    ITEM :=
        REP object_declaration exception_declaration
        generic_declaration number_declaration package_body
        package_declaration pragma subprogram_body
        subprogram_declaration subtype_declaration
task_body
        task_declaration type_declaration use_type_clause
use_package_clause
        protected_declaration protected_body;
    ITEM_S := declarative_part;
    ITERATION := for reverse void
    while; LIMITED := limited void;
    MODIFIER := all constant void;
    NAME :=
        indexed_component_or_function_call identifier attribute
        indexed_component selected_component slice;
    NAME_RANGE := NAME RANGE;
    NAME_VOID := NAME void;
    NUMERIC :=
        abs modulus remainder unary_plus unary_minus addition
substraction
        multiplication division exponentiation attribute
        function_call identifier indexed_component
        numeric_literal qualified_expression
        selected_component;
    OBJECT_DEF := EXP_VOID renaming;
    OBJECT_QUALIFIER := aliased constant aliased_constant access
    void; OBJECT_TYPE := array subtype_indication;
    OBJ := object_declaration;
    OBJ_S := discriminant_part unknown_discriminant;
    PACK_DEF := instantiation package_specification
        formal_package_box renaming;
```

```
PACK_SPEC := package_specification;
PARAM := parameter in_parameter access_parameter
    in_out_parameter out_parameter;
PARAM_ASSOC := ACTUAL
named_association; PARAM_ASSOC_S :=
actual_parameter_part; PARAM_S :=
procedure_formal_part; PRAGMAS :=
pragmas;
PRAGMA := pragma;
PRIVATE_VOID := private_unit void;
```

```
    PROT_DEF := protected_specification;
    PROT_ELEM := PROT_OPER COMP;
    PROT_ITEM := subprogram_declaration subprogram_body entry_body
REP pragma;
    PROT_OPER := subprogram_declaration entry_declaration REP
    pragma;
    PROT_ELEM_S := protected_element_declarations_list;
    PROT_ITEM_S_STUB := protected_operation_items_list stub;
    PROT_OPER_S := protected_operation_declarations_list;
    PROTECTED := protected void;
    PREFIX := FCT_NAME
    function_call; RANGE := range
    attribute; RANGE_VOID := RANGE
    void;
    RECORD_OR_ARRAY_AGGREGATE := record_or_array_aggregate
        null_record_aggregate;
    RECORD_DEF := null_record components_list;
    REP := at_clause record_representation_clause
attribute_definition_clause;
    SELECTOR := identifier operator character_literal all;
    SELECT_ALTERNATIVE :=
        accept_alternative delay_alternative terminate_alternative;
    SELECT_CLAUSE := select_clause;
    SELECT_CLAUSE_S :=
    select_clauses_list; STM :=
        abort_statement accept_statement assignment_statement
block_statement
        procedure_or_entry_call_statement case_statement
        assembly_code conditional_entry_call
        delay_relative_statement
delay_until_statement
        requeue_statement exit_statement goto_statement
        if_statement labeled_statement loop_statement
        named_statement
        null_statement raise_statement return_statement
        selective_accept timed_entry_call asynchronous_select
        pragma;
    STM_S := statements_list;
    STM_S_VOID := statements_list
    void;
    SUBPROGRAM_DEF := FCT_NAME box instantiation renaming
is_abstract void;
    SUBUNIT_BODY := package_body subprogram_body task_body
protected_body;
    SUB_PARAM_S := function_formal_part procedure_formal_part;
    TAGGED := tagged abstract_tagged void;
    TASK_DEF := renaming task_specification;
    TASK_ITEM := entry_declaration REP pragma;
    TASK_ITEM_S := task_items_list;
    TRIG_STM := CALL DELAY;
    TYPE_SPEC :=
        access_to_object_type access_to_subprogram_type array
derived_type
        enumeration_type_definition fixed_point_constraint
```

```
        decimal_fixed_point_constraint
        floating_point_constraint integer_type private_type
        record_type task_specification
        protected_specification void;
    UNIT_ITEM :=
        package_body subprogram_body
package_declaration subprogram_declaration
        generic_declaration subunit;
    VOID_WITH_ABORT := with_abort
    void; VARIANT := variant pragma;
    VARIANT_S := variants_list;
```

```
end chapter;

end
```

```
definition
```

3. C++

Abstract syntax

```
definition of C version 1 is

chapter ABSTRACT_SYNTAX

chapter TOP
  abstract syntax

  program -> EXTDEFS ;
  EXTDEFS := extdefs ;
  extdefs -> EXTDEF * ;
  EXTDEF := fndef extdecl empty_extdecl template_def
explicit_instanciation
          explicit_specialization  namespace_def
          using_declaration using_directive  asm extern_def
          macro_call PREPROCESSOR SQL_STMT ;
  asm -> STRING ;
  extern_def -> STRING EXTERN_DEFS ;
  EXTERN_DEFS := EXTDEFS EXTDEF ;
  STRING := string strings macro_call

; end chapter;

chapter DECLARATIONS
  abstract syntax

  empty_extdecl -> implemented as void ;

  extdecl -> DECLARATION_SPECIFIERS INITDECLS_OPT
  ; decl -> DECLARATION_SPECIFIERS INITDECLS_OPT ;
  INITDECLS_OPT := none initdecls ;
  initdecls -> INITDCL +;
  INITDCL := dcltr_noinit dcltr_affinit dcltr_callinit
  bit_field; dcltr_noinit -> DECLARATOR ;
  dcltr_affinit -> DECLARATOR INITIALIZER ;
  dcltr_callinit -> DECLARATOR EXPRLIST ;
  INITIALIZER := SIMPLE_EXPR initializer_list
  ; initializer_list -> INITIALIZER * ;

  DECLARATOR := parenth_dcltr func_dcltr array_dcltr ptr_dcltr
ref_dcltr memptr_dcltr
     QUALIFIED_ID ;
  ABSDCLTR := parenth_dcltr func_dcltr array_dcltr ptr_dcltr
ref_dcltr memptr_dcltr
     absdcltr none ;
```

```
ANY_DECLARATOR := absdcltr DECLARATOR ;
absdcltr -> implemented as void ;

parenth_dcltr -> DECLARATOR ;
```

```
  func_dcltr -> ANY_DECLARATOR FUNC_PARAMETERS TYPE_QUALIFIERS
EXCEPTION_SPECIFICATION ;
  array_dcltr -> ANY_DECLARATOR EXPR_OPT ;
  ptr_dcltr -> TYPE_QUALIFIERS
  ANY_DECLARATOR ; ref_dcltr ->
  TYPE_QUALIFIERS ANY_DECLARATOR ;
  memptr_dcltr -> NESTED_NAME_SPECIFIER TYPE_QUALIFIERS
  ANY_DECLARATOR ;


  STRUCT_DECLS_OPT := none struct_decls ;
  STRUCT_DECL := struct_decl macro_call
  PREPROCESSOR ; struct_decls -> STRUCT_DECL *;
  struct_decl -> TYPE_SPECIFIERS MEMBERS ;
  MEMBERS := members ;
  members -> MEMBER * ;
  MEMBER := DECLARATOR bit_field ;
  bit_field -> DECLARATOR_OPT
  SIMPLE_EXPR ; DECLARATOR_OPT := none
  DECLARATOR ;

  FUNC_PARAMETERS := PARMLIST identifiers
  ; PARMLIST := parmlist var_parmlist ;
  parmlist -> PARM_DECL * ;
  var_parmlist -> FIXED_PARMLIST
  ; FIXED_PARMLIST := parmlist ;
  PARM_DECL := parm_decl ;
  parm_decl -> DECLARATION_SPECIFIERS PARAM_DCLTR SIMPLE_EXPR_OPT ;
  PARAM_DCLTR := ANY_DECLARATOR ;
  identifiers -> IDENTIFIER *

; end chapter;

chapter FUNCTIONS
  abstract syntax

  fndef -> DECLARATION_SPECIFIERS DECLARATOR KR_DECLS FNBODY ;
  var_kr_arglist -> FIXED_KR_DECLS ;
  kr_arglist -> KR_DECL + ;
  subfndef -> DECLARATION_SPECIFIERS DECLARATOR COMPOUND ;

  KR_DECLS := none var_kr_arglist kr_arglist ;
  KR_DECL := decl macro_call ;
  FIXED_KR_DECLS := kr_arglist ;
  FNBODY := COMPOUND ctor_initializer try_block ;
  COMPOUND := compound block ;

end chapter;

chapter TYPES
  abstract syntax

  DECLARATION_SPECIFIERS := declaration_specifiers macro_call ;
  declaration_specifiers -> DECLARATION_SPECIFIER * ;
  DECLARATION_SPECIFIER := TYPE_SPECIFIER storage_class ;
```

```
TYPE_SPECIFIER := typespec STRUCT_SPECIFIER QUALIFIED_ID
            typeof macro_call type_qualifier ;
QUALIFIED_TYPE_OPT := QUALIFIED_ID none ;
TYPE_NAME := IDENTIFIER template_name ;
STRUCT_SPECIFIER := struct union enum class typename_id ;
NESTED_NAME_SPECIFIER := scope global_scope ;
```

```
  scope -> TYPE_NAME + ;
  global_scope -> TYPE_NAME * ;
  typeof -> TYPEOF_ARG;
  TYPEOF_ARG := SIMPLE_EXPR TYPENAME ;

  TYPE_SPECIFIERS := type_specifiers ;
  type_specifiers -> TYPE_SPECIFIER * ;

  typespec -> implemented as name ;
  storage_class -> implemented as name
  ; type_qualifier -> implemented as
  name ;

  struct -> IDENTIFIER_OPT STRUCT_DECLS_OPT
  ; union -> IDENTIFIER_OPT STRUCT_DECLS_OPT
  ; enum -> QUALIFIED_TYPE_OPT ENUMLIST_OPT
  ; class -> CLASS_HEAD CLASS_DECLS_OPT ;
  typename_id -> QUALIFIED_ID ;
  IDENTIFIER_OPT := none IDENTIFIER ;

  enumlist -> ENUMERATOR *;
  enumerator -> IDENTIFIER SIMPLE_EXPR_OPT ;
  SIMPLE_EXPR_OPT := none SIMPLE_EXPR ;
  ENUMLIST_OPT := none enumlist ;
  ENUMERATOR := enumerator ;

  type_id -> TYPE_SPECIFIERS ABSDCLTR ;
  TYPENAME := type_id ;
  TYPE_QUALIFIERS := none type_qualifiers ;
  type_qualifiers -> TYPE_QUALIFIER + ;
  TYPE_QUALIFIER := type_qualifier ;
  parenth_type -> TYPENAME ;
  new_gnu_type -> TYPENAME EXPR
  ; type_list -> TYPENAME *;

end chapter;

chapter CLASS
  abstract syntax

  CLASS_HEAD := class_head ;
  class_head -> AGGR QUALIFIED_TYPE_OPT BASE_CLASS_OPT ;
  AGGR := class_kw struct_kw union_kw ;
  class_kw -> implemented as void ;
  struct_kw -> implemented as void ;
  union_kw -> implemented as void ;
  BASE_CLASS_OPT := none base_classes
  ; base_classes -> BASE_CLASS + ;
  BASE_CLASS := base_class ;
  base_class -> ACCESS_LIST QUALIFIED_ID ;
  ACCESS_LIST := access_list none ;
  access_list -> BASE_SPECIFIER + ;
  BASE_SPECIFIER := ACCESS_SPECIFIER storage_class ;
  ACCESS_SPECIFIER_OPT := ACCESS_SPECIFIER none ;
  ACCESS_SPECIFIER := access_specifier ;
```

```
access_specifier -> implemented as name ;
CLASS_DECLS_OPT := CLASS_DECLS none ;
CLASS_DECLS := class_decls ;
class_decls -> CLASS_DECL_SECTION
* ;
```

```
  CLASS_DECL_SECTION := class_decls_section ;
  class_decls_section -> ACCESS_SPECIFIER_OPT
  COMPONENT_DECLS ; COMPONENT_DECLS := component_decl_list
  ;
  component_decl_list -> COMPONENT_DECL * ;
-- MG - 23 novembre 1999 - ajout de template_def
  COMPONENT_DECL := member_decl fndef using_declaration
using_directive none macro_call template_def
explicit_instanciation explicit_specialization ;
  member_decl -> DECLARATION_SPECIFIERS INITDECLS_OPT ;


end chapter;

chapter

EXPRESSIONS
  abstract syntax

  EXPRLIST := exprlist ;
  exprlist -> SIMPLE_EXPR * ;

  EXPR := SIMPLE_EXPR expr ;
  expr -> SIMPLE_EXPR + ;

  pmap -> SIMPLE_EXPR SIMPLE_EXPR
  ; pmpp -> SIMPLE_EXPR
  SIMPLE_EXPR ; plus ->
  SIMPLE_EXPR SIMPLE_EXPR ; minus
  -> SIMPLE_EXPR SIMPLE_EXPR ;
  mul -> SIMPLE_EXPR SIMPLE_EXPR
  ; div -> SIMPLE_EXPR
  SIMPLE_EXPR ; rem ->
  SIMPLE_EXPR SIMPLE_EXPR ; lsh -
  > SIMPLE_EXPR SIMPLE_EXPR ; rsh
  -> SIMPLE_EXPR SIMPLE_EXPR ; lt
  -> SIMPLE_EXPR SIMPLE_EXPR ;
  gt -> SIMPLE_EXPR SIMPLE_EXPR ;
  ge -> SIMPLE_EXPR SIMPLE_EXPR ;
  le -> SIMPLE_EXPR SIMPLE_EXPR ;
  eq -> SIMPLE_EXPR SIMPLE_EXPR ;
  neq -> SIMPLE_EXPR SIMPLE_EXPR
  ; bwand -> SIMPLE_EXPR
  SIMPLE_EXPR ; bwor ->
  SIMPLE_EXPR SIMPLE_EXPR ; bwxor
  -> SIMPLE_EXPR SIMPLE_EXPR ;
  and -> SIMPLE_EXPR SIMPLE_EXPR
  ; or -> SIMPLE_EXPR SIMPLE_EXPR
  ;
  cond -> SIMPLE_EXPR EXPR SIMPLE_EXPR
  ; ass -> SIMPLE_EXPR SIMPLE_EXPR ;
  plus_ass -> SIMPLE_EXPR SIMPLE_EXPR ;
  minus_ass -> SIMPLE_EXPR SIMPLE_EXPR
  ; mul_ass -> SIMPLE_EXPR SIMPLE_EXPR
  ; div_ass -> SIMPLE_EXPR SIMPLE_EXPR
```

```
  ; rem_ass -> SIMPLE_EXPR SIMPLE_EXPR
  ;
  bwand_ass -> SIMPLE_EXPR SIMPLE_EXPR
  ; bwxor_ass -> SIMPLE_EXPR
  SIMPLE_EXPR ; bwor_ass -> SIMPLE_EXPR
  SIMPLE_EXPR ; lsh_ass -> SIMPLE_EXPR
  SIMPLE_EXPR ; rsh_ass -> SIMPLE_EXPR
  SIMPLE_EXPR ;
  throw -> SIMPLE_EXPR_OPT ;
  SIMPLE_EXPR := CAST_EXPR pmap pmpp throw
                plus minus mul div rem lsh rsh lt gt ge le eq neq
                bwand
bwor
```

```
                bwxor and or cond ass plus_ass minus_ass mul_ass
                div_ass rem_ass bwand_ass bwxor_ass bwor_ass
                lsh_ass rsh_ass ;

  CAST_EXPR := UNARY_EXPR cast ;
  cast -> TYPENAME CAST_EXPR ;

  UNARY_EXPR := PRIMARY
               deref addr uminus uplus pre_incr pre_decr bwnot not
sizeo
f          new global_scope_new
           delete global_scope_delete array_delete
global_scope_array_delet
  e ; deref -> CAST_EXPR
  ;  addr -> CAST_EXPR ;
  uminus -> CAST_EXPR ;
  uplus -> CAST_EXPR ;
  pre_incr -> CAST_EXPR ;
  pre_decr -> CAST_EXPR ;
  bwnot -> CAST_EXPR ;
  not -> CAST_EXPR ;
  sizeof -> SIZEOF_ARG ;
  SIZEOF_ARG := UNARY_EXPR TYPENAME
  ; new -> EXPRLIST NEW_TYPE_ID
  EXPRLIST ;
  global_scope_new -> EXPRLIST NEW_TYPE_ID EXPRLIST
  ;
  NEW_TYPE_ID := parenth_type type_id new_gnu_type
  ; delete -> SIMPLE_EXPR ;
  global_scope_delete -> SIMPLE_EXPR ;
  array_delete -> SIMPLE_EXPR SIMPLE_EXPR ;
  global_scope_array_delete -> SIMPLE_EXPR
  SIMPLE_EXPR ;

  PRIMARY := QUALIFIED_ID integer float character string
parenth_expr call
            true false this strings
            functional_cast dynamic_cast static_cast
reinterpret_cast const_cast typeid
            index dot arrow post_incr post_decr macro_call
            PREPROCESSOR
;
  integer -> implemented as string ;
  float -> implemented as string ;
  character -> implemented as string
  ; STRING1 := string ;
  strings -> STRING1 + ;
  string -> implemented as string
  ; true -> implemented as void ;
  false -> implemented as void ;
  this -> implemented as void ;
  parenth_expr -> EXPR ;
  call -> PRIMARY EXPRLIST ;
  index -> PRIMARY EXPR ;
  dot -> PRIMARY IDENTIFIER ;
```

```
arrow -> PRIMARY IDENTIFIER
;
post_incr -> PRIMARY ;
post_decr -> PRIMARY ;
functional_cast -> TYPE_SPECIFIER EXPRLIST ;
dynamic_cast -> TYPENAME SIMPLE_EXPR ;
static_cast -> TYPENAME SIMPLE_EXPR ;
reinterpret_cast -> TYPENAME SIMPLE_EXPR ;
const_cast -> TYPENAME SIMPLE_EXPR ;
```

```
  typeid -> SIMPLE_EXPR ;

end chapter ;

chapter STATEMENTS
  abstract syntax

  STMTS := stmts ;
  stmts -> STMT * ;

-- MG - 24 novembre 1999 - ajouts de asm,
namespace_def, using_declaration
  STMT := COMPOUND expr_stmt if while do for loop switch break
continue return goto
         empty_stmt case default label decl subfndef macro_call
PREPROCESSOR SQL_STMT
            asm namespace_def using_declaration using_directive;
  compound -> STMTS ;
  block -> OPEN_BLOCK STMTS CLOSE_BLOCK ;
  loop -> OPEN_BLOCK STMTS CLOSE_BLOCK ;
  OPEN_BLOCK := macro_call none ;
  CLOSE_BLOCK := macro_call none ;
  expr_stmt -> EXPR ;
  if -> EXPR STMTS STMTS_OPT ;
  STMTS_OPT := STMTS none ;
  while -> EXPR STMTS ;
  do -> STMTS EXPR ;
  for -> EXPR_OPT EXPR_OPT EXPR_OPT STMTS ;
  switch -> EXPR STMTS ;
  break -> implemented as void ;
  continue -> implemented as void ;
  return -> EXPR_OPT ;
  goto -> IDENTIFIER ;
  empty_stmt -> implemented as void
  ; case -> SIMPLE_EXPR ;
  default -> implemented as void ;
  label -> IDENTIFIER ;

end chapter ;

chapter MEMBER_FUNCTION
  abstract syntax

  destructor -> IDENTIFIER ;
  operator -> OPERATOR ;
  OPERATOR := OPER type_id ;
  OPER := op_mult op_div op_mod op_plus op_minus op_bwor op_bwand
op_bwxor op_bwnot
     op_comma op_eq op_less op_greater op_leq op_geq op_neq
     op_ass_plus op_ass_minus op_ass_mult op_ass_div op_ass_mod
     op_ass_bwxor op_ass_bwand op_ass_bwor op_ass_lshift
     op_ass_rshift
op_ass
     op_lshift op_rshift op_incr op_decr op_and op_or op_not
     op_cond op_arrow op_pmap op_parenth op_croch
```

```
   op_new op_new_array op_delete op_delete_array ;

op_mult -> implemented as void ;
op_div -> implemented as void ;
```

```
op_mod -> implemented as void ;
op_plus -> implemented as void ;
op_minus -> implemented as void ;
op_bwand -> implemented as void ;
op_bwor -> implemented as void ;
op_bwxor -> implemented as void ;
op_bwnot -> implemented as void ;
op_comma -> implemented as void ;
op_eq -> implemented as void ;
op_less -> implemented as void ;
op_greater -> implemented as void
; op_leq -> implemented as void ;
op_geq -> implemented as void ;
op_neq -> implemented as void ;
op_ass_plus -> implemented as void
; op_ass_minus -> implemented as
void ; op_ass_mult -> implemented
as void ; op_ass_div ->
implemented as void ;

op_ass_mod -> implemented as void
; op_ass_bwxor -> implemented as
void ; op_ass_bwand -> implemented
as void ; op_ass_bwor ->
implemented as void ;
op_ass_lshift -> implemented as
void ; op_ass_rshift ->
implemented as void ; op_ass ->
implemented as void ; op_lshift ->
implemented as void ; op_rshift ->
implemented as void ; op_incr ->
implemented as void ; op_decr ->
implemented as void ; op_and ->
implemented as void ;
op_or -> implemented as void ;
op_not -> implemented as void ;
op_cond -> implemented as void ;
op_arrow -> implemented as void ;
op_pmap -> implemented as void ;
op_parenth -> implemented as void
; op_croch -> implemented as void
; op_new -> implemented as void ;
op_new_array -> implemented as
void ; op_delete -> implemented as
void ;
op_delete_array -> implemented as void ;

ctor_initializer -> MEMBER_INIT_LIST
COMPOUND ; MEMBER_INIT_LIST :=
member_init_list ; member_init_list ->
MEMBER_INIT * ;  MEMBER_INIT := member_init
;
member_init -> MEMBER_NAME EXPRLIST ;
MEMBER_NAME := QUALIFIED_ID none;
```

```
end chapter ;

chapter TEMPLATE
  abstract syntax

-- MG - 24/11/99 - 3ème argument : remplacement de TEMPLATE_DEF
par EXTDEF
  template_def -> EXPORT_OPT TEMPLATE_HEADER EXTDEF ;
```

```
  EXPORT_OPT := export none ;
  export -> implemented as void
  ;
  TEMPLATE_HEADER := template_parms ;
  template_parms -> TEMPLATE_PARM + ;
  TEMPLATE_PARM := type_parameter template_parameter parm_decl ;
  type_parameter -> TYPE_PARM_DEF TYPE_PARM_INIT ;
  TYPE_PARM_DEF := class_head typename_id ;
  template_parameter -> TEMPLATE_HEADER TEMPLATE_PARM_DEF
  TYPE_PARM_INIT
;
  TEMPLATE_PARM_DEF := class_head ;
  template_name -> IDENTIFIER TEMPLATE_ARG_LIST ;
  TEMPLATE_ARG_LIST := template_arg_list ;
  template_arg_list -> TEMPLATE_ARG * ;
  TEMPLATE_ARG := TYPE_NAME SIMPLE_EXPR ;
  TYPE_PARM_INIT := none type_id
  TYPE_SPECIFIER ; template_id -> QUALIFIED_ID
  ; explicit_instanciation -> EXTDEF ;
  explicit_specialization -> EXTDEF ;

end chapter ;

chapter NAMESPACE
  abstract syntax

  namespace_def -> IDENTIFIER_OPT NAMESPACE_DEF ;
  NAMESPACE_DEF := EXTDEFS QUALIFIED_ID ;
  using_declaration -> QUALIFIED_ID
  ; using_directive -> QUALIFIED_ID
  ;

end chapter ;

chapter EXCEPTION
  abstract syntax

  EXCEPTION_SPECIFICATION := none exception_spec ;
  exception_spec -> RAISE_IDENTIFIERS ;
  RAISE_IDENTIFIERS := type_list ;
  try_block -> TRY_STMTS HANDLER_SEQ ;
  TRY_STMTS := COMPOUND
  ctor_initializer ; HANDLER_SEQ :=
  try_handlers ; try_handlers ->
  TRY_HANDLER * ; TRY_HANDLER :=
  handler ;
  handler -> PARMLIST COMPOUND ;

end chapter ;

chapter

PREPROCESSOR
  abstract syntax
```

```
  PREPROCESSOR := pp_none pp_define pp_undef pp_conditional
pp_line pp_error pp_pragma
          pp_include ;
  pp_define -> PP_MACRO_NAME PP_MACRO_ARGS PP_TEXT
  ; pp_undef -> PP_MACRO_NAME PP_TEXT ;
  PP_MACRO_NAME := identifier ;
  PP_MACRO_ARGS := pp_macro_args
  none; pp_macro_args -> IDENTIFIER
  *;
  pp_conditional -> PP_IF EVERY PP_ELIF_PARTS PP_ELSE_PART PP_ENDIF
  ;
```

```
  PP_IF := pp_if pp_ifdef pp_ifndef
  ; PP_ELIF_PARTS := pp_elif_parts
  none ; pp_elif_parts ->
  PP_ELIF_PART + ; PP_ELIF_PART :=
  pp_elif_part ; pp_elif_part ->
  PP_ELIF EVERY ; PP_ELIF := pp_elif
  ;
  PP_ELSE_PART := pp_else_part none
  ; pp_else_part -> PP_ELSE EVERY ;
  PP_ELSE := pp_else ;
  PP_ENDIF := pp_endif ;

  pp_if -> PP_TEXT ;
  pp_ifdef -> PP_IFDEF_IDENT PP_TEXT
  ; pp_ifndef -> PP_IFDEF_IDENT
  PP_TEXT ; PP_IFDEF_IDENT :=
  identifier ; pp_elif -> PP_TEXT ;
  pp_else -> PP_TEXT ;
  pp_endif -> PP_TEXT ;
  pp_line -> PP_TEXT ;
  pp_error -> PP_TEXT ;
  pp_pragma -> PP_TEXT
  ; pp_none -> PP_TEXT
  ;
  pp_include -> PP_FILENAME PP_TEXT ;
  PP_FILENAME := none pp_external_file pp_local_file
  ; pp_external_file -> implemented as string ;
  pp_local_file -> implemented as string ;
  PP_TEXT := pp_text ;
  pp_text -> PP_TEXT_LINE
  * ;
  PP_TEXT_LINE := pp_text_line ;
  pp_text_line -> implemented as string ;

  macro_call -> implemented as string;

end chapter;

chapter SQL
  abstract syntax

  SQL_STMT := sql_stmt ;
  sql_stmt -> SQL_TYPE SQL_LINES
  ; SQL_LINES := sql_lines ;
  sql_lines -> SQL_LINE * ;
  SQL_LINE := sql_line ;
  SQL_TYPE := identifier ;
  sql_line -> implemented as string

; end chapter;

chapter MISC
  abstract syntax
```

```
identifier -> implemented as
name ; none -> implemented as
void; EXPR_OPT := EXPR none ;
IDENTIFIER := identifier;
QUALIFIED_ID := qualified_id    UNQUALIFIED_ID ;
qualified_id -> NESTED_NAME_SPECIFIER
UNQUALIFIED_ID ;
```

```
  UNQUALIFIED_ID := IDENTIFIER destructor operator
  template_name ;


end chapter ;

frames
   prefix -> implemented as tree;
       controls copy save;
   postfix -> implemented as tree;
       controls copy save;
   focus -> implemented as integer;
       controls copy;

end chapter;

end definition
```

Concrete syntax

```
rules definition of C version 1
              is

-- %start program

-- All identifiers that are not reserved words
-- %token %IDENT

-- Reserved words that specify storage class
-- ie : auto register static extern typedef inline virtual
-- %token %STORAGECLASS

-- Reserved words that specify type.
-- ie: void char short int long float double signed unsigned
-- %token %TYPESPEC

-- Reserved words that qualify type
-- ie: const volatile
-- %token %TYPEQUAL

-- Character or numeric constants.
-- %token %INTEGER %FLOAT %CHARACTER

-- String constants
-- %token %STRING

-- "...", used for functions with variable arglists.
-- %token %ELLIPSIS

-- the reserved words
-- %token SIZEOF ENUM STRUCT UNION IF ELSE WHILE DO FOR
SWITCH CASE DEFAULT
-- %token BREAK CONTINUE RETURN GOTO
-- C++
-- CLASS DELETE NEW FRIEND OPERATOR PRIVATE PROTECTED PUBLIC
-- THROW TRY CATCH NAMESPACE USING TEMPLATE EXPORT
```

```
-- TYPEID DYNAMIC_CAST STATIC_CAST REINTERPRET_CAST CONST_CAST

-- Used to resolve s/r with epsilon
```

```
%[LEFT %EMPTY ]%
-- Add precedence rules to solve dangling else s/r conflict
%[NONASSOC 'if' ]%
%[NONASSOC 'else' ]%
%[LEFT %IDENT, %TYPENAME, %STORAGECLASS, %TYPESPEC, %TYPEQUAL,
'enum',
'class', 'struct', 'union' , "...", 'typeof', 'operator', 'typename'
]%
%[LEFT "{", ",", ";" ]%
%[NONASSOC 'throw' ]%

-- Define the operator tokens and their precedences.
%[RIGHT "+=", "-=", "*=", "/=", "%=", "&=", "^=", "|=", "<<=",
">>=", "="
]%
%[RIGHT "?" , ":" ]%
%[LEFT "||" ]%
%[LEFT "&&" ]%
%[LEFT "|" ]%
%[LEFT "^" ]%
%[LEFT "&" ]%
%[LEFT "==" , "!="]%
%[LEFT "<" , ">" , "<=" , ">=" ]%
%[LEFT "<<" , ">>" ]%
%[LEFT "+" , "-" ]%
%[LEFT "*" , "/" , "%" ]%
-- C++
%[LEFT "->*", ".*" ]%
%[RIGHT %UNARY , "++" , "--" ]%
%[LEFT %HYPERUNARY ]%
%[LEFT "->" , "." , "(" , "[" ]%
-- C++
%[NONASSOC %SCOPE]%
%[RIGHT "::"]%
%[NONASSOC 'new', 'delete', 'try', 'catch' ]%

chapter PARSER

chapter TOP

rules

entry_point := program ;
     program
entry_point := '[EXTDEF]' extdef ;
     extdef
entry_point := '[STMTS]' stmts
     ; stmts
entry_point := '[STMT]' stmt_or_decl_or_label ;
     stmt_or_decl_or_label
entry_point := '[EXPR]'
expr ;
     expr
entry_point := '[CLASS_DECLS]' component_decl_list ;
     component_decl_list
```

```
entry_point := '[COMPONENT_DECL]' component_decl ;
     component_decl

-- ANSI C forbids an empty source file
program := prog_extdefs ;
     :program<prog_extdefs>
```

```
recover prog_extdefs as
extdef; prog_extdefs := ;
     :extdefs<>
prog_extdefs := prog_extdefs extdef ;
     prog_extdefs:<..,extdef>


recover extdef;
extdef := extdef0
; extdef0
extdef0 := %[CBLOCK_BEGIN]% extdef1 %[CBLOCK_END]% ;
     extdef1

extdef1 := fndef ;
     fndef
extdef1 := datadef %{C_Add_Typedef_Name();}% ;
     datadef
-- C++
extdef1 := template_def ;
     template_def
extdef1 :=
     explicit_instanciation ;
     explicit_instanciation
extdef1 := explicit_specialization ;
     explicit_specialization
extdef1 := namespace_def ;
     namespace_def
extdef1 := using_decl
     ; using_decl
extdef1 := 'asm' "(" string ")" ";" ;
     :asm<string>
extdef1 := extern_def;
     extern_def
-- ANSI C++ does not allow extra `;' outside of a function
-- empty
declaration
extdef1   := ";" ;
     :empty_extdecl
extdef1 := sql_stmt ;
       sql_stmt
extdef1 := extdefs
     preprocessor ; extdefs
     preprocessor

-- ANSI C forbids data definition with no type or storage class
-- data definition has no type or storage class
recover datadef ";" ;
datadef   := initdecls ";" ;
     :extdecl<:declaration_specifiers<>,initdecls>
datadef := declmods initdecls ";" ;
     :extdecl<declmods,initdecls>
datadef   := typed_declspecs initdecls ";" ;
     :extdecl<typed_declspecs,initdecls>
datadef := declmods ";" ;
```

```
     :extdecl<declmods,:none>
datadef   := typed_declspecs ";"
;
     :extdecl<typed_declspecs,:none>
datadef := declmacro_call ";" ;    %[PREC %EMPTY]%
     :extdecl<declmacro_call, :none>
datadef := declmacro_call ;        %[PREC
     %IDENT]% declmacro_call
```

```
recover fndef fnbody;
fndef := typed_declspecs declarator fnbody ;
     :fndef<typed_declspecs,declarator, :none,fnbody>
fndef := declmods declarator fnbody ;
     :fndef<declmods,declarator,
:none,fnbody> fndef := declarator fnbody ;
     :fndef<:declaration_specifiers<>,declarator,
:none,fnbody> fndef := declmacro_call fnbody ;
     :fndef<declmacro_call, :none, :none,
fnbody> fndef := typed_declspecs declarator
ctor_init_body ;
     :fndef<typed_declspecs,declarator, :none,
ctor_init_body> fndef := declmods declarator
ctor_init_body ;
     :fndef<declmods,declarator, :none, ctor_init_body>
fndef := declarator ctor_init_body ;
     :fndef<:declaration_specifiers<>,declarator, :none,
ctor_init_body> fndef := declmacro_call ctor_init_body ;
     :fndef<declmacro_call, :none, :none,
ctor_init_body> fndef := typed_declspecs declarator
fntry_block ;
     :fndef<typed_declspecs,declarator, :none,
fntry_block> fndef := declmods declarator fntry_block
;
     :fndef<declmods,declarator, :none, fntry_block>
fndef := declarator fntry_block ;
     :fndef<:declaration_specifiers<>,declarator, :none,
     fntry_block>
fndef := declmacro_call fntry_block ;
     :fndef<declmacro_call, :none, :none, fntry_block>

fnbody := open_blk %[CBLOCK_END]% %[CBLOCK_BEGIN]% fnbody1
close_blk ;
       %{
         VTP_TreeP close = Parser_Pop();
         VTP_TreeP stmts = Parser_Pop();
       VTP_TreeP open = Parser_Pop();
       VTP_TreeP compound ;
       if ((VTP_TREE_OPERATOR(open) == C_1_op_none)
           && (VTP_TREE_OPERATOR(close) == C_1_op_none)) {
         compound = VTP_TreeMake(C_1_op_compound);
         VTP_TreeSetChild(compound,stmts,0);
         Parser_SetCoordNN(compound,open,close);
         VTP_TreeDestroy(open);
         VTP_TreeDestroy(close);
       } else {
         compound = VTP_TreeMake(C_1_op_block);
         VTP_TreeSetChild(compound,open,0);
         VTP_TreeSetChild(compound,stmts,1);
         VTP_TreeSetChild(compound,close,2);
         Parser_SetCoordNN(compound,open,clos
         e);
       }
```

```
        Parser_Push(compound);
      }%

recover fnbody1;
fnbody1 := stmts
;
      stmts

extern_def := 'extern' string extdef ;
      :extern_def<string, extdef>
extern_def := 'extern' string "{" comp_extdefs "}" ;
      :extern_def<string, comp_extdefs>
```

```
-- ANSI C forbids an empty source
file recover comp_extdefs as
extdef; comp_extdefs := ;
     :extdefs<>
comp_extdefs := comp_extdefs extdef ;
     comp_extdefs:<..,extdef>

end chapter ;

chapter

IDENTIFIER rules

puretype_template_name := typename %{/*[*/
C_Type_Name_Push();/*]*/}% "<"
%{/*[*/ C_Template_Args_List_Push();/*]*/}% template_arg_list ">"
%{/*[*/ C_Template_Args_List_Pop();/*]*/}% ; %[PREC %UNARY]%
     :template_name<typename, template_arg_list>
notype_template_name := notype_identifier %{/*[*/
C_Type_Name_Push();/*]*/}% "<" %{/*[*/
C_Template_Args_List_Push();/*]*/}% template_arg_list ">"
%{/*[*/ C_Template_Args_List_Pop();/*]*/}% ; %[PREC %UNARY]%
     :template_name<notype_identifier, template_arg_list>

-- reduce first an ident in identifier and typename in type_name
notype_identifier := %IDENT ;
     :identifier[%IDENT]
typename := %TYPENAME ;
     :identifier[%TYPENAME]
identifier := notype_identifier
;
     notype_identifier
notype_unqualified_id := "~"
identifier ;
     :destructor<identifier>
notype_unqualified_id := operator_name
;
     operator_name
notype_unqualified_id := notype_identifier ;
     notype_identifier
notype_unqualified_id := notype_template_name ;
     notype_template_name
type_name := typename %{/*[*/ C_Type_Name_Push(); /*]*/}%;
     typename
type_name := puretype_template_name ;
     puretype_template_name
identifier := typename ;
     typename
type_name := notype_identifier %{/*[*/ C_Type_Name_Push(); /*]*/}%;
     notype_identifier
type_name := notype_template_name ;
     notype_template_name
unqualified_id :=
```

```
     notype_unqualified_id ;
     notype_unqualified_id
unqualified_id := puretype_template_name ;
     puretype_template_name
unqualified_id := typename
     ; typename

qualified_id := unqualified_id ;
     unqualified_id
```

```
qualified_id := nested_name_specifier unqualified_id
%{C_Reset_Current_Scope();}% ;
     :qualified_id<nested_name_specifier, unqualified_id>
qualified_id := nested_name_specifier
template_unqualified_id
%{C_Reset_Current_Scope();}% ;
     :qualified_id<nested_name_specifier, template_unqualified_id>
template_unqualified_id := 'template' unqualified_id ;
     :template_id<unqualified_id>

--notype_qualified_id := nested_name_specifier notype_unqualified_id
%{C_Reset_Current_Scope();}% ;
--   :qualified_id<nested_name_specifier, notype_unqualified_id>
--notype_qualified_id := nested_name_specifier
template_notype_unqualified_id
%{C_Reset_Current_Scope();}% ;
--
     :qualified_id<nested_name_specifi
er, template_notype_unqualified_id>
--overqualified_id := notype_unqualified_id ;
--   notype_unqualified_id
--overqualified_id := notype_qualified_id ;
--   notype_qualified_id
overqualified_id :=
qualified_id ;
     qualified_id
overqualified_id := global_scope_name_specifier qualified_id
%{C_Reset_Current_Scope();}% ;
     :qualified_id<global_scope_name_specifier, qualified_id>
overqualified_id := global_scope_name_specifier
template_unqualified_id
%{C_Reset_Current_Scope();}% ;
     :qualified_id<global_scope_name_specifier,
     template_unqualified_id>
--overqualified_id := global_scope_name_specifier
notype_unqualified_id
%{C_Reset_Current_Scope();}% ;
--   :qualified_id<global_scope_name_specifier,
notype_unqualified_id>
--overqualified_id := global_scope_name_specifier
template_notype_unqualified_id
%{C_Reset_Current_Scope();}% ;
--
     :qualified_id<global_scope_name_specif
ier, template_notype_unqualified_id>
--template_notype_unqualified_id := 'template' notype_unqualified_id
;
--   :template_id<notype_unqualified_id>
template_overqualified_id := 'template'
overqualified_id ;
     :template_id<overqualified_id>
any_id := qualified_id ;
     qualified_id
any_id := global_scope_name_specifier unqualified_id
```

```
%{C_Reset_Current_Scope();}% ;
     :qualified_id<global_scope_name_specifier, unqualified_id>
any_id := global_scope_name_specifier template_unqualified_id
%{C_Reset_Current_Scope();}% ;
     :qualified_id<global_scope_name_specifier,
     template_unqualified_id>

global_scope_name_specifier := "::" ;
     :global_scope<>
global_scope_name_specifier := global_scope_name_specifier type_name
"::"
%{/*[*/C_Current_Scope_Push();/*]*/}% ; %[PREC
     %SCOPE]% global_scope_name_specifier:<..,
     type_name>
nested_name_specifier := type_name "::"
%{/*[*/C_Current_Scope_Push();/*]*/}% ;          %[PREC %SCOPE]%
     :scope<type_name>
nested_name_specifier := nested_name_specifier type_name "::"
%{/*[*/C_Current_Scope_Push();/*]*/}% ; %[PREC %SCOPE]%
```

```
     nested_name_specifier:<.., type_name>

-- any identifier (type or not) but not destructor or operator
qualified_ident_or_type := qualified_type_name ;
     qualified_type_name
qualified_type_name := type_name ;                     %[PREC
%IDENT]%
     type_name
qualified_type_name := nested_type ;
     nested_type
-- l'ajout de reset_current_scope introduit 1 S/R
nested_type := nested_name_specifier type_name
%{C_Reset_Current_Scope();}%;           %[PREC %IDENT]%
     :qualified_id<nested_name_specifier, type_name>
overqualified_type_name := qualified_type_name ;
     qualified_type_name
-- l'ajout de reset_current_scope introduit 1 S/R
overqualified_type_name := global_scope_name_specifier type_name
%{C_Reset_Current_Scope();}%; %[PREC %IDENT]%
     :qualified_id<global_scope_name_specifier, type_name>
simple_type_specifier := typespec ;
     typespec
simple_type_specifier :=
typename ;
     typename
simple_type_specifier := puretype_template_name ;
       puretype_template_name
simple_type_specifier := nested_name_specifier typename
%{C_Reset_Current_Scope();}% ;
     :qualified_id<nested_name_specifier, typename>
simple_type_specifier := nested_name_specifier
puretype_template_name
%{C_Reset_Current_Scope();}% ;
       :qualified_id<nested_name_specifier,
puretype_template_name> simple_type_specifier :=
global_scope_name_specifier typename
%{C_Reset_Current_Scope();}% ;
     :qualified_id<global_scope_name_specifier,
typename> simple_type_specifier :=
global_scope_name_specifier puretype_template_name
%{C_Reset_Current_Scope();}% ;
       :qualified_id<global_scope_name_specif
ier, puretype_template_name>

string := string1 ;
     string1
string := string_list string1 ;
     string_list:<..,string1>
string_list := string1 ;
     :strings<string1>
string_list := string_list string1 ;
     string_list:<..,string1>
string1 := %STRING ;
     :string[%STRING]
string1 :=
```

```
stringmacro_call ;
     stringmacro_call
end chapter ;

chapter EXPRESSION

rules
```

```
recover expr;
expr := %[CBLOCK_BEGIN]% expr1 %[CBLOCK_END]% ;
     expr1
recover
expr_stmt;
expr_stmt := expr1 ;
     expr1
expr1 := nonnull_exprlist ;
     %{
       VTP_TreeP tree =
       Parser_Pop(); VTP_TreeP
       expr ;
       if (VTP_TreeLength(tree) == 1) {
         expr =
         VTP_TreeDisown(tree,0);
         VTP_TreeDestroy(tree);
       } else {
         expr = tree ;
         VTP_TREE_OPERATOR(expr) = C_1_op_expr ;
          }
       Parser_Push(expr);
     }%

recover exprlist ;
exprlist := ;
     :exprlist<>
exprlist := %[CBLOCK_BEGIN]% nonnull_exprlist %[CBLOCK_END]% ;
     nonnull_exprlist

nonnull_exprlist := expr_no_commas ;
     :exprlist<expr_no_commas>
nonnull_exprlist := nonnull_exprlist "," expr_no_commas ;
     nonnull_exprlist:<..,expr_no_commas>

recover simple_expr;
simple_expr :=
expr_no_commas ;
     expr_no_commas

unary_expr := primary ;
     primary
unary_expr := "*" cast_expr ;                %[PREC %UNARY ]%
     :deref<cast_expr>
unary_expr := "&" cast_expr ;                %[PREC %UNARY ]%
     :addr<cast_expr>
unary_expr := "-" cast_expr ;                %[PREC %UNARY ]%
     :uminus<cast_expr>
unary_expr := "+" cast_expr ;                %[PREC %UNARY ]%
     :uplus<cast_expr>
unary_expr := "++" cast_expr ;               %[PREC %UNARY ]%
     :pre_incr<cast_expr>
unary_expr := "--" cast_expr ;               %[PREC %UNARY ]%
     :pre_decr<cast_expr>
unary_expr := "~" cast_expr ;                %[PREC %UNARY ]%
     :bwnot<cast_expr>
```

```
unary_expr := "!" cast_expr ;                %[PREC %UNARY ]%
     :not<cast_expr>
unary_expr := 'sizeof' unary_expr ;          %[PREC %UNARY ]%
     :sizeof<unary_expr>
unary_expr := 'sizeof' "(" type_id ")" ;     %[PREC
%HYPERUNARY]%
     :sizeof<type_id>
unary_expr := "::" 'new' new_type_id ;       %[PREC %EMPTY]%
```

```
     :global_scope_new<:none, new_type_id, :none>
unary_expr := "::" 'new' new_type_id new_initializer ;
     :global_scope_new<:none, new_type_id, new_initializer>
unary_expr := "::" 'new' new_placement new_type_id ;   %[PREC
%EMPTY]%
     :global_scope_new<new_placement, new_type_id, :none>
unary_expr := "::" 'new' new_placement new_type_id
new_initializer ;
     :global_scope_new<new_placement, new_type_id,
new_initializer> unary_expr := "::" 'delete' cast_expr ;
                                                    %[PREC
%UNARY]%
     :global_scope_delete<cast_expr>
unary_expr := "::" 'delete' "[" "]" cast_expr ;    %[PREC
%UNARY]%
     :global_scope_array_delete<:none, cast_expr>
unary_expr := "::" 'delete' "[" expr "]"cast_expr ;    %[PREC
%UNARY]%
     :global_scope_array_delete<expr, cast_expr>
unary_expr := 'new' new_type_id ;            %[PREC %EMPTY]%
     :new<:none, new_type_id, :none>
unary_expr := 'new' new_type_id new_initializer ;
     :new<:none, new_type_id, new_initializer>
unary_expr := 'new' new_placement new_type_id ;    %[PREC
%EMPTY]%
     :new<new_placement, new_type_id, :none>
unary_expr := 'new' new_placement new_type_id new_initializer ;
     :new<new_placement, new_type_id, new_initializer>
unary_expr := 'delete' cast_expr ;           %[PREC
%UNARY]%
     :delete<cast_expr>
unary_expr := 'delete' "[" "]" cast_expr ;   %[PREC %UNARY]%
     :array_delete<:none, cast_expr>
unary_expr := 'delete' "[" expr "]"cast_expr ;    %[PREC
%UNARY]%
     :array_delete<expr, cast_expr>

new_placement := "(" exprlist ")" ;
     exprlist

new_initializer := "(" exprlist ")" ;
     exprlist

new_type_id := type_specifier_seq new_declarator ;
     :type_id<type_specifier_seq, new_declarator>
new_type_id := type_specifier_seq ;          %[PREC %EMPTY]%
     :type_id<type_specifier_seq, :none>
new_type_id := "(" type_id ")" ;
     :parenth_type<type_id>
new_type_id := "(" type_id ")" "[" expr "]";
     :new_gnu_type<type_id, expr>

new_declarator := "*" type_qualifiers new_declarator ;
     :ptr_dcltr<type_qualifiers,new_declarator>
new_declarator := "&" type_qualifiers new_declarator ;
```

```
     :ref_dcltr<type_qualifiers,new_declarator>
new_declarator := ptr_to_mem type_qualifiers
new_declarator ;
     :memptr_dcltr<ptr_to_mem, type_qualifiers,new_declarator>
new_declarator := "*" type_qualifiers ;           %[PREC
%EMPTY]%
     :ptr_dcltr<type_qualifiers,:none>
new_declarator := "&" type_qualifiers ;           %[PREC
%EMPTY]%
     :ref_dcltr<type_qualifiers,:none>
new_declarator := ptr_to_mem type_qualifiers ;    %[PREC
%EMPTY]%
     :memptr_dcltr<ptr_to_mem, type_qualifiers,:none>
new_declarator := direct_new_declarator ;    %[PREC
     %EMPTY]% direct_new_declarator
direct_new_declarator := "[" expr "]" ;
```

```
     :array_dcltr<:none, expr>
direct_new_declarator := direct_new_declarator "[" expr "]" ;
     :array_dcltr<direct_new_declarator, expr>

cast_expr := unary_expr ;
     unary_expr
cast_expr := "(" type_id ")" cast_expr ;     %[PREC %UNARY]%
     :cast<type_id,cast_expr>

-- WARNING : constant-expression are equal to expr_no_commas,
-- They should exclude assignment-expression and throw-
expression
expr_no_commas := cast_expr ;
     cast_expr
expr_no_commas := expr_no_commas "->*" expr_no_commas ;
     :pmap<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas ".*" expr_no_commas
;
     :pmpp<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "+" expr_no_commas ;
     :plus<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "-" expr_no_commas
;
     :minus<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "*" expr_no_commas
;
     :mul<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "/" expr_no_commas
;
     :div<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "%" expr_no_commas
;
     :rem<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "<<" expr_no_commas
;
     :lsh<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas ">>"
expr_no_commas ;
     :rsh<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "<" expr_no_commas
;
     :lt<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas ">" expr_no_commas
;
     :gt<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "<=" expr_no_commas
;
     :le<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas ">=" expr_no_commas
;
     :ge<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "==" expr_no_commas
;
     :eq<expr_no_commas.0,expr_no_commas.1>
```

```
expr_no_commas := expr_no_commas "!=" expr_no_commas
;
     :neq<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "&" expr_no_commas
;
     :bwand<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "|" expr_no_commas
;
     :bwor<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "^" expr_no_commas
;
     :bwxor<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "&&" expr_no_commas ;
     :and<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "||" expr_no_commas
;
     :or<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "?" expr ":" expr_no_commas ;
     :cond<expr_no_commas.0,expr,expr_no_commas.1>
expr_no_commas := expr_no_commas "=" expr_no_commas ;
     :ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "+=" expr_no_commas
;
```

```
     :plus_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "-=" expr_no_commas
;
     :minus_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "*=" expr_no_commas
;
     :mul_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "/=" expr_no_commas ;
     :div_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "%=" expr_no_commas
;
     :rem_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "&=" expr_no_commas
;
     :bwand_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "^=" expr_no_commas ;
     :bwxor_ass<expr_no_commas.0,expr_no_commas.1
> expr_no_commas := expr_no_commas "|="
expr_no_commas ;
     :bwor_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas "<<="
expr_no_commas ;
     :lsh_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := expr_no_commas ">>=" expr_no_commas ;
     :rsh_ass<expr_no_commas.0,expr_no_commas.1>
expr_no_commas := 'throw' ;
     :throw<:none>
expr_no_commas := 'throw' expr_no_commas ;
     :throw<expr_no_commas>

primary := overqualified_id ;
     overqualified_id
primary := %INTEGER ;
     :integer[%INTEGE
R] primary := %FLOAT
;
     :float[%FLOAT]
primary := %CHARACTER ;
     :character[%CHARACTER]
primary := 'true' ;
     :true
primary := 'false' ;
     :false
primary := string ;
     string
primary := exprmacro_call ;
     exprmacro_call
-- ne pas valider une expression entre paratheses
-- car toute expression simple peut etre entre parenthese
-- et cela peut induire des erreur de
syntaxe primary := "(" expr1 ")" ;
     :parenth_expr<expr1>
primary := primary "(" exprlist ")" ; %[PREC "."]%
     :call<primary,exprlist>
```

```
primary := primary "[" expr "]" ;  %[PREC "."]%
     :index<primary,expr>
primary := primary "." overqualified_id ;
     :dot<primary,overqualified_id>
primary := primary "->"
overqualified_id ;
     :arrow<primary,overqualified_id>
primary := primary "." template_overqualified_id ;
     :dot<primary,template_overqualified_id>
primary := primary "->" template_overqualified_id ;
     :arrow<primary,template_overqualified_id>
primary := primary "++" ;
```

```
     :post_incr<primary>
primary := primary "--" ;
     :post_decr<primar
y> primary := 'this' ;
     :this
primary := functional_cast ;
     functional_cast
primary := 'dynamic_cast' "<" type_id ">" "(" expr ")";
     :dynamic_cast<type_id, expr>
primary := 'static_cast' "<" type_id ">" "(" expr ")";
     :static_cast<type_id, expr>
primary := 'reinterpret_cast' "<" type_id ">" "(" expr ")";
     :reinterpret_cast<type_id, expr>
primary := 'const_cast' "<" type_id ">" "(" expr ")";
     :const_cast<type_id, expr>
primary := 'typeid' "(" type_id
")";
     :typeid<type_id>
primary := 'typeid' "(" expr ")";
     :typeid<expr>
primary := expr  preprocessor ;
     expr  preprocessor

functional_cast := simple_type_specifier "(" exprlist ")" ;
     :functional_cast<simple_type_specifier, exprlist>


end chapter ;

chapter

DECLARATIONS rules

recover decl ";";
decl := typed_declspecs initdecls ";" ;
     :decl<typed_declspecs,initdecls>
decl := declmods initdecls ";" ;
     :decl<declmods,initdecls>
decl := typed_declspecs ";" ;
     :decl<typed_declspecs,:none>
-- empty declaration
decl := declmods ";"
;
     :decl<declmods,:none>
decl := declmacro_call ";" ;       %[PREC %EMPTY]%
     :decl<declmacro_call, :none>
decl := declmacro_call ;           %[PREC
     %IDENT]% declmacro_call

-- Declspecs which contain at least one type specifier or typedef
name.
-- (Just `const' or `volatile' is not enough.)
-- A typedef'd name following these is taken as a name to be
declared. typed_declspecs := type_specifier reserved_declspecs ;
     reserved_declspecs:<type_specifier,..>
```

```
typed_declspecs := declmods type_specifier reserved_declspecs ;
     %{
        VTP_TreeP t3 = Parser_Pop();
        VTP_TreeP t2 = Parser_Pop();
        VTP_TreeP t1 = Parser_Pop();
```

```
        VTP_TreeAdopt(t1,t2,-1);
        while (VTP_TreeLength(t3) > 0) {
          VTP_TreeAdopt(t1,VTP_TreeDisown(t3,0),-1);
        }
        Parser_SetCoordNN(t1,t1,t3);
        VTP_TreeDestroy(t3);
        Parser_Push(t1);
     }%

type_specifier_seq := type_specifier ;
     :type_specifiers<type_specifier>
type_specifier_seq := type_qualifier ;
     :type_specifiers<type_qualifier>
type_specifier_seq := type_specifier_seq type_specifier ;
     type_specifier_seq:<.., type_specifier>
type_specifier_seq := type_specifier_seq type_qualifier ;
     type_specifier_seq:<.., type_qualifier>

decl_specifier_seq := type_specifier_seq ;
     %{
        VTP_TreeP t1 = Parser_Pop();
        VTP_TreeP t2 =
        VTP_TreeMake(C_1_op_declaration_specifiers);

        while (VTP_TreeLength(t1) > 0) {
          VTP_TreeAdopt(t2,VTP_TreeDisown(t1,0),-1);
        }
        Parser_SetCoordNN(t2,t1,t1);
        VTP_TreeDestroy(t1);
        Parser_Push(t2);
     }%

reserved_declspecs := ;
     :declaration_specifiers<>
reserved_declspecs := reserved_declspecs typespecqual_reserved ;
     reserved_declspecs:<..,typespecqual_reserved>
-- the <storage_class> is not at beginning of declaration
reserved_declspecs := reserved_declspecs storage_class ;
     reserved_declspecs:<..,storage_class>

-- List of just storage classes and type modifiers.
-- A declaration can start with just this, but then it cannot be
used
-- to redeclare a typedef-name.
declmod := type_qualifier ;
     type_qualifier
declmod := storage_class ;
     storage_class
declmods := declmod ;
     :declaration_specifiers<declm
od> declmods := declmods declmod ;
     declmods:<..,declmod>

-- Used instead of declspecs where storage classes are not
allowed
```

```
-- (that is, for typenames and structure components).
-- Don't accept a typedef-name if anything but a modifier
precedes it. typed_typespecs := type_specifier
reserved_typespecquals ;
     reserved_typespecquals:<type_specifier,..>
typed_typespecs := nonempty_type_quals
type_specifier reserved_typespecquals ;
```

```
     %{
        VTP_TreeP       t3       =
        Parser_Pop();    VTP_TreeP
        t2     =     Parser_Pop();
        VTP_TreeP       t1       =
        Parser_Pop();

        VTP_TreeAdopt(t1,t2,-
        1);
        while (VTP_TreeLength(t3) > 0) {
          VTP_TreeAdopt(t1,VTP_TreeDisown(t3,0),-1);
        }
        Parser_SetCoordNN(t1,t1,t3);
        VTP_TreeDestroy(t3);
        Parser_Push(t1);
     }%

reserved_typespecquals := ;
     :type_specifiers<>
reserved_typespecquals := reserved_typespecquals
typespecqual_reserved ;
     reserved_typespecquals:<..,typespecqual_reserved>

-- A type_specifier (but not a type qualifier).
-- Once we have seen one of these in a declaration,
-- if a typedef name appears then it is being redeclared.
type_specifier := typespec ;
     typespec
type_specifier := struct_specifier ;
     struct_specifier
type_specifier := overqualified_type_name ;
     overqualified_type_name
-- GNU C++ extension
type_specifier := 'typeof' "(" %[CBLOCK_BEGIN]% simple_expr
%[CBLOCK_END]% ")" ;
     :typeof<simple_expr>
type_specifier := 'typeof' "(" type_id ")" ;
     :typeof<type_id>
type_specifier := typemacro_call
;
     typemacro_call

-- A type_specifier that is a reserved word, or a type qualifier.
typespecqual_reserved := typespec ;
     typespec
typespecqual_reserved := type_qualifier ;
     type_qualifier
typespecqual_reserved := struct_specifier ;
     struct_specifier

initdecls := initdcl ;
     :initdecls<initdcl>
initdecls := initdecls "," initdcl ;
     initdecls:<..,initdcl>
```

```
initdcl := declarator "=" initializer ;
     :dcltr_affinit<declarator, initializer>
-- Ambiguous form: we can not decide if it is a function or a data
-- initdcl := declarator_id "(" identifier ")" ;
--   :dcltr_callinit<declarator_id,:exprlist<identifier>>
-- initdcl := simple_declarator "(" identifiers ")" ;
--
     :dcltr_callinit<simple_declarator,identifi
ers> initdcl := declarator "(" nonnull_exprlist
")";
```

```
     :dcltr_callinit<declarator,
nonnull_exprlist> initdcl := declarator ;
     :dcltr_noinit<declarator>
-- identifiers := identifier ;
--   :exprlist<identifier>
-- identifiers := identifiers identifier ;
--   identifiers:<.., identifier>

typespec := %TYPESPEC ;
     :typespec[%TYPESPEC]
-- WARNING extern is a keyword and a storage class
storage_class := 'extern' ;
     :storage_class["extern"]
storage_class := %STORAGECLASS
;
     :storage_class[%STORAGECLASS]
type_qualifier := %TYPEQUAL ;
     :type_qualifier[%TYPEQUAL]
access_specifier := %ACCESSPEC ;
     :access_specifier[%ACCESSPEC]

-- Initializers. initializer is the entry
point. initializer :=   simple_expr ;
     simple_expr
initializer := "{" %[CBLOCK_END]% %[CBLOCK_BEGIN]%
initlist_maybe_comma
"}" ;
     initlist_maybe_comma

-- `initlist_maybe_comma' is the guts of an initializer in braces.
-- ANSI C forbids empty initializer braces
initlist_maybe_comma := ;
     :initializer_list<>
initlist_maybe_comma := initlist1
     maybecomma ; initlist1

initlist1 := initializer ;
     :initializer_list<initializer>
initlist1 := initlist1 "," initializer ;
     initlist1:<..,initializer>

declarator2 := declarator2 "(" parmlist ")" type_qualifiers
exception_specification_opt ;
     :func_dcltr<declarator2,parmlist, type_qualifiers,
exception_specification_opt>
declarator2 := declarator2 "[" expr "]" ;
     :array_dcltr<declarator2,expr>
declarator2 := declarator2 "[" "]" ;
     :array_dcltr<declarator2,:none>
declarator := "*" type_qualifiers
declarator ;
     :ptr_dcltr<type_qualifiers,declarator>
declarator := "&" type_qualifiers declarator ;
     :ref_dcltr<type_qualifiers,declarator>
declarator := ptr_to_mem type_qualifiers
```

```
declarator ;
     :memptr_dcltr<ptr_to_mem,
type_qualifiers,declarator> declarator := declarator2
;
     declarator2
declarator2 := "(" declarator1 ")" ;
     :parenth_dcltr<declarator1>
declarator2 := qualified_id ;
```

```
     qualified_id
recover declarator1 ;
declarator1 := declarator ;
     declarator

ptr_to_mem := nested_name_specifier "*" ;
     nested_name_specifier
ptr_to_mem := global_scope_name_specifier "*" ;
     global_scope_name_specifier

struct_specifier := class_id "{" %[CBLOCK_END]% %[CBLOCK_BEGIN]%
%{C_TypeTablePushClass();}% component_decl_list
%{C_TypeTablePopClass();}% "}" %[CBLOCK_END]% %[CBLOCK_BEGIN]% ;
%[PREC
%EMPTY]%
     :class<class_id,component_decl_list>
struct_specifier := class_id ;     %[PREC
%IDENT]%
     :class<class_id, :none>
struct_specifier := class_head "{" %[CBLOCK_END]% %[CBLOCK_BEGIN]%
%{C_TypeTablePushClass();}%  component_decl_list
%{C_TypeTablePopClass();}% "}"
                             %[CBLOCK_END]% %[CBLOCK_BEGIN]%    ;
     :class<class_head,component_decl_list>
struct_specifier := 'enum' notype_identifier
%{C_Add_Type_Name();}% "{" enumlist maybecomma "}" %[CBLOCK_END]%
%[CBLOCK_BEGIN]%                                    ;
     :enum<notype_identifier,enumlist>
struct_specifier := 'enum' typename %{C_Add_Type_Name();}% "{"
enumlist maybecomma "}" %[CBLOCK_END]% %[CBLOCK_BEGIN]% ;
     :enum<typename,enumlist>
struct_specifier := 'enum' "{" enumlist maybecomma "}"
%[CBLOCK_END]%
%[CBLOCK_BEGIN]% ;
     :enum<:none,enumlist>
struct_specifier := 'enum' overqualified_type_name
%{C_Add_Type_Name();}%;
     :enum<overqualified_type_name,:none>
struct_specifier := 'typename'
overqualified_type_name
%{C_Add_Type_Name();}%;
     :typename_id<overqualified_type_name>

maybecomma := ;
     :none
maybecomma := ","
;
     :none


enumlist := ;
     :enumlist<>
enumlist := enumlist1
;
```

```
     enumlist1
enumlist1 := enumerator ;
     :enumlist<enumerator>
enumlist1 := enumlist1 "," enumerator ;
     enumlist1:<..,enumerator>

enumerator := identifier ;
     :enumerator<identifier,:none>
enumerator := identifier "=" simple_expr
;
     :enumerator<identifier,simple_expr>

--recover type_id ;
```

```
type_id := typed_typespecs absdcl ;
     :type_id<typed_typespecs,absdcl>
type_id := nonempty_type_quals absdcl
;
     :type_id<nonempty_type_quals,absdcl>

-- an abstract declarator
absdcl := ;
     :absdcltr
absdcl := absdcl1
;
     absdcl1

nonempty_type_quals := type_qualifier ;
     :type_specifiers<type_qualifier>
nonempty_type_quals := nonempty_type_quals
     type_qualifier ;
     nonempty_type_quals:<..,type_qualifier>

type_qualifiers := ;
     :none
type_qualifiers := type_qualifiers1 ;
     type_qualifiers1
type_qualifiers1 := type_qualifier ;
     :type_qualifiers<type_qualifier>
type_qualifiers1 := type_qualifiers1
type_qualifier ;
     type_qualifiers1:<.., type_qualifier>

-- a nonempty abstract declarator
recover parenth_absdcl ;
parenth_absdcl := absdcl1 ;
     absdcl1
absdcl1 := "(" parenth_absdcl ")" ;
     :parenth_dcltr<parenth_absdcl>
absdcl1 := "*" type_qualifiers absdcl1 ;%[PREC %UNARY]%
     :ptr_dcltr<type_qualifiers,absdcl1>
absdcl1 := "*" type_qualifiers ;   %[PREC %UNARY]%
     :ptr_dcltr<type_qualifiers,:absdcltr>
absdcl1 := "&" type_qualifiers absdcl1 ;%[PREC %UNARY]%
     :ref_dcltr<type_qualifiers,absdcl1>
absdcl1 := "&" type_qualifiers ;   %[PREC %UNARY]%
     :ref_dcltr<type_qualifiers,:absdcltr>
absdcl1 := ptr_to_mem type_qualifiers absdcl1 ; %[PREC %UNARY]%
     :memptr_dcltr<ptr_to_mem,
type_qualifiers,absdcl1> absdcl1 := ptr_to_mem
type_qualifiers ;                      %[PREC
%UNARY]%
     :memptr_dcltr<ptr_to_mem,
type_qualifiers,:absdcltr> absdcl1 := absdcl1 "("
parmlist ")" type_qualifiers
exception_specification_opt;  %[PREC "."]%
     :func_dcltr<absdcl1,parmlist, type_qualifiers,
exception_specification_opt>
```

```
absdcl1 := absdcl1 "[" expr "]" ;  %[PREC "."]%
     :array_dcltr<absdcl1,expr>
absdcl1 := absdcl1 "[" "]" ;       %[PREC "."]%
     :array_dcltr<absdcl1,:none>
absdcl1 := "(" parmlist ")" type_qualifiers
exception_specification_opt;
     %[PREC "."]%
     :func_dcltr<:absdcltr,parmlist,type_qualifiers,
exception_specification_opt>
absdcl1 := "[" expr "]" ;     %[PREC "."]%
     :array_dcltr<:absdcltr,expr>
absdcl1 := "[" "]" ;          %[PREC "."]%
```

```
     :array_dcltr<:absdcltr,:none>

end chapter ;

chapter STATEMENTS

rules


recover stmts ;
stmts := ;
     :stmts<>
stmts := %[CBLOCK_BEGIN]% stmts1 %[CBLOCK_END]% ;
     stmts1
recover stmts1 as stmts0 ;
stmts1 := stmts0 ;
     :stmts<stmts0>
stmts1 := stmts1 stmts0 ;
     stmts1:<..,stmts0>
stmts0 := stmt_or_decl_or_label %[CBLOCK_END]% %[CBLOCK_BEGIN]%
     ; stmt_or_decl_or_label

compstmt := open_blk stmts close_blk ;
     %{
       VTP_TreeP close = Parser_Pop();
       VTP_TreeP stmts = Parser_Pop();
       VTP_TreeP open = Parser_Pop();
       VTP_TreeP compound ;
       if ((VTP_TREE_OPERATOR(open) == C_1_op_none)
           && (VTP_TREE_OPERATOR(close) == C_1_op_none)) {
         compound = VTP_TreeMake(C_1_op_compound);
         VTP_TreeSetChild(compound,stmts,0);
         Parser_SetCoordNN(compound,open,close);
         VTP_TreeDestroy(open);
         VTP_TreeDestroy(close);
       } else {
         compound = VTP_TreeMake(C_1_op_block);
         VTP_TreeSetChild(compound,open,0);
         VTP_TreeSetChild(compound,stmts,1);
         VTP_TreeSetChild(compound,close,2);
         Parser_SetCoordNN(compound,open,close);
       }
       Parser_Push(compound);
     }%

open_blk := "{" %{C_TypeTablePushAuto();}%;
     :none
open_blk := openblockmacro_call
     ; openblockmacro_call
close_blk := "}" %{C_TypeTablePopAuto();}%;
     :none
close_blk := closeblockmacro_call ;
     closeblockmacro_call

loopstmt := openloopmacro_call stmts close_blk ;
```

```
     :loop<openloopmacro_call, stmts, close_blk>
loopstmt := open_blk stmts closeloopmacro_call ;
     :loop<open_blk, stmts, closeloopmacro_call>
```

```
loopstmt := openloopmacro_call stmts closeloopmacro_call ;
     :loop<openloopmacro_call, stmts, closeloopmacro_call>


recover labeled_stmt_list ";";
labeled_stmt_list := %[CBLOCK_BEGIN]% labeled_stmt_list1
     %[CBLOCK_END]% ; labeled_stmt_list1
labeled_stmt_list1 := stmt ;
     :stmts<stmt>
labeled_stmt_list1 := sql_stmt
;
     :stmts<sql_stmt>
labeled_stmt_list1 := label labeled_stmt_list1 ;
     labeled_stmt_list1:<label,..>
labeled_stmt_list1 := pp_dir labeled_stmt_list1 ;
     labeled_stmt_list1:<pp_dir,..>

stmt_or_decl_or_label := stmt ;
     %{
       Parser_Push(Cxx_StmtOrDecl(Parser_Pop(
                        )));
     }%
stmt_or_decl_or_label := decl %{C_Add_Typedef_Name();}%;
     %{
       Parser_Push(Cxx_DeclOrStmt(Parser_Pop(
                        )));
     }%
-- ANSI C forbids label at end of compound statement
stmt_or_decl_or_label := label ;
     label
stmt_or_decl_or_label := sql_stmt ;
       sql_stmt
stmt_or_decl_or_label := stmts  preprocessor ;
     stmts  preprocessor

-- Parse a single real statement, not including any labels.
-- recover stmt;
-- MG - 24 novembre 1999
stmt := namespace_def ;
     namespace_def
-- MG - 24 novembre 1999
stmt := using_decl ;
     using_decl
-- MG - 24 novembre 1999
stmt := 'asm' "(" string ")" ";" ;
     :asm<string>
stmt := compstmt ;
     compstmt
stmt := loopstmt
;
     loopstmt
stmt := expr_stmt ";" ;
     :expr_stmt<expr_stmt>
stmt := 'if' "(" condition ")" %[CBLOCK_BEGIN]% labeled_stmt_list
```

```
%[CBLOCK_END]%
          'else' %[CBLOCK_END]%%[CBLOCK_BEGIN]% labeled_stmt_list ;
     :if<condition,labeled_stmt_list.0,labeled_stmt_list.
1> stmt := 'if' "(" condition ")" %[CBLOCK_BEGIN]%
          labeled_stmt_list %[CBLOCK_END]% ;           %[PREC
          'if']%
     :if<condition,labeled_stmt_list,:none>
stmt := 'while' "(" condition ")"
%[CBLOCK_END]%
        %[CBLOCK_BEGIN]% labeled_stmt_list ;
```

```
     :while<condition,labeled_stmt_list>
stmt := 'do' labeled_stmt_list 'while' "(" condition ")" ";" ;
     :do<labeled_stmt_list,condition>
stmt := 'for' "(" for_init_stmt %[CBLOCK_END]% %[CBLOCK_BEGIN]%
null_condition ";" %[CBLOCK_END]% %[CBLOCK_BEGIN]% xexpr ")"
%[CBLOCK_END]%
          %[CBLOCK_BEGIN]% labeled_stmt_list ;
     :for<for_init_stmt,null_condition,xexpr,labeled_stmt_list>
stmt := 'for' "(" for_init_stmt %[CBLOCK_END]% %[CBLOCK_BEGIN]%
condition ";" %[CBLOCK_END]% %[CBLOCK_BEGIN]% xexpr ")"
%[CBLOCK_END]%
          %[CBLOCK_BEGIN]% labeled_stmt_list ;
     :for<for_init_stmt,condition,xexpr,labeled_stmt_list
> stmt := 'switch' "(" condition ")" %[CBLOCK_END]%
          %[CBLOCK_BEGIN]% labeled_stmt_list ;
     :switch<condition,labeled_stmt_list>
stmt := 'break' ";" ;
     :break
stmt := 'continue' ";" ;
     :continue
stmt := 'return' ";" ;
     :return<:none>
stmt := 'return' expr ";" ;
     :return<expr>
stmt := 'goto' notype_identifier ";" ;
     :goto<notype_identifier>
stmt := try_block ;
     try_block
stmt := ";" ;
     :empty_stmt
stmt := stmtmacro_call ;
     stmtmacro_call

-- Any kind of label, including jump labels and case labels.
-- ANSI C accepts labels only before statements, but we allow them
-- also at the end of a compound statement.
recover label ":";
label := 'case' %[CBLOCK_BEGIN]% simple_expr %[CBLOCK_END]% ":" ;
     :case<simple_expr>
label := 'default' ":" ;
     :default
label := notype_identifier ":" ;
     :label<notype_identifier>

xexpr := ;
     :none
xexpr := expr
;
     expr

for_init_stmt := ";" ;
     :none
for_init_stmt := expr ";" ;
     expr
```

```
for_init_stmt := decl
     %{C_Add_Typedef_Name();}%; decl

null_condition := ;
     :none
recover
condition;
```

```
condition := %[CBLOCK_BEGIN]% condition1
     %[CBLOCK_END]% ; condition1
condition1 := expr1 ;
     expr1
condition1 := %[CBLOCK_BEGIN]% condition_decl %[CBLOCK_END]% ;
     condition_decl
condition_decl := decl_specifier_seq condition_decl2 ;
     :decl<decl_specifier_seq, condition_decl2>
condition_decl1 := declarator "=" simple_expr ;
     :dcltr_affinit<declarator, simple_expr>
condition_decl2 := condition_decl1 ;
     :initdecls<condition_decl1>

end chapter ;

chapter PARAMETERS

rules

-- This is what appears inside the parens in a function
declarator. recover parmlist ;
parmlist := %[CBLOCK_BEGIN]% parmlist_1
     %[CBLOCK_END]% ; parmlist_1

-- This is what appears inside the parens in a function
declarator. parmlist_1 := ;
     :parmlist<>
-- error ANSI C requires a named argument before ...
parmlist_1 := "..." ;
     :var_parmlist<:parmlist<>>
parmlist_1 := parms ;
     parms
parmlist_1 := parms "," "..." ;
     :var_parmlist<parms>
parmlist_1 := parms "..." ;
     :var_parmlist<parms>

parms := parm ;
     :parmlist<parm>
parms := parms "," parm
;
     parms:<..,parm>

-- recover parm;
-- A single parameter declaration or parameter type name,
-- as found in a parmlist.
parm := typed_declspecs declarator parm_init ;
     :parm_decl<typed_declspecs,declarator,parm_init
> parm := typed_declspecs absdcl parm_init ;
     :parm_decl<typed_declspecs,absdcl,parm_init>
parm := declmods declarator parm_init ;
     :parm_decl<declmods,declarator,parm_init>
parm := declmods absdcl parm_init ;
     :parm_decl<declmods,absdcl,parm_init>
```

```
parm_init := ;
     :none
parm_init := "=" expr_no_commas ;
```

```
     expr_no_commas

end chapter;

chapter CLASS
rules

aggr := 'class' ;
     :class_kw
aggr := 'struct'
;
     :struct_kw
aggr := 'union' ;
     :union_kw


class_head := aggr ;
     :class_head<aggr, :none, :none>
class_id := aggr overqualified_type_name %{C_Add_Type_Name();}%
;
     :class_head<aggr, overqualified_type_name, :none>
class_head := class_id ":" base_class_list;
     %{
       VTP_TreeP base = Parser_Pop();
       VTP_TreeP id = Parser_Pop();
       VTP_TreeDestroySetChild(id, base, -
       1); Parser_Push(id);
     }%

recover base_class_list ;
base_class_list := base_class_list1
;
     base_class_list1
base_class_list1 := base_class ;
     :base_classes<base_class>
base_class_list1 := base_class_list1 ","
                                       base_class
     ; base_class_list1:<..,base_class>

base_class := overqualified_type_name ;
     :base_class<:none, overqualified_type_name>
base_class := access_list overqualified_type_name ;
     :base_class<access_list, overqualified_type_name>

access_list1 := access_specifier
     ; access_specifier
access_list1 := storage_class ;
     storage_class
access_list := access_list1 ;
     :access_list<access_list1>
access_list := access_list
access_list1 ;
     access_list:<.., access_list1>

recover component_decl_list;
```

```
component_decl_list :=
component_decl_list0 ;
     component_decl_lis
t0 component_decl_list0
:= ;
     :class_decls<>
component_decl_list0 := class_decls_section1 ;
     :class_decls<class_decls_section1>
class_decls_section1 := component_decl_list1 ;
     :class_decls_section<:none, component_decl_list1>
```

```
component_decl_list0 := component_decl_list0
     class_decls_section2 ; component_decl_list0:<..,
     class_decls_section2>
class_decls_section2 := access_specifier ":" %[CBLOCK_END]%
             %[CBLOCK_BEGIN]% component_decl_list2 ;
     :class_decls_section<access_specifier,
     component_decl_list2>

component_decl_list2 := ;
     :component_decl_list<>
component_decl_list2 :=
component_decl_list1;
     component_decl_list1
component_decl_list1 := component_decl ;
     :component_decl_list<component_decl>
component_decl_list1 := component_decl_list1
%[CBLOCK_END]%
          %[CBLOCK_BEGIN]% component_decl
     ;
     component_decl_list1:<..,component_d
     ecl>

component_decl := component_datadecl ;
     component_datadecl
recover component_datadecl ";";
component_datadecl := components ";" ;
     %{
       VTP_TreeP x0 = Parser_Pop();
       VTP_TreeP y0 = VTP_TreeMake(C_1_op_member_decl);
       VTP_TreeP y1 =
       VTP_TreeMake(C_1_op_declaration_specifiers);
       VTP_TreeSetChild(y0, y1, 0);
       VTP_TreeSetChild(y0, x0,
       1); Parser_SetCoordNT(y0,
       x0, $2);
       Parser_PopUntilToken($2);
       Parser_Push(y0);
       C_Add_Typedef_Name();
     }%
component_datadecl := typed_declspecs components ";" ;
     %{
       VTP_TreeP x1 = Parser_Pop();
       VTP_TreeP x0 = Parser_Pop();
       VTP_TreeP y0 = VTP_TreeMake(C_1_op_member_decl);
       VTP_TreeSetChild(y0, x0, 0);
       VTP_TreeSetChild(y0, x1, 1);
       Parser_SetCoordNT(y0, x0, $3);
       Parser_PopUntilToken($3);
       Parser_Push(y0);
       C_Add_Typedef_Name();
     }%
component_datadecl := typed_declspecs ";" ;
     %{
       VTP_TreeP x0 = Parser_Pop();
```

```
        VTP_TreeP y0 = VTP_TreeMake(C_1_op_member_decl);
        VTP_TreeP y1 = Parser_AtomTreeCreate(C_1_op_none,
        vtp_at_void,
"")
;       VTP_TreeSetChild(y0, x0, 0);
        VTP_TreeSetChild(y0, y1, 1);
        Parser_SetCoordNT(y0, x0, $2);
        Parser_PopUntilToken($2);
        Parser_Push(y0);
        C_Add_Typedef_Name();
      }%
component_datadecl := declmods components ";" ;
      %{
```

```
        VTP_TreeP x1 = Parser_Pop();
        VTP_TreeP x0 = Parser_Pop();
        VTP_TreeP y0 =
        VTP_TreeMake(C_1_op_member_decl);
        VTP_TreeSetChild(y0, x0, 0);
        VTP_TreeSetChild(y0, x1, 1);
        Parser_SetCoordNT(y0, x0, $3);
        Parser_PopUntilToken($3);
        Parser_Push(y0);
        C_Add_Typedef_Name();
     }%
component_datadecl := declmods ";" ;
     %{
        VTP_TreeP x0 = Parser_Pop();
        VTP_TreeP y0 = VTP_TreeMake(C_1_op_member_decl);
        VTP_TreeP y1 = Parser_AtomTreeCreate(C_1_op_none,
        vtp_at_void,
"")
;       VTP_TreeSetChild(y0, x0, 0);
        VTP_TreeSetChild(y0, y1, 1);
        Parser_SetCoordNT(y0, x0, $2);
        Parser_PopUntilToken($2);
        Parser_Push(y0);
        C_Add_Typedef_Name();
     }%
component_decl := fndef ;
     fndef
component_decl := component_decl_list  preprocessor ;
     component_decl_list  preprocessor
component_decl := declmacro_call ";" ;%[PREC %EMPTY]%
     :member_decl<declmacro_call, :none>
component_decl := declmacro_call ;      %[PREC %IDENT]%
     declmacro_call
component_decl := using_decl ;
     using_decl
-- MG - 23 novembre 1999
component_decl :=
template_def;
     template_def
component_decl := explicit_instanciation;
     explicit_instanciation
component_decl :=
     explicit_specialization;
     explicit_specialization
component_decl := ";" ;
     :none

components := component_declarator ;
     :initdecls<component_declarator>
components := components "," component_declarator ;
     components:<..,component_declarator>

component_declarator := declarator ;
     :dcltr_noinit<declarator> ;
```

```
component_declarator := declarator "="
simple_expr ;
     :dcltr_affinit<declarator, simple_expr>
component_declarator := declarator ":"
simple_expr ;
     :bit_field<declarator,simple_expr>
component_declarator := ":" simple_expr ;
     :bit_field<:none,simple_ex

pr> end chapter;
```

```
chapter MEMBER_FUNCTION
rules

operator_name := 'operator' "*" ;
      :operator<:op_mult>
operator_name := 'operator' "/"
;
      :operator<:op_div>
operator_name := 'operator' "%"
;
      :operator<:op_mod>
operator_name := 'operator' "+"
;
      :operator<:op_plus>
operator_name := 'operator' "-"
;
      :operator<:op_minus>
operator_name := 'operator' "&"
;
      :operator<:op_bwand>
operator_name := 'operator' "|"
;
      :operator<:op_bwor>
operator_name := 'operator' "^"
;
      :operator<:op_bwxor>
operator_name := 'operator' "~"
;
      :operator<:op_bwnot>
operator_name := 'operator' ","
;
      :operator<:op_comma>
operator_name := 'operator' "==" ;
      :operator<:op_eq>
operator_name := 'operator' "<"
;
      :operator<:op_less>
operator_name := 'operator' ">"
;
      :operator<:op_greater>
operator_name := 'operator' "<=" ;
      :operator<:op_leq>
operator_name := 'operator' ">="
;
      :operator<:op_geq>
operator_name := 'operator' "!="
;
      :operator<:op_neq>
operator_name := 'operator' "+=" ;
      :operator<:op_ass_plus>
operator_name := 'operator' "-="
;
      :operator<:op_ass_minus>
```

```
 operator_name := 'operator' "*="
 ;
      :operator<:op_ass_mult>
operator_name := 'operator' "/=" ;
      :operator<:op_ass_div>
 operator_name := 'operator' "%="
 ;
      :operator<:op_ass_mod>
operator_name := 'operator' "^=" ;
      :operator<:op_ass_bwxor>
 operator_name := 'operator' "&="
 ;
      :operator<:op_ass_bwand>
 operator_name := 'operator' "|="
 ;
      :operator<:op_ass_bwor>
 operator_name := 'operator' "<<="
 ;
      :operator<:op_ass_lshift>
 operator_name := 'operator' ">>="
 ;
      :operator<:op_ass_rshift>
 operator_name := 'operator' "="
 ;
      :operator<:op_ass>
```

```
operator_name := 'operator' "<<" ;
      :operator<:op_lshift>
operator_name := 'operator' ">>"
;
      :operator<:op_rshift>
operator_name := 'operator' "++"
;
      :operator<:op_incr>
operator_name := 'operator' "--"
;
      :operator<:op_decr>
operator_name := 'operator' "&&"
;
      :operator<:op_and>
operator_name := 'operator' "||"
;
      :operator<:op_or>
operator_name := 'operator' "!"
;
      :operator<:op_not>
operator_name := 'operator' "?" ":" ;
      :operator<:op_cond>
operator_name := 'operator' "->"
;
      :operator<:op_arrow>
operator_name := 'operator' "->*"
;
      :operator<:op_pmap>
operator_name := 'operator' "(" ")"
;
      :operator<:op_parenth>
operator_name := 'operator' "[" "]"
;
      :operator<:op_croch>
operator_name := 'operator' 'new' ;
      :operator<:op_new>
operator_name := 'operator' 'new' "[" "]" ;
      :operator<:op_new_array>
operator_name := 'operator' 'delete'
;
      :operator<:op_delete>
operator_name := 'operator' 'delete' "[" "]" ;
      :operator<:op_delete_array>
operator_name := 'operator' conversion_type_id ;
      :operator<conversion_type_id>

conversion_type_id := type_specifier_seq conversion_declarator ;
      :type_id<type_specifier_seq,
conversion_declarator> conversion_declarator := ;
                                        %[PREC
%EMPTY]%
      :none
conversion_declarator := "*" type_qualifiers conversion_declarator ;
      :ptr_dcltr<type_qualifiers,conversion_declarator>
```

```
conversion_declarator := "&" type_qualifiers conversion_declarator ;
     :ref_dcltr<type_qualifiers,conversion_declarator>
conversion_declarator := ptr_to_mem type_qualifiers
conversion_declarator
;
     :memptr_dcltr<ptr_to_mem,
     type_qualifiers,conversion_declarator>

ctor_initializer := ":" ;
     :member_init_list<>
ctor_initializer := ":" member_init_list ;
     member_init_list
recover member_init_list ;
member_init_list := member_init_list1
;
     member_init_list1
member_init_list1 :=
member_init ;
     :member_init_list<member_init>
member_init_list1 := member_init_list1 "," member_init ;
     member_init_list1:<.., member_init>
```

```
member_init := "(" exprlist ")" ;
     :member_init<:none, exprlist>
member_init := qualified_ident_or_type "(" exprlist ")" ;
     :member_init<qualified_ident_or_type,

exprlist> end chapter;

chapter TEMPLATE
rules

export := 'export' ;
     :export

-- MG - 24/11/99 - remplacement de fndef et datadef par extdef
template_def := template_header extdef ;
     :template_def<:none, template_header, extdef>
template_def := export template_header extdef ;
     :template_def<export, template_header, extdef>

recover template_arg_list ;
template_arg_list := ;
     :template_arg_list<>
template_arg_list :=
template_arg_list1 ;
     template_arg_list1
template_arg_list1 := template_arg %{/*[*/ C_Template_Arg_Pop();
/*]*/}%
;
     :template_arg_list<template_arg>
template_arg_list1 := template_arg_list1 "," template_arg %{/*[*/
C_Template_Arg_Pop(); /*]*/}%;
     template_arg_list1:<.., template_arg>
template_arg := type_id ;
     type_id
template_arg := expr_no_commas ;   %[PREC
     %UNARY]% expr_no_commas

explicit_instanciation := 'template' extdef ;
     :explicit_instanciation<extdef>

explicit_specialization := 'template' "<" ">" extdef ;
     :explicit_specialization<extdef>

template_header := 'template' "<" template_parm_list ">"
%[CBLOCK_END]%
%[CBLOCK_BEGIN]% ;
     template_parm_list

template_parm_list := template_parm ;
     :template_parms<template_parm>
template_parm_list := template_parm_list ","
     template_parm ;
     template_parm_list:<..,template_parm>

template_parm := type_parameter ;
```

```
     type_parameter
template_parm := parm ;
     parm

type_parameter := parm_class_head type_parm_init ;
     :type_parameter<parm_class_head,type_parm_init>
type_parameter := parm_typename type_parm_init ;
```

```
     :type_parameter<parm_typename,type_parm_init>
type_parameter := template_header parm_class_head type_parm_init ;
     :template_parameter<template_header,
parm_class_head,type_parm_init>
parm_class_head := aggr ;
     :class_head<aggr,:none,:none>
parm_class_head := aggr identifier %{C_Add_Type_Name();}%;
     :class_head<aggr,identifier,:none>
parm_typename := 'typename' ;
     :typename_id<:none>
parm_typename := 'typename' identifier ;
     :typename_id<identifier>

type_parm_init := ;
     :none
type_parm_init := "=" type_specifier ;
     type_specifier
type_parm_init := "=" type_id ;
     type_id

end chapter;

chapter NAMESPACE
rules

namespace_def := 'namespace' identifier "{"
%{C_TypeTablePushClass();}% comp_extdefs %{C_TypeTablePopClass();}%
"}" ;
     %{
       VTP_TreeP x1 = Parser_Pop();
       VTP_TreeP x0;
       VTP_TreeP y0 = VTP_TreeMake(C_1_op_namespace_def);
          /* keep x0 on the stack for
       C_Add_Namespace_Name() */ C_Add_Namespace_Name();
       x0 = Parser_Pop();
       VTP_TreeSetChild(y0, x0,
       0);
       VTP_TreeSetChild(y0, x1, 1);
       Parser_SetCoordTT(y0, $1, $7);
       Parser_PopUntilToken($1);
       Parser_Push(y0);
     }%
namespace_def := 'namespace' "{" %{C_TypeTablePushClass();}%
comp_extdefs
%{C_TypeTablePopClass();}% "}" ;
     :namespace_def<:none, comp_extdefs>
namespace_def := 'namespace' identifier "=" any_id ";" ;
     :namespace_def<identifier, any_id>

using_decl := 'using' 'namespace' any_id
                                        %{C_Add_Using_Directive()
;}% ";" ;
     :using_directive<any_id>
using_decl := 'using' any_id ";" ;
     :using_declaration<any_id>
```

```
end chapter;

chapter EXCEPTION
rules

exception_specification_opt

:= ;
```

```
     :none
exception_specification_opt := 'throw' "(" raise_types ")" ;
     :exception_spec<raise_types>

recover raise_types;
raise_types := ;
     :type_list<>
raise_types :=
raise_types1;
     raise_types1
raise_types1 := type_id ;
     :type_list<type_id>
raise_types1 := raise_types1 "," type_id ;
     raise_types1:<..,type_id>

try_block := 'try' compstmt handler_seq ;
     :try_block<compstmt, handler_seq>

fntry_block := 'try' fnbody handler_seq ;
     :try_block<fnbody, handler_seq>
fntry_block := 'try' ctor_init_body handler_seq ;
     :try_block<ctor_init_body, handler_seq>
ctor_init_body := ctor_initializer fnbody ;
       :ctor_initializer<ctor_initializer,fnbody>

handler_seq := ;
     :try_handlers<>
handler_seq := handler_seq catch_handler ;
     handler_seq:<.., catch_handler>
catch_handler := 'catch' handler_args compstmt ;
      :handler<handler_args, compstmt>

handler_args := "(" "..." ")" ;
     :var_parmlist<:parmlist<>>
handler_args := "(" parm ")" ;
     :parmlist<par

m> end chapter;

chapter PREPROCESSOR

--
-- MODELE pour gerer le preprocesseur a un endroit ou les
-- #if* doiuvent contenir des elements reconnus par la regle
$RULE
--
-- chapter $RULE  PREPROCESSOR
-- rules
--
-- $RULE  preprocessor := pp_dir ;
--        pp_dir
-- $RULE  preprocessor := $RULE  pp_conditional ;
--        $RULE  pp_conditional
--
-- $RULE  pp_conditional := pp_if
```

```
--                              $RULE
--                              $RULE  pp_elif_parts
--                              $RULE  pp_else_part
--                              pp_endif ;
--          :pp_conditional<pp_if,
```

```
--                        $RULE,
--                        $RULE  pp_elif_parts,
--                        $RULE  pp_else_part,
--                        pp_endif>
--
-- $RULE  pp_elif_parts := ;
--         :none
-- $RULE  pp_elif_parts := $RULE  pp_elif_parts_ne ;
--         $RULE  pp_elif_parts_ne
--
-- $RULE  pp_elif_parts_ne := $RULE  pp_elif_part ;
--         :pp_elif_parts<$RULE  pp_elif_part>
-- $RULE  pp_elif_parts_ne := $RULE  pp_elif_parts_ne $RULE
pp_elif_part
;
--         $RULE  pp_elif_parts_ne:<..,$RULE  pp_elif_part>
--
-- $RULE  pp_elif_part := pp_elif $RULE ;
--         :pp_elif_part<pp_elif,$RULE>
--
-- $RULE  pp_else_part := ;
--         :none
-- $RULE  pp_else_part := pp_else $RULE ;
--         :pp_else_part<pp_else,$RULE>
--
-- end chapter ;

chapter EXTDEFS  PREPROCESSOR
rules

extdefs  preprocessor := pp_dir ;
       pp_dir
extdefs  preprocessor := extdefs  pp_conditional ;
       extdefs  pp_conditional

extdefs  pp_conditional := pp_if
                      pp_extdefs
                      extdefs  pp_elif_parts
                      extdefs  pp_else_part
                      pp_endif ;
       :pp_conditional<pp_if,
                      pp_extdefs,
                      extdefs  pp_elif_parts,
                      extdefs  pp_else_part,
                      pp_endif>

extdefs  pp_elif_parts := ;
       :none
extdefs  pp_elif_parts := extdefs  pp_elif_parts_ne ;
       extdefs  pp_elif_parts_ne

extdefs  pp_elif_parts_ne := extdefs  pp_elif_part ;
       :pp_elif_parts<extdefs  pp_elif_part>
extdefs  pp_elif_parts_ne := extdefs  pp_elif_parts_ne
extdefs  pp_elif_part ;
```

```
        extdefs  pp_elif_parts_ne:<..,extdefs  pp_elif_part>

extdefs  pp_elif_part := pp_elif pp_extdefs ;
        :pp_elif_part<pp_elif,pp_extdefs>
```

```
extdefs  pp_else_part := ;
        :none
extdefs  pp_else_part := pp_else pp_extdefs ;
        :pp_else_part<pp_else,pp_extdefs>

-- ANSI C forbids an empty source
file recover pp_extdefs as extdef;
pp_extdefs := ;
     :extdefs<>
pp_extdefs := pp_extdefs extdef ;
     pp_extdefs:<..,extdef>

end chapter;

chapter COMPONENT_DECL_LIST  PREPROCESSOR
rules

component_decl_list  preprocessor := pp_dir ;
        pp_dir
component_decl_list  preprocessor := component_decl_list
pp_conditional
;
        component_decl_list  pp_conditional

component_decl_list  pp_conditional := pp_if
                       component_decl_list
                       component_decl_list  pp_elif_parts
                       component_decl_list  pp_else_part
                       pp_endif ;
        :pp_conditional<pp_if,
                      component_decl_list,
                      component_decl_list  pp_elif_parts,
                      component_decl_list  pp_else_part,
                      pp_endif>

component_decl_list  pp_elif_parts := ;
        :none
component_decl_list  pp_elif_parts :=
component_decl_list  pp_elif_parts_ne
;
        component_decl_list  pp_elif_parts_ne

component_decl_list  pp_elif_parts_ne :=
component_decl_list  pp_elif_part ;
        :pp_elif_parts<component_decl_list  pp_elif_part>
component_decl_list  pp_elif_parts_ne :=
component_decl_list  pp_elif_parts_ne component_decl_list
pp_elif_part ;

component_decl_list  pp_elif_parts_ne:<..,component_decl_list
pp_elif_pa rt>

component_decl_list  pp_elif_part := pp_elif component_decl_list ;
        :pp_elif_part<pp_elif,component_decl_list>
```

```
component_decl_list  pp_else_part := ;
        :none
component_decl_list  pp_else_part := pp_else component_decl_list ;
        :pp_else_part<pp_else,component_decl_list>
```

```
end chapter;

chapter STMTS
PREPROCESSOR rules

stmts  preprocessor := pp_dir
        ; pp_dir
stmts  preprocessor := stmts  pp_conditional
        ; stmts  pp_conditional

-- ATTENTION en raison de l'ambiguite sur les #if le recouvrement
d'erreur ne marche pas
-- recover stmts__pp_conditional pp_endif ;
stmts  pp_conditional := pp_if
                        pp_stmts
                        stmts
                        pp_elif_parts
                        stmts
                        pp_else_part
                        pp_endif ;
        :pp_conditional<pp_if,
                       pp_stmts,
                       stmts
                       pp_elif_parts,
                       stmts
                       pp_else_part,
                       pp_endif>

stmts  pp_elif_parts := ;
        :none
stmts  pp_elif_parts := stmts  pp_elif_parts_ne ;
        stmts  pp_elif_parts_ne

stmts  pp_elif_parts_ne := stmts  pp_elif_part ;
        :pp_elif_parts<stmts  pp_elif_part>
stmts  pp_elif_parts_ne := stmts  pp_elif_parts_ne stmts
        pp_elif_part ; stmts  pp_elif_parts_ne:<..,stmts
        pp_elif_part>

stmts  pp_elif_part := pp_elif pp_stmts ;
        :pp_elif_part<pp_elif,pp_stmts>

stmts  pp_else_part := ;
        :none
stmts  pp_else_part := pp_else pp_stmts ;
        :pp_else_part<pp_else,pp_stmts>

pp_stmts := ;
     :stmts<>
pp_stmts := %[CBLOCK_BEGIN]% pp_stmts1 %[CBLOCK_END]% ;
     pp_stmts1
recover pp_stmts1 as stmt_or_decl_or_label
; pp_stmts1 := stmt_or_decl_or_label ;
     :stmts<stmt_or_decl_or_label>
pp_stmts1 := pp_stmts1 %[CBLOCK_END]% %[CBLOCK_BEGIN]%
```

```
stmt_or_decl_or_label ;
     pp_stmts1:<..,stmt_or_decl_or_lab

el> end chapter;

chapter EXPR  PREPROCESSOR
rules
```

```
expr  preprocessor := expr  pp_conditional
       ; expr  pp_conditional

expr  pp_conditional := pp_if
                      xexpr
                      expr  pp_elif_parts
                      expr  pp_else_part
                      pp_endif ;
       :pp_conditional<pp_if,
                     xexpr,
                     expr  pp_elif_parts,
                     expr  pp_else_part,
                     pp_endif>

expr  pp_elif_parts := ;
       :none
expr  pp_elif_parts := expr  pp_elif_parts_ne ;
       expr  pp_elif_parts_ne

expr  pp_elif_parts_ne := expr  pp_elif_part ;
       :pp_elif_parts<expr  pp_elif_part>
expr  pp_elif_parts_ne := expr  pp_elif_parts_ne expr
       pp_elif_part ; expr  pp_elif_parts_ne:<..,expr
       pp_elif_part>

expr  pp_elif_part := pp_elif xexpr ;
       :pp_elif_part<pp_elif,xexpr>

expr  pp_else_part := ;
       :none
expr  pp_else_part := pp_else xexpr ;
       :pp_else_part<pp_else,xexpr>

end chapter;

rules

pp_dir := pp_define ;
     pp_define
pp_dir := pp_undef ;
     pp_undef
pp_dir := pp_line ;
     pp_line
pp_dir := pp_error ;
     pp_error
pp_dir := pp_pragma ;
     pp_pragma
pp_dir := pp_include ;
     pp_include
pp_dir := pp_none ;
     pp_none

pp_define := %PPDEFINE pp_macro_name pp_macro_args pp_text ;
     :pp_define<pp_macro_name,pp_macro_args,pp_text>
```

```
pp_undef := %PPUNDEF pp_macro_name pp_text ;
     :pp_undef<pp_macro_name,pp_te

xt> pp_macro_name := %PPMACRONAME

;
```

```
     :identifier[%PPMACRONAME]

pp_macro_args := ;
     :none
pp_macro_args := %PPDEBARGS pp_macro_arg_list %PPENDARGS ;
     pp_macro_arg_list
pp_macro_arg_list := ;
     :pp_macro_args<>
pp_macro_arg_list := pp_macro_arg_list_ne ;
     pp_macro_arg_list_ne

pp_macro_arg_list_ne := identifier ;
     :pp_macro_args<identifier>
pp_macro_arg_list_ne := pp_macro_arg_list_ne "," identifier ;
     pp_macro_arg_list_ne:<..,identifier>

declmacro_call := %DECLMACRO ;
     :macro_call[%DECLMACRO]
stringmacro_call :=
%STRINGMACRO ;
     :macro_call[%STRINGMACRO]
exprmacro_call := %EXPRMACRO ;
     :macro_call[%EXPRMACRO]
typemacro_call := %TYPEMACRO ;
     :macro_call[%TYPEMACRO]
openblockmacro_call := %OPENBLOCKMACRO
%{C_TypeTablePushAuto();}%;
     :macro_call[%OPENBLOCKMACRO]
closeblockmacro_call := %CLOSEBLOCKMACRO
%{C_TypeTablePopAuto();}%;
     :macro_call[%CLOSEBLOCKMACRO]
openloopmacro_call := %OPENLOOPMACRO
%{C_TypeTablePushAuto();}%;
     :macro_call[%OPENLOOPMACRO]
closeloopmacro_call := %CLOSELOOPMACRO
%{C_TypeTablePopAuto();}%;
     :macro_call[%CLOSELOOPMACRO]
stmtmacro_call := %STMTMACRO ;
     :macro_call[%STMTMACRO]

pp_if := %PPIF pp_text ;
     :pp_if<pp_text>
pp_if := %PPIFDEF pp_ifdef_ident pp_text ;
     :pp_ifdef<pp_ifdef_ident,pp_text>
pp_if := %PPIFNDEF pp_ifdef_ident
pp_text ;
     :pp_ifndef<pp_ifdef_ident,pp_text>

pp_ifdef_ident := %PPIFDEFIDENT ;
     :identifier[%PPIFDEFIDENT]

pp_elif := %PPELIF pp_text ;
     :pp_elif<pp_text>
pp_else := %PPELSE pp_text
```

```
;
     :pp_else<pp_text>
pp_endif := %PPENDIF pp_text ;
     :pp_endif<pp_text>
pp_line := %PPLINE pp_text
;
     :pp_line<pp_text>
pp_error := %PPERROR
pp_text ;
     :pp_error<pp_text>
pp_pragma := %PPPRAGMA pp_text ;
     :pp_pragma<pp_text>
```

```
pp_include := %PPINCLUDE pp_filename pp_text ;
     :pp_include<pp_filename,pp_text>

pp_filename := ;
     :none
pp_filename := %PPEXTFILE ;
--   :pp_external_file[%PPEXTFILE]
     %{
       VTP_TreeP y0 =
Parser_AtomTreeCreate(C_1_op_pp_external_file, vtp_at_string,
Parser_GetGeneric($1));
       C_LoadTypeTable(Parser_GetGeneric($1), 0);
       Parser_SetCoordTT(y0, $1,
       $1);
       Parser_PopUntilToken($1);
       Parser_Push(y0);
     }%
pp_filename := %PPLOCFILE ;
--   :pp_local_file[%PPLOCFILE]
     %{
       VTP_TreeP y0 =
Parser_AtomTreeCreate(C_1_op_pp_local_file, vtp_at_string,
Parser_GetGeneric($1));
       C_LoadTypeTable(Parser_GetGeneric($1),
       1); Parser_SetCoordTT(y0, $1, $1);
       Parser_PopUntilToken($1);
       Parser_Push(y0);
     }%

pp_none := %PPNONE pp_text ;
     :pp_none<pp_text>

pp_text := pp_text_line;
     :pp_text<pp_text_line>

pp_text := pp_text
     pp_text_line ;
     pp_text:<..,pp_text_line>

pp_text_line := %PPTEXTLINE ;
     :pp_text_line[%PPTEXTLIN

E] end chapter ;

chapter SQL

rules
sql_stmt := %SQLSTART sql_stmt1
       ";" ; sql_stmt1
recover sql_stmt1;
sql_stmt1 := sql_type sql_lines ;
       :sql_stmt<sql_type,
sql_lines> sql_type := %SQLMACRO ;
       :identifier[%SQLMACRO
]; sql_lines := ;
```

```
        :sql_lines<>
sql_lines := sql_lines sql_line;
        sql_lines:<.., sql_line>
sql_line := %SQLLINE ;
        :sql_line[%SQLLINE

]; end chapter ;
```

```
end chapter ;

end definition
```

4. Java

Abstract syntax

```
definition of JAVA version 1 is

chapter ABSTRACT_SYNTAX

chapter TOP
  abstract syntax

  comp_unit -> PACKAGE_STMT IMPORT_STMT_LIST TYPE_DECL_LIST ;
  package_stmt -> MODIFIER_LIST NAME;
  import_stmt_list -> IMPORT_STMT *;
  import_stmt -> STATIC_OPT
  NAME_OR_GEN_NAME; type_decl_list ->
  TYPE_DECL * ;

  PACKAGE_STMT := none package_stmt ;
  IMPORT_STMT_LIST := import_stmt_list
  ; IMPORT_STMT := import_stmt;
  TYPE_DECL_LIST := type_decl_list ;
  TYPE_DECL := CLASS_DECL
  INTERFACE_DECL;
end chapter;


chapter CLASSES
  abstract syntax

  class_decl -> CLASS_HEAD TYPE_PARAM_LIST CLASS_EXT
  FIELD_DECL_LIST;
  class_head -> MODIFIER_LIST IDENTIFIER;
  class_ext -> EXTENDS INTERFACE_LIST;

  interface_decl -> INTERFACE_HEAD TYPE_PARAM_LIST
INTERFACE_EXT FIELD_DECL_LIST ;
  interface_head -> MODIFIER_LIST IDENTIFIER;
  field_decl_list -> FIELD_DECL *;

  extends -> NAME;
  extends_list -> NAME +;
  interface_list -> NAME
  +;
  variable_decl -> MODIFIER_LIST TYPE
  VAR_DECLARATOR_LIST ; var_dcltr_list ->
  VAR_DECLARATOR +;
  array_type -> SIMPLE_TYPE ARRAY_DCLTR ;
  var_dcltr_noinit -> IDENTIFIER ARRAY_DCLTR ;
  var_dcltr_affinit -> IDENTIFIER ARRAY_DCLTR INITIALIZER ;
  var_dcltr_foreach -> IDENTIFIER ARRAY_DCLTR EXPR ;
```

```
  modifier_list -> MODIFIER *;
  array_initializers -> INITIALIZER_LIST ;
  initializer_list -> INITIALIZER *;
  method_decl -> MODIFIER_LIST TYPE_PARAM_LIST RESULT_TYPE
METHOD_DECLARATOR THROW_LIST METHOD_BODY ;
  method_dcltr -> IDENTIFIER PARAMETER_LIST ARRAY_DCLTR ;
  parameter_list -> PARAMETER *;
```

```
-- change for java.1.1 : modifiers in
  param_decl param_decl -> MODIFIER_LIST TYPE
  PARAM_DCLTR ;
  var_arg_decl -> MODIFIER_LIST TYPE PARAM_DCLTR ;
  type_param_list -> TYPE_PARAM *;
  type_param -> IDENTIFIER BOUND_LIST ;
  throw_list -> NAME *;
  static_initializer -> COMPOUND
  ; object_initializer ->
  COMPOUND ; array_dcltr ->
  ARRAY_BRACKET + ;
  array_bracket -> implemented as void; -- '[]'
  bound_list -> TYPE *;

  CLASS_DECL := class_decl enum_decl;
  CLASS_HEAD := class_head;
  TYPE_PARAM_LIST := type_param_list;
  CLASS_EXT := class_ext;
  CLASS_MODIFIERS := modifier_list;
  EXTENDS := none extends ;
  EXTENDS_LIST := none extends_list;
  INTERFACE_LIST := none
  interface_list;
  INTERFACE_DECL := interface_decl annotation_type_decl ;
  INTERFACE_HEAD := interface_head;
  INTERFACE_EXT := extends_list;
  FIELD_DECL_LIST := field_decl_list;
-- constructors are managed as methods
-- FIELD_D := variable_decl method_decl static_initializer ;
-- changes for java.1.1 : nested classes and interfaces
  FIELD_DECL := variable_decl method_decl
  static_initializer
object_initializer TYPE_DECL;
  PARAMETER := param_decl
  var_arg_decl; PARAM_DCLTR :=
  var_dcltr_noinit; TYPE_PARAM :=
  type_param ; MODIFIER_LIST :=
  modifier_list ;
  MODIFIER := annotation public private protected static 'abstract'
final native synchronized transient threadsafe volatile;
  VAR_DECLARATOR_LIST := var_dcltr_list ;
  VAR_DECLARATOR := var_dcltr_noinit var_dcltr_affinit
  var_dcltr_foreach; RESULT_TYPE := TYPE void_type none;
  CLASS_TYPE := TYPE void_type ;
  TYPE := SIMPLE_TYPE array_type
  ;
  SIMPLE_TYPE := PRIMITIVE_TYPE NAME ;
  PRIMITIVE_TYPE := int float boolean char byte short long double ;
  ARRAY_DCLTR := array_dcltr none ;
  ARRAY_BRACKET := array_bracket ;
  INITIALIZER_LIST := initializer_list;
  INITIALIZER := EXPR array_initializers ;
  METHOD_DECLARATOR := method_dcltr ;
-- used in array_dcltr to simplify the abstract syntax,
```

```
-- the allowed operators depend on fathers,
-- solved by concrete syntax.
  THROW_LIST := none throw_list ;
  PARAMETER_LIST := parameter_list ;
  BOUND_LIST := bound_list ;

  METHOD_BODY := none compound;

end chapter;
```

```
chapter
  EXPRESSIONS
  abstract syntax

  expr_list -> EXPR * ;

-- arithmetic
  operators plus ->
  EXPR EXPR ;
  minus -> EXPR EXPR
  ; mul -> EXPR EXPR
  ; div -> EXPR EXPR
  ; rem -> EXPR EXPR
  ;
-- relational & logical
  operators lt -> EXPR EXPR ;
  gt -> EXPR EXPR ;
  ge -> EXPR EXPR ;
  le -> EXPR EXPR ;
  eq -> EXPR EXPR ;
  neq -> EXPR EXPR
  ; and -> EXPR
  EXPR ; or -> EXPR
  EXPR ;
-- the ternair operator
  cond -> EXPR EXPR EXPR
  ;
-- bit operators
  lsh -> EXPR EXPR ;
  rsh -> EXPR EXPR ;
  rrsh -> EXPR EXPR ;
  bwand -> EXPR EXPR ;
  bwor -> EXPR EXPR ;
  bwxor -> EXPR EXPR ;
-- assignment operators
  ass -> EXPR EXPR ;
  plus_ass -> EXPR EXPR ;
  minus_ass -> EXPR EXPR ;
  mul_ass -> EXPR EXPR ;
  div_ass -> EXPR EXPR ;
  rem_ass -> EXPR EXPR ;
  bwand_ass -> EXPR EXPR ;
  bwxor_ass -> EXPR EXPR ;
  bwor_ass -> EXPR EXPR ;
  lsh_ass -> EXPR EXPR ;
  rsh_ass -> EXPR EXPR ;
  rrsh_ass -> EXPR EXPR ;
-- typical java operator
  instance_of -> EXPR TYPE;

  cast -> TYPE UNARY_EXPR;

  pre_incr -> EXPR
  ; pre_decr ->
```

```
  EXPR ; bwnot ->
  EXPR ; not ->
  EXPR ; uminus ->
  EXPR ; uplus ->
  EXPR ;
-- change java.1.1 -> ARRAY_INITIALIZER for new_array
  new_array -> SIMPLE_TYPE DIMS ARRAY_INITIALIZER_OPT;
-- change java.1.1 -> anonymous and inner class
  new_class -> ENCLOSING_INSTANCE NON_WILD_ARGS NAME EXPRLIST
ANONYMOUS_BODY;
```

```
  integer -> implemented as string ;
  number -> implemented as string ;
  character -> implemented as string
  ; string -> implemented as string ;
  STRING := string ;
  true -> implemented as void
  ; false -> implemented as
  void ; this -> implemented
  as void ; null ->
  implemented as void; super -
  > implemented as void;
  parenth_expr -> EXPR ;
  call -> METHOD_ACCESS EXPRLIST
  ; index -> PRIMARY EXPR ;
  post_incr -> PRIMARY ;
  post_decr -> PRIMARY ;
  class_object -> CLASS_TYPE;
  scoped_this -> SCOPE;
  scoped_super -> SCOPE;
  field_access -> PRIMARY IDENTIFIER;

  non_wild_args -> TYPE *;
  generic_arguments -> NON_WILD_ARGS
  CALL;

  EXPRLIST := expr_list ;
  EXPR := CAST_EXPR mul div plus minus rem lsh rsh
  rrsh
           lt gt ge le eq neq bwand bwor bwxor and or ass plus_ass
minus_ass mul_ass div_ass rem_ass bwand_ass bwxor_ass
            bwor_ass lsh_ass rsh_ass rrsh_ass cond
  instance_of; EXPR_OPT := EXPR none;
  CAST_EXPR := UNARY_EXPR cast ;
  UNARY_EXPR := PRIMARY NEW not uplus uminus pre_incr pre_decr bwnot
          NAME post_incr post_decr;
  NEW := new_class new_array;
  ARRAY_INITIALIZER_OPT := array_initializers
  none; DIMS := array_dim_list none ;
  array_dim_list -> ARRAY_DIM + ;
  ARRAY_DIM := array_dim array_dcltr ;
  array_dim -> EXPR ;
  PRIMARY := integer number character STRING
          parenth_expr call index field_access class_object
          super this scoped_this scoped_super
          true false null generic_arguments;
  METHOD_ACCESS := NAME field_access super this scoped_super;
  ENCLOSING_INSTANCE := PRIMARY none;
  ANONYMOUS_BODY := FIELD_DECL_LIST none;
  NON_WILD_ARGS := non_wild_args;
  CALL := call;

end chapter ;

chapter STATEMENTS
  abstract syntax
```

```
compound -> STMTS ;
stmts -> STMT_AND_DECL
*; expr_stmt -> EXPR ;
if -> EXPR STMTS STMTS_OPT ;
while -> EXPR STMTS ;
```

```
  do -> STMTS EXPR ;
  for -> FOR_INIT EXPR_OPT FOR_UPDATE STMTS ;
  switch -> EXPR STMTS ;
  break -> IDENTIFIER_OPT;
  continue -> IDENTIFIER_OPT;
  return -> EXPR_OPT
  ; throw -> EXPR ;
  case -> EXPR ;
  default -> implemented as void
  ; label -> IDENTIFIER ;
  synchronized_stmt -> EXPR   COMPOUND;
  empty_stmt -> implemented as void;
  assert -> EXPR EXPR_OPT ;

  STMTS := stmts ;
  STMT_AND_DECL := variable_decl TYPE_DECL STMT;
  STMT := compound expr_stmt if while do for switch break
       continue return case default label empty_stmt
       synchronized_stmt throw try_block assert ;
  STMTS_OPT := STMTS none ;
  COMPOUND := compound;
  FOR_UPDATE := EXPRLIST;
  FOR_INIT := EXPRLIST variable_decl ;


end chapter ;

chapter EXCEPTION
  abstract syntax

  try_block -> COMPOUND HANDLER_LIST FINALLY ;
  try_handlers -> TRY_HANDLER * ;
  handler -> PARAMETER_LIST COMPOUND
  ;

  HANDLER_LIST := try_handlers
  ; TRY_HANDLER := handler ;
  FINALLY := COMPOUND none;

end chapter ;

chapter

ANNOTATIONS
  abstract syntax

  annotation -> IDENTIFIER PAIR_LIST
  ; pair -> IDENTIFIER_OPT VALUE ;
  pair_list -> PAIR * ;
  elt_value_array_init -> VALUE * ;

  annotation_type_decl -> ANNOT_HEAD
  ANNOT_FIELD_DECL_LIST ; annot_head -> MODIFIER_LIST
  IDENTIFIER; annot_field_decl_list -> ANNOT_FIELD_DECL
  * ; annot_field_decl -> ANNOT_FIELD_DECL ;
```

```
annotation_method -> MODIFIER_LIST TYPE IDENTIFIER
DEFAULT_VALUE ; default_value -> VALUE ;

PAIR_LIST := pair_list ;
PAIR := pair ;
VALUE := PRIMARY NAME cond annotation elt_value_array_init ;
```

```
  ANNOT_HEAD := annot_head ;
  ANNOT_FIELD_DECL_LIST :=
  annot_field_decl_list ;
  ANNOT_FIELD_DECL := variable_decl annotation_method TYPE_DECL
  ; DEFAULT_VALUE := none default_value ;

end chapter ;

chapter ENUMS
  abstract syntax

  enum_decl -> ENUM_HEAD INTERFACE_LIST ENUM_FIELD_DECL_LIST ;
  enum_head -> MODIFIER_LIST IDENTIFIER ;
  enum_field_decl_list -> ENUM_FIELD_DECL * ;
  enum_field_decl -> ENUM_CONST_LIST
  FIELD_DECL_LIST ; enum_const_list -> ENUM_CONST *
  ;
  enum_const -> MODIFIER_LIST IDENTIFIER EXPRLIST
  ANONYMOUS_BODY ;

  ENUM_HEAD := enum_head ;
  ENUM_FIELD_DECL_LIST :=
  enum_field_decl_list ; ENUM_FIELD_DECL :=
  enum_field_decl ; ENUM_CONST_LIST :=
  enum_const_list ; ENUM_CONST := enum_const
  ;

end chapter ;


chapter MISC
  abstract syntax

  scope -> IDENTIFIER *;
  identifier -> implemented as name ;
  none -> implemented as void;
  qualified_id -> SCOPE IDENTIFIER;
  generic_name -> SCOPE ;
  typed_identifier -> IDENTIFIER TYPE_ARGS_LIST;
  type_args_list -> TYPE_ARG +;
  wild_type -> implemented as void;
  extends_type -> TYPE;
  super_type -> TYPE;

  boolean -> implemented as
  void; byte -> implemented as
  void; char -> implemented as
  void; short -> implemented
  as void; int -> implemented
  as void; long -> implemented
  as void; float ->
  implemented as void; double
  -> implemented as void;
  void_type -> implemented as void;
```

```
public -> implemented as void;
protected -> implemented as
void; private -> implemented as
void; static -> implemented as
void; 'abstract' -> implemented
as void; final -> implemented
as void; native -> implemented
as void; volatile ->
implemented as void;
```

```
  synchronized -> implemented as void;
  transient -> implemented as void;
  threadsafe -> implemented as void;

  IDENTIFIER_OPT := IDENTIFIER none;
  IDENTIFIER := identifier typed_identifier;
  SCOPE := scope;
  NAME := qualified_id IDENTIFIER;
  NAME_OR_GEN_NAME := NAME generic_name;
  TYPE_ARGS_LIST := type_args_list;
  TYPE_ARG := TYPE wild_type extends_type super_type;

  NO_AUTO_ANNOT := none type_decl_list;
  COORD_ON_COMMENT := method_decl class_decl
  interface_decl ; STATIC_OPT := none static ;

end chapter ;

frames
   prefix -> implemented as tree;
       controls copy save;
   postfix -> implemented as
   tree;
       controls copy save;
   focus -> implemented as integer;
       controls copy;

end chapter;

end definition
```

Concrete syntax

```
rules definition of JAVA version 1 is

-- %start program

-- All identifiers that are not reserved words
-- %token %IDENT

-- Reserved words that specify type.
-- ie: boolean byte char short int float long double void
-- %token %TYPESPEC

-- Character or numeric constants.
-- %token %INTEGER %FLOAT %CHARACTER

-- String constants
-- %token %STRING

-- the reserved words
-- %token IF ELSE WHILE DO FOR SWITCH CASE DEFAULT
-- %token BREAK CONTINUE RETURN ASSERT
-- CLASS NEW PRIVATE PROTECTED PUBLIC FINAL NATIVE VOLATILE
-- THROW TRY CATCH USING SYNCHRONIZED ABSTRACT THREADSAFE
```

```
TRANSIENT
-- CAST ENUM

-- Used to resolve s/r with epsilon
```

```
%[LEFT %EMPTY ]%
-- Add precedence rules to solve dangling else s/r conflict
%[NONASSOC 'if' ]%
%[NONASSOC 'else' ]%
%[LEFT %IDENT ]%
%[LEFT "{", ",", ";" ]%
%[NONASSOC 'throw']%

-- Define the operator tokens and their precedences.
%[RIGHT "+=", "-=", "*=", "/=", "%=", "&=", "^=", "|=", "<<=",
">>=", ">>>=", "=" ]%
%[RIGHT "?" , ":" ]%
%[LEFT "||" ]%
%[LEFT "&&" ]%
%[LEFT "|" ]%
%[LEFT "^" ]%

%[LEFT "&" ]%
%[LEFT "==" , "!="]%
%[LEFT "<" , ">" , "<=" , ">="]%
%[LEFT "<<" , ">>", ">>>" ]%
%[LEFT "+" , "-" ]%
%[LEFT "*" , "/" , "%" ]%
%[RIGHT %UNARY , "++" , "--" ]%
%[LEFT %HYPERUNARY ]%
%[LEFT "." , "(" , "[" ]%
%[NONASSOC 'new', 'try', 'catch', 'instanceof']%


chapter PARSER

chapter TOP

rules

entry_point := comp_unit ;
     comp_unit

comp_unit := package_stmt import_stmt_list  type_decl_list ;
     :comp_unit<package_stmt,import_stmt_list,type_decl_
list> comp_unit := package_stmt type_decl_list ;
     :comp_unit<package_stmt,:import_stmt_list<>,type_decl_list>
comp_unit := import_stmt_list  type_decl_list ;
     :comp_unit<:none,import_stmt_list,type_decl_list>
comp_unit := type_decl_list ;
     :comp_unit<:none,:import_stmt_list<>,type_decl_list>

recover package_stmt ";";
package_stmt := 'package' name ";"

;
     :package_stmt<:modifier_list<>, name>
package_stmt := modifier_list 'package' name
";"
```

```
;
     :package_stmt<modifier_list, name>


recover import_stmt_list as import_stmt1;
import_stmt_list := import_stmt1;
     :import_stmt_list<import_stmt1>
import_stmt_list := import_stmt_list
                                     import_stmt1
;
```

```
     import_stmt_list:<..,import_stmt1>
recover import_stmt1 ";";
import_stmt1 := 'import' name ";";
     :import_stmt<:none, name>
import_stmt1 := 'import' gen_name ";";
     :import_stmt<:none, gen_name>
import_stmt1 := 'import' 'static' name
";";
     :import_stmt<:static, name>
import_stmt1 := 'import' 'static' gen_name ";";
     :import_stmt<:static, gen_name>


type_decl_list := ;
     :type_decl_list<>
type_decl_list :=
type_decl_list1;
     type_decl_list1
recover type_decl_list1 as type_decl1;
type_decl_list1 := type_decl1;
     :type_decl_list<type_decl1>
type_decl_list1 := type_decl_list1
                                  type_decl1
;
     type_decl_list1:<..,type_decl1>
type_decl1 := ";";
     :none
type_decl1 := class_declaration   ;
     class_declaration
type_decl1 := enum_declaration

;
     enum_declaration
type_decl1 := interface_declaration ;
     interface_declaration
type_decl1 := annotation_type_decl ;
     annotation_type_decl

end chapter ;



chapter CLASSES

rules

class_declaration := class_head type_params class_ext
class_definition ;
     :class_decl<class_head, type_params, class_ext,
     class_definition>

class_head := modifier_list 'class' identifier;
     :class_head<modifier_list,identifier>

class_head := 'class' identifier;
```

```
     :class_head<:modifier_list<>,identifier>

type_params := ;
     :none;
type_params := "<" type_param_list ">";
     type_param_list

type_params := "<" type_param_cont ">>";
     :type_param_list<type_param_cont>
type_params := "<" type_param_cont ">>>";
     :type_param_list<type_param_cont>
```

```
-- recover type_param_list as type_param;
type_param_list := type_param;
     :type_param_list<type_param>
type_param_list := type_param_list "," type_param;
     type_param_list:<..,type_param>

type_param := identifier bound;
     :type_param<identifier, bound>

type_param_cont := identifier bound_cont;
     :type_param<identifier, bound_cont>

bound := ;
     :none
bound := 'extends' bound_list;
     bound_list

bound_cont := 'extends' bound_list_cont;
     :bound_list<bound_list_cont>

bound_list := type;
     :bound_list<type>
bound_list := bound_list "&" type;
     bound_list:<..,type>

bound_list_cont := type "<" type_args_list;
     :typed_identifier<type, type_args_list>
bound_list_cont := type "<" bound_list_cont2;
     :typed_identifier<type, bound_list_cont2>

bound_list_cont2 := type_args_list_cont;
     :type_args_list<type_args_list_cont>

class_ext := extends interface_list;
     :class_ext<extends, interface_list>

class_definition := "{" "}";
     :field_decl_list<>
class_definition := "{" field_decl_list
     "}"; field_decl_list

recover field_decl_list as field_decl;
field_decl_list := field_decl;
     :field_decl_list<field_decl>
field_decl_list := field_decl_list
                                  field_decl
;
     field_decl_list:<..,field_decl>


interface_declaration := interface_head type_params
interface_ext class_definition;
     :interface_decl<interface_head, type_params,
interface_ext,class_definition>
```

```
interface_head := modifier_list "interface" identifier;
     :interface_head<modifier_list,identifier>

interface_head := "interface" identifier;
```

```
     :interface_head<:modifier_list<>,identifier>

interface_ext := ;
     :none
interface_ext := 'extends' interface_ext1;
     interface_ext1
interface_ext1 :=
name;
     :extends_list<name>
interface_ext1 := interface_ext1 "," name;
     interface_ext1:<..,name>

extends := ;
     :none
extends := 'extends' name;
     :extends<name>

interface_list := ;
     :none
interface_list := 'implements' interface_list1;
     interface_list1
interface_list1 := name ;
     :interface_list<name>
interface_list1 := interface_list1 ","
                                    name
     ; interface_list1:<..,name>

modifier_list := modifier;
     :modifier_list<modifier>
modifier_list := modifier_list
modifier;
      modifier_list:<..,modifier>

modifier := 'public';
       :public
modifier :=
'private';
       :private
modifier :=
'protected';
       :protected
modifier := 'static';
       :static
final := 'final';
     :final
modifier :=
final;
       final
modifier := 'volatile';
       :volatile
modifier := 'native';
       :native
modifier := 'synchronized';
       :synchronized
```

```
mabstract := 'abstract';
        :'abstract'
modifier :=
mabstract;
        mabstract
modifier :=
'threadsafe';
        :threadsafe
modifier := 'transient';
        :transient
modifier := annotation
;
     annotation
```

```
recover field_decl;
field_decl := ";" ;
     :none;
field_decl := method_decl     ;
     method_decl
field_decl := variable_decl ";"
     ; variable_decl
field_decl := static_init ;
     static_init
-- changes for java.1.1 : object initializers
field_decl := object_init ;
     object_init
-- changes for java.1.1 : nested classes and
interfaces field_decl := class_declaration ;
     class_declaration
field_decl := interface_declaration ;
     interface_declaration
field_decl :=
annotation_type_decl ;
     annotation_type_decl
field_decl := enum_declaration ;
     enum_declaration

method_decl := modifier_list type_params type method_dcltr
throw_list method_body ;
     :method_decl<modifier_list, type_params,
type,method_dcltr,throw_list,method_body>
method_decl := modifier_list type_params void_type
method_dcltr throw_list method_body ;
     :method_decl<modifier_list, type_params,
void_type,method_dcltr,throw_list,method_body>
method_decl := modifier_list type_params method_dcltr
throw_list method_body ;
     :method_decl<modifier_list, type_params,
:none,method_dcltr,throw_list,method_body>
method_decl := type_params type method_dcltr throw_list
               method_body ;
     :method_decl<:modifier_list<>, type_params,
type,method_dcltr,throw_list,method_body>
method_decl := type_params void_type method_dcltr
throw_list method_body ;
     :method_decl<:modifier_list<>, type_params,
void_type,method_dcltr,throw_list,method_body>
method_decl := type_params method_dcltr throw_list     method_body
;
     :method_decl<:modifier_list<>, type_params,
:none,method_dcltr,throw_list,method_b
ody> method_body := ";";
     :none
method_body := block_stmts;
     block_stmts

method_dcltr := identifier "(" parameter_list ")";
     :method_dcltr<identifier,parameter_list,:none>
```

```
-- obsolete form for methods returning arrays
method_dcltr := identifier "(" parameter_list ")" array_dcltr;
     :method_dcltr<identifier,parameter_list, array_dcltr>


static_init := 'static' block_stmts;
```

```
     :static_initializer<block_stmts>

object_init := block_stmts;
     :object_initializer<block_stmts>

throw_list := ;
     :none
throw_list := 'throws' ;
     :throw_list<>
throw_list := 'throws' throw_list1;
     throw_list1
throw_list1 := name;
     :throw_list<name>
throw_list1 := throw_list1 ","
     name; throw_list1:<..,name>


end chapter ;


chapter IDENTIFIER

rules

identifier := %IDENT ;
     :identifier[%IDENT]
identifier := identifier1
gen_list;
     :typed_identifier<identifier1, gen_list>

identifier1 := %IDENT ;
     :identifier[%IDENT]
identifier1 := identifier ;
     identifier


gen_list := "<" type_args_list ">";
     type_args_list
gen_list := "<" type_args_list_cont ">>";
     :type_args_list<type_args_list_cont>
gen_list := "<" gen_list2;
     :type_args_list<gen_list2>

gen_list2 := type gen_list3;
     :typed_identifier<type, gen_list3>

gen_list3 := "<" type_args_list_cont ">>>";
     :type_args_list<type_args_list_cont>

type_args_list := type_arg;
     :type_args_list<type_arg>
type_args_list := type_args_list "," type_arg;
     type_args_list:<..,type_arg>

type_args_list_cont := type "<" type_args_list;
```

```
    :typed_identifier<type, type_args_list>

type_arg := type;
    type
```

```
type_arg := ext_type;
     ext_type

ext_type := "?";
     :wild_type
ext_type := "?" 'extends' type;
     :extends_type<type>
ext_type := "?" 'super' type;
     :super_type<type>

name := qualified_id;
     qualified_id
qualified_id := scope ;
     %{
     /* to avoid ambiguities parse a scope
          and return a qualified_id or an identifier */
     VTP_TreeP scope = Parser_Pop();
     VTP_TreeP ident = VTP_TreeDisown(scope, -
     1); if (VTP_TreeLength(scope) > 0) {
          /* real qualified id => extract last ident */
       VTP_TreeP qualif =
       VTP_TreeMake(JAVA_1_op_qualified_id);
       VTP_TreeSetChild(qualif, scope, 0);
       VTP_TreeSetChild(qualif, ident, 1);
       Parser_SetCoordNN(qualif, scope,
       ident);
          /* reajusting the coordinates of scope */
       ident = VTP_TreeDown(scope, -1);
       Parser_SetCoordNN(scope, scope, ident);
       Parser_Push(qualif);
     } else {
          /* identifier */
       VTP_TreeDestroy(scope);
       Parser_Push(ident);
     }
     }%

scope := identifier;
     :scope<identifier>
scope := scope "." identifier;
     scope:<..,identifier>


gen_name := scope "." "*" ;
     :generic_name<scope>

string := %STRING ;
     :string[%STRING]
end chapter ;

chapter

EXPRESSSION rules
```

```
expr := %[CBLOCK_BEGIN]% expr_r %[CBLOCK_END]% ;
     expr_r
recover expr_r;
expr_r := expr1;
     expr1
```

```
expr_list := ;
     :expr_list<>
expr_list :=
expr_list1;
     expr_list1
expr_list1 := expr
;
     :expr_list<expr>
expr_list1 := expr_list1 "," expr
     ; expr_list1:<.., expr>

unary_expr := "++" unary_expr ;     %[PREC %UNARY ]%
     :pre_incr<unary_expr>
unary_expr := "--" unary_expr ;     %[PREC %UNARY ]%
     :pre_decr<unary_expr>
unary_expr := "+" unary_expr ;      %[PREC %UNARY ]%
     :uplus<unary_expr>
unary_expr := "-" unary_expr ;      %[PREC %UNARY ]%
     :uminus<unary_expr>
unary_expr :=
     unary_expr_notpm;
     unary_expr_notpm

unary_expr_notpm := "~" unary_expr ;     %[PREC %UNARY ]%
     :bwnot<unary_expr>
unary_expr_notpm := "!" unary_expr ;     %[PREC %UNARY ]%
     :not<unary_expr>
unary_expr_notpm :=
     postfix_expr;
     postfix_expr
unary_expr_notpm :=
     cast_expr; cast_expr

postfix_expr := primary;
     primary
-- name n'est pas une primary pour eviter l'ambiguite dans
field_access
postfix_expr :=
     name; name
postfix_expr := postfix_expr "++";
     :post_incr<postfix_expr>
postfix_expr := postfix_expr "--";
     :post_decr<postfix_expr>


cast_expr := "(" primitive_type_r ")" unary_expr ;%[PREC %UNARY]%
     :cast<primitive_type_r,unary_expr>
-- avoid ambiguity with (a)+b
cast_expr := "(" reference_type_r ")" unary_expr_notpm ;    %[PREC
%UNARY]%
     :cast<reference_type_r,unary_expr_notpm>
recover reference_type_r;-- to avoid ambiguities with
(expr_r) reference_type_r := reference_type ;
     reference_type
recover primitive_type_r;-- to avoid ambiguities with
```

```
(expr_r) primitive_type_r := primitive_type ;
     primitive_type

-- WARNING : constant-expression are equal to expr1,
-- They should exclude assignment-expression and throw-expression
expr1 := unary_expr
     ; unary_expr;
expr1 := expr1 'instanceof' type ; %[PREC 'instanceof']%
```

```
     :instance_of<expr1,type>
expr1 := expr1 "+" expr1 ;
     :plus<expr1.0,expr1.1>
expr1 := expr1 "-" expr1 ;
     :minus<expr1.0,expr1.1>
expr1 := expr1 "*" expr1 ;
     :mul<expr1.0,expr1.1>
expr1 := expr1 "/" expr1 ;
     :div<expr1.0,expr1.1>
expr1 := expr1 "%" expr1 ;
     :rem<expr1.0,expr1.1>
expr1 := expr1 "<<" expr1 ;
     :lsh<expr1.0,expr1.1>
expr1 := expr1 ">>" expr1
;
     :rsh<expr1.0,expr1.1>
expr1 := expr1 ">>>" expr1
;
     :rrsh<expr1.0,expr1.1>
expr1 := expr1 "<" expr2 ;
     :lt<expr1,expr2>
expr1 := expr3 ">" expr1
;
     :gt<expr3,expr1>
expr1 := expr1 "<="
expr1 ;
     :le<expr1.0,expr1.1>
expr1 := expr1 ">=" expr1
;
     :ge<expr1.0,expr1.1>
expr1 := expr1 "==" expr1
;
     :eq<expr1.0,expr1.1>
expr1 := expr1 "!=" expr1
;
     :neq<expr1.0,expr1.1>
expr1 := expr1 "&" expr1 ;
     :bwand<expr1.0,expr1.1>
expr1 := expr1 "|" expr1 ;
     :bwor<expr1.0,expr1.1>
expr1 := expr1 "^" expr1 ;
     :bwxor<expr1.0,expr1.1>
expr1 := expr1 "&&" expr1 ;
     :and<expr1.0,expr1.1>
expr1 := expr1 "||" expr1
;
     :or<expr1.0,expr1.1>
expr1 := expr1 "?" expr ":" expr1 ;
     :cond<expr1.0,expr,expr1.1>
expr1 := expr1 "=" expr1 ;
     :ass<expr1.0,expr1.1>
expr1 := expr1 "+=" expr1
;
     :plus_ass<expr1.0,expr1.1>
```

```
expr1 := expr1 "-=" expr1 ;
     :minus_ass<expr1.0,expr1.1>
expr1 := expr1 "*=" expr1 ;
     :mul_ass<expr1.0,expr1.1>
expr1 := expr1 "/=" expr1 ;
     :div_ass<expr1.0,expr1.1>
expr1 := expr1 "%=" expr1 ;
     :rem_ass<expr1.0,expr1.1>
expr1 := expr1 "&=" expr1 ;
     :bwand_ass<expr1.0,expr1.1>
expr1 := expr1 "^=" expr1 ;
     :bwxor_ass<expr1.0,expr1.1>
expr1 := expr1 "|=" expr1 ;
```

```
      :bwor_ass<expr1.0,expr1.1>
expr1 := expr1 "<<=" expr1 ;
      :lsh_ass<expr1.0,expr1.1>
expr1 := expr1 ">>=" expr1 ;
      :rsh_ass<expr1.0,expr1.1>
expr1 := expr1 ">>>=" expr1 ;
      :rrsh_ass<expr1.0,expr1.1>

expr2 := unary_expr ;
      unary_expr;
expr2 := expr2 'instanceof' type ; %[PREC 'instanceof']%
      :instance_of<expr2,type>
expr2 := expr2 "+" expr2 ;
      :plus<expr2.0,expr2.1>
expr2 := expr2 "-" expr2 ;
      :minus<expr2.0,expr2.1>
expr2 := expr2 "*" expr2 ;
      :mul<expr2.0,expr2.1>
expr2 := expr2 "/" expr2 ;
      :div<expr2.0,expr2.1>
expr2 := expr2 "%" expr2 ;
      :rem<expr2.0,expr2.1>
expr2 := expr2 "<<" expr2
;
      :lsh<expr2.0,expr2.1>
expr2 := expr2 ">>" expr2 ;
      :rsh<expr2.0,expr2.1>
expr2 := expr2 ">>>" expr2 ;
      :rrsh<expr2.0,expr2.1>
expr2 := expr2 "<" expr2 ;
      :lt<expr2.0,expr2.1>
expr2 := expr2 "<=" expr2 ;
      :le<expr2.0,expr2.1>
expr2 := expr2 ">=" expr2
;
      :ge<expr2.0,expr2.1>
expr2 := expr2 "==" expr2
;
      :eq<expr2.0,expr2.1>
expr2 := expr2 "!=" expr2 ;
      :neq<expr2.0,expr2.1>
expr2 := expr2 "&" expr2 ;
      :bwand<expr2.0,expr2.1>
expr2 := expr2 "|" expr2 ;
      :bwor<expr2.0,expr2.1>
expr2 := expr2 "^" expr2 ;
      :bwxor<expr2.0,expr2.1>
expr2 := expr2 "&&" expr2 ;
      :and<expr2.0,expr2.1>
expr2 := expr2 "||" expr2 ;
      :or<expr2.0,expr2.1>
expr2 := expr2 "?" expr ":" expr2 ;
      :cond<expr2.0,expr,expr2.1>
expr2 := expr2 "=" expr2 ;
      :ass<expr2.0,expr2.1>
```

```
expr2 := expr2 "+=" expr2
;
     :plus_ass<expr2.0,expr2.1>
expr2 := expr2 "-=" expr2 ;
     :minus_ass<expr2.0,expr2.1>
expr2 := expr2 "*=" expr2 ;
     :mul_ass<expr2.0,expr2.1>
```

```
expr2 := expr2 "/=" expr2 ;
     :div_ass<expr2.0,expr2.1>
expr2 := expr2 "%=" expr2 ;
     :rem_ass<expr2.0,expr2.1>
expr2 := expr2 "&=" expr2 ;
     :bwand_ass<expr2.0,expr2.1>
expr2 := expr2 "^=" expr2 ;
     :bwxor_ass<expr2.0,expr2.1>
expr2 := expr2 "|=" expr2 ;
     :bwor_ass<expr2.0,expr2.1>
expr2 := expr2 "<<=" expr2 ;
     :lsh_ass<expr2.0,expr2.1>
expr2 := expr2 ">>=" expr2 ;
     :rsh_ass<expr2.0,expr2.1>
expr2 := expr2 ">>>=" expr2 ;
     :rrsh_ass<expr2.0,expr2.1>

expr3 := unary_expr ;
     unary_expr;
expr3 := expr3 'instanceof' type ; %[PREC 'instanceof']%
     :instance_of<expr3,type>
expr3 := expr3 "+" expr3 ;
     :plus<expr3.0,expr3.1>
expr3 := expr3 "-" expr3 ;
     :minus<expr3.0,expr3.1>
expr3 := expr3 "*" expr3 ;
     :mul<expr3.0,expr3.1>
expr3 := expr3 "/" expr3 ;
     :div<expr3.0,expr3.1>
expr3 := expr3 "%" expr3 ;
     :rem<expr3.0,expr3.1>
expr3 := expr3 "<<" expr3
;
     :lsh<expr3.0,expr3.1>
expr3 := expr3 ">>" expr3
;
     :rsh<expr3.0,expr3.1>
expr3 := expr3 ">>>" expr3
;
     :rrsh<expr3.0,expr3.1>
expr3 := expr3 ">" expr3 ;
     :gt<expr3.0,expr3.1>
expr3 := expr3 "<=" expr3
;
     :le<expr3.0,expr3.1>
expr3 := expr3 ">=" expr3
;
     :ge<expr3.0,expr3.1>
expr3 := expr3 "==" expr3
;
     :eq<expr3.0,expr3.1>
expr3 := expr3 "!=" expr3
;
     :neq<expr3.0,expr3.1>
```

```
expr3 := expr3 "&" expr3 ;
     :bwand<expr3.0,expr3.1>
expr3 := expr3 "|" expr3 ;
     :bwor<expr3.0,expr3.1>
expr3 := expr3 "^" expr3 ;
     :bwxor<expr3.0,expr3.1>
expr3 := expr3 "&&" expr3 ;
     :and<expr3.0,expr3.1>
expr3 := expr3 "||" expr3 ;
     :or<expr3.0,expr3.1>
expr3 := expr3 "?" expr ":" expr3 ;
```

```
     :cond<expr3.0,expr,expr3.1>
expr3 := expr3 "=" expr3 ;
     :ass<expr3.0,expr3.1>
expr3 := expr3 "+=" expr3 ;
     :plus_ass<expr3.0,expr3.1>
expr3 := expr3 "-=" expr3 ;
     :minus_ass<expr3.0,expr3.1>
expr3 := expr3 "*=" expr3 ;
     :mul_ass<expr3.0,expr3.1>
expr3 := expr3 "/=" expr3 ;
     :div_ass<expr3.0,expr3.1>
expr3 := expr3 "%=" expr3 ;
     :rem_ass<expr3.0,expr3.1>
expr3 := expr3 "&=" expr3 ;
     :bwand_ass<expr3.0,expr3.1>
expr3 := expr3 "^=" expr3 ;
     :bwxor_ass<expr3.0,expr3.1>
expr3 := expr3 "|=" expr3 ;
     :bwor_ass<expr3.0,expr3.1>
expr3 := expr3 "<<=" expr3 ;
     :lsh_ass<expr3.0,expr3.1>
expr3 := expr3 ">>=" expr3 ;
     :rsh_ass<expr3.0,expr3.1>
expr3 := expr3 ">>>=" expr3 ;
     :rrsh_ass<expr3.0,expr3.1>


primary := primary_no_new_array ;
     primary_no_new_array
primary := 'new' new_array;   %[PREC %EMPTY ]%
     new_array

primary_no_new_array := literal;
     literal
primary_no_new_array:= "(" expr_r ")" ;
     :parenth_expr<expr_r>
primary_no_new_array := 'new' name "(" expr_list ")" anonymous_body
;%[PREC %EMPTY ]%
     :new_class<:none, :none, name, expr_list, anonymous_body>
primary_no_new_array := enclosing_instance 'new' identifier "("
expr_list ")" anonymous_body ;%[PREC %EMPTY ]%
     :new_class<enclosing_instance, :none, identifier, expr_list,
anonymous_body>
primary_no_new_array := 'new' non_wild_args name "(" expr_list ")"
anonymous_body ;%[PREC %EMPTY ]%
     :new_class<:none, non_wild_args, name, expr_list,
     anonymous_body>
primary_no_new_array := enclosing_instance 'new' non_wild_args
identifier "(" expr_list ")" anonymous_body ;%[PREC %EMPTY ]%
     :new_class<enclosing_instance, non_wild_args, identifier,
expr_list, anonymous_body>
primary_no_new_array := field_access;
     field_access
primary_no_new_array := method_invocation;
     method_invocation
```

```
primary_no_new_array := array_access;
     array_access
-- change for java.1.1. to get an object from a names class
primary_no_new_array := type "." 'class';
```

```
     :class_object<type>
primary_no_new_array := void_type "." 'class';
     :class_object<void_type>
-- change for java.1.1. to get enclosing instance of inner class
instance primary_no_new_array := scoped_this;
     scoped_this
primary_no_new_array := generic_arguments;
     generic_arguments;

--java.1.1 change -> inner class
enclosing_instance := primary "." ;
     primary
enclosing_instance := name "."
     ; name
--java.1.1 change -> anonymous class
anonymous_body := ;
     :none
anonymous_body := class_definition ;
     class_definition

new_array := primitive_type dims;
     :new_array<primitive_type,dims,:none>
new_array := name dims;
     :new_array<name,dims,:none>
-- change java.1.1 -> initialization allowed with
'new' new_array := primitive_type dims
array_initializers;
     :new_array<primitive_type,dims,array_initializ
ers> new_array :=  name dims array_initializers;
     :new_array<name,dims,array_initializers>
dims := array_dim ;
     :array_dim_list<array_dim>
dims := dims array_dim ;
     dims:<.., array_dim>
array_dim := "[" expr "]" ;
     :array_dim<expr>
array_dim := "[" "]" ;
     :array_bracket


-- change for java.1.1 -> inner classes
scoped_this := scope "." 'this' ;
     :scoped_this<scope>
scoped_super := scope "." 'super' ;
     :scoped_super<scope>

literal := %INTEGER ;
     :integer[%INTEGER]
literal := %FLOAT ;
     :number[%FLOAT]
literal := %CHARACTER ;
     :character[%CHARACTER]
literal := string ;
     string
   literal:=
```

```
  super; super
literal:= this;
     this
literal := 'true' ;
```

```
     :true
literal:= 'false'
;
     :false
literal :=
'null';
     :null
super := 'super' ;
     :super
this:= 'this';
     :this

method_invocation := name "(" expr_list ")" ;     %[PREC "."]%
     :call<name,expr_list>
method_invocation := field_access "(" expr_list ")" ;  %[PREC
"."]%
     :call<field_access,expr_list>
-- for constructors
method_invocation := this "(" expr_list ")" ;
     :call<this,expr_list>
method_invocation := super "(" expr_list ")" ;
     :call<super,expr_list>
method_invocation := scoped_super "(" expr_list ")" ;
     :call<scoped_super,expr_list>

field_access := primary "." identifier;
     :field_access<primary,identifier>
field_access := super "." identifier;
     :field_access<super,identifier>
field_access := scoped_super "." identifier;
     :field_access<scoped_super,identifier>


array_access := name "[" expr "]";
     :index<name,expr>
array_access := primary_no_new_array "[" expr "]";
     :index<primary_no_new_array,expr>

generic_arguments := non_wild_args method_invocation;
     :generic_arguments<non_wild_args, method_invocation>

non_wild_args := "<" type_list ">";
     type_list

type_list := type;
     :non_wild_args<type>
type_list := type_list ","
type;
     type_list:<..,typ

e> end chapter ;

chapter DECLARATIONS

rules
```

```
-- recover variable_decl;
variable_decl := modifier_list type var_dcltr_list     ;
     :variable_decl<modifier_list,type,var_dcltr_list>
variable_decl := type var_dcltr_list    ;
     :variable_decl<:modifier_list<>,type,var_dcltr_list>
```

```
local_variable_decl := %[CBLOCK_BEGIN]% local_variable_decl1
%[CBLOCK_END]% ;
     local_variable_decl1
recover local_variable_decl1;
local_variable_decl1 :=type var_dcltr_list ;
     :variable_decl<:modifier_list<>,type,var_dcltr_list>
-- change java.1.1
local_variable_decl1 :=local_modifiers type var_dcltr_list;
     :variable_decl<local_modifiers,type,var_dcltr_list>
local_modifiers := final;
     :modifier_list<final>

var_dcltr_list1 := var_declarator;
     :var_dcltr_list<var_declarator>
var_dcltr_list1 := var_dcltr_list1 "," var_declarator;
     var_dcltr_list1:<..,var_declarator>
var_dcltr_list := var_dcltr_list1;
     var_dcltr_list1

var_declarator :=
     var_dcltr_affinit;
     var_dcltr_affinit;
var_declarator := var_dcltr_noinit;
     var_dcltr_noinit;
var_declarator := var_dcltr_foreach;
     var_dcltr_foreach;


recover initializer ;
var_dcltr_affinit := identifier array_dcltr_opt "="
initializer;
     :var_dcltr_affinit<identifier,array_dcltr_opt,
     initializer>

var_dcltr_noinit := identifier array_dcltr_opt ;
     :var_dcltr_noinit<identifier, array_dcltr_opt>

var_dcltr_foreach := identifier array_dcltr_opt ":" initializer
;
     :var_dcltr_foreach<identifier, array_dcltr_opt,
     initializer>

initializer := expr;
     expr
initializer :=
               array_initializer
     s; array_initializers


array_initializers := "{" initializer_list "}";
     :array_initializers<initializer_list>
array_initializers := "{" initializer_list "," "}";
     :array_initializers<initializer_list>

--recover initializer_list;
```

```
initializer_list := ;
     :initializer_list<>
initializer_list := initializer;
     :initializer_list<initializer>
initializer_list := initializer_list ","
     initializer; initializer_list:<..,initializer>
--ajout java.1.1 local inner class
local_class_or_interf_declaration := %[CBLOCK_BEGIN]%
local_class_or_interf_declaration1 %[CBLOCK_END]% ;
```

```
     local_class_or_interf_declaration1
recover local_class_or_interf_declaration1
"}";

local_class_or_interf_declaration1 := local_class_head
type_params class_ext class_definition ;
     :class_decl<local_class_head, type_params, class_ext,
class_definition>
local_class_or_interf_declaration1 := local_interf_head
type_params interface_ext class_definition ;
     :interface_decl<local_interf_head, type_params,
interface_ext, class_definition>

local_class_head := local_class_modifiers 'class' identifier;
     :class_head<local_class_modifiers,
identifier> local_class_head :=    'class'
identifier;
     :class_head<:none,
identifier>
local_class_modifiers := final;
     :modifier_list<final>
local_class_modifiers :=
                         mabstract
;
     :modifier_list<mabstract>

local_interf_head := local_interf_modifiers "interface"
identifier;
     :interface_head<local_interf_modifiers,identifier>
local_interf_head := "interface" identifier;
     :interface_head<:none,identifi
er> local_interf_modifiers := final;
     :modifier_list<final>
local_interf_modifiers :=
                          mabstract
;
     :modifier_list<mabstract>


primitive_type := 'boolean';
        :boolean
primitive_type :=
'byte';
        :byte
primitive_type :=
'char';
        :char
primitive_type :=
'short';
        :short
primitive_type :=
'int';
        :int
primitive_type :=
```

```
'float';
        :float
primitive_type :=
'long';
        :long
primitive_type :=
'double';
        :double
void_type := 'void';
        :void_type

type := primitive_type;
     primitive_type
type := reference_type;
     reference_type
reference_type := name;
     name
reference_type := array_type;
     array_type
```

```
array_type := primitive_type array_dcltr;
     :array_type<primitive_type, array_dcltr>
array_type := name array_dcltr;
     :array_type<name, array_dcltr>
array_dcltr := array_bracket;
     :array_dcltr<array_bracket>
array_dcltr := array_dcltr
array_bracket;
     :array_dcltr<array_dcltr, array_bracket>
array_dcltr_opt := ;
     :none
array_dcltr_opt := array_dcltr;
     array_dcltr
array_bracket := "[" "]";
     :array_bracket

end chapter ;


chapter STATEMENTS

rules

recover stmts1 as
stmt1; stmts :=
          stmts1  ;
     stmts1
stmts1 :=
stmt1;
     :stmts<stmt1>
stmts1 := stmts1
stmt1;
     stmts1:<..,stmt1>

recover stmt1;
stmt1 := local_variable_decl
                           ";"
     ; local_variable_decl
-- add for java.1.1
stmt1 := local_class_or_interf_declaration;
     local_class_or_interf_declaration
stmt1 := stmt  ;
     stmt
stmt1 := label1
         ; label1

block_stmts := "{" block_stmts1 "}";
     :compound<
block_stmts1> recover
block_stmts1; block_stmts1
:= ;
     :stmts<>
block_stmts1 := stmts;
     stmts
```

```
labeled_stmt := stmt;
     :stmts<stmt>
labeled_stmt := label1 labeled_stmt;
     labeled_stmt:<label1,..>

-- Parse a single real statement, not including any
labels. stmt := block_stmts;
     block_stmts
stmt := expr ";"
;
```

```
     :expr_stmt<expr>
stmt := 'if' "(" expr ")" labeled_stmt 'else' labeled_stmt;
     :if<expr,labeled_stmt.0,labeled_stmt.1>
stmt := 'if' "(" expr ")" labeled_stmt;            %[PREC 'if']%
     :if<expr,labeled_stmt,:none>
stmt := 'while' "(" expr ")" labeled_stmt;
     :while<expr,labeled_stmt>
stmt := 'do' labeled_stmt 'while' "(" expr ")" ";" ;
     :do<labeled_stmt,expr>
stmt := 'for' "(" for_init_stmt ";" xexpr ";" expr_list ")"
labeled_stmt ;
     :for<for_init_stmt,xexpr,expr_list,labeled_s
tmt> stmt := 'for' "(" for_init_stmt ")"
                                    labeled_stmt
;
     :for<for_init_stmt,:none, :none,
labeled_stmt> stmt := 'switch' "(" expr ")"
                              labeled_stmt ;
     :switch<expr,labeled_stmt>
stmt := 'synchronized' "(" expr ")" block_stmts;
     :synchronized_stmt<expr,block_stmts>
stmt := 'throw' expr ";";
     :throw<expr>
stmt := 'break' identifier ";";
     :break<identifier>
stmt := 'break' ";" ;
     :break<:none>
stmt := 'continue' identifier ";" ;
     :continue<identifier>
stmt := 'continue' ";" ;
     :continue<:none>
stmt := 'return' ";" ;
     :return<:none>
stmt := 'return' expr ";" ;
     :return<expr
> stmt :=
try_block ;
     try_block
stmt := 'assert' expr ";" ;
     :assert<expr, :none>
stmt := 'assert' expr ":" expr ";" ;
     :assert<expr.0, expr.1>
stmt := ";" ;
     :empty_stmt

-- Any kind of label, including jump labels and case labels.
-- JAVA accepts labels only before statements, but we allow
them
-- also at the end of a compound statement.
label1 := %[CBLOCK_BEGIN]% label
%[CBLOCK_END]%;
     label

recover label; -- to avoid ambiguities with expr
and local_variable_decl
```

```
label := 'case' expr ":" ;
     :case<expr>
label := "default" ":" ;
     :default
label :=identifier ":" ;
     :label<identifier>


xexpr := ;
```

```
     :none
xexpr := expr
;
     expr

for_init_stmt := expr_list

     ; expr_list
for_init_stmt := local_variable_decl
     ; local_variable_decl


end chapter ;

chapter PARAMETERS

rules

-- This is what appears inside the parents in a method
declarator.
parameter_list := %[CBLOCK_BEGIN]% parameter_list_1
     %[CBLOCK_END]% ; parameter_list_1

-- This is what appears inside the parents in a method
declarator. recover parameter_list_1 as parm;
parameter_list_1 := ;
     :parameter_list<>
parameter_list_1 := parm ;
     :parameter_list<parm>
parameter_list_1 := parameter_list_1 "," parm;
     parameter_list_1:<..,parm>

-- A single parameter declaration or parameter type name,
-- as found in a parameter_list.
-- change for java.1.1 : modifiers for parameters
declaration parm :=type var_dcltr_noinit;
     :param_decl<:modifier_list<>,type,
var_dcltr_noinit> parm :=    param_modifiers type
var_dcltr_noinit;
     :param_decl<param_modifiers ,type, var_dcltr_noinit>
parm := param_modifiers type "..." var_dcltr_noinit;
     :var_arg_decl<param_modifiers ,type,
var_dcltr_noinit> parm :=    type "..."
var_dcltr_noinit;
     :var_arg_decl<:modifier_list<> ,type, var_dcltr_noinit>
param_modifiers := local_modifiers;
     local_modifiers

end chapter;



chapter EXCEPTION
rules
```

```
try_block := 'try' block_stmts handler_list finally ;
     :try_block<block_stmts, handler_list, finally>

handler_list := ;
     :try_handlers<>
handler_list := handler_list catch_handler ;
     handler_list:<.., catch_handler>
```

```
catch_handler := 'catch' "(" parameter_list ")" block_stmts;
        :handler<parameter_list,block_stmts>
finally := ;
        :none
finally := 'finally' block_stmts;
        block_stmts


end chapter;

chapter

ANNOTATIONS
rules

annotation := "@" identifier ;
     :annotation<identifier, :none>
annotation := "@" identifier pairs ;
     :annotation<identifier, pairs>

pairs := "(" pair_list ")" ;
     pair_list

pair_list := ;
     :pair_list<>
pair_list := pair ;
     :pair_list<pair>
pair_list := pair_list "," pair
     ; pair_list:<.., pair>

pair := value ;
     :pair<:none, value>
pair := identifier "=" value ;
     :pair<identifier, value>

value := primary ;
     primary
value := name ;
     name
value := cond
;
     cond
value := annotation ;
     annotation
value := elt_value_array_list;
     elt_value_array_list
value := "{" elt_value_array_list "}" ;
     elt_value_array_list

cond := expr1 "?" expr ":" expr1 ;
     :cond<expr1.0,expr,expr1.1>

elt_value_array_list := value2 ;
     :elt_value_array_init<value2>
elt_value_array_list := elt_value_array_list "," value2 ;
```

```
     elt_value_array_list:<.., value2>

value2 := primary ;
     primary
value2 := name ;
     name
```

```
value2 := cond ;
     cond
value2 := annotation ;
     annotation

annotation_type_decl := annot_head annot_field_decls ;
     :annotation_type_decl<annot_head, annot_field_decls>

annot_head := modifier_list "@" "interface" identifier ;
     :annot_head<modifier_list, identifier>
annot_head := "@" "interface" identifier ;
     :annot_head<:modifier_list<>, identifier>


annot_field_decls := "{" "}";
     :annot_field_decl_list<>
annot_field_decls := "{" annot_field_decl_list "}";
     annot_field_decl_list

-- recover annot_field_decl_list as annot_field_decl;
annot_field_decl_list := annot_field_decl ;
     :annot_field_decl_list<annot_field_decl>
annot_field_decl_list := annot_field_decl_list
annot_field_decl ;
     annot_field_decl_list:<.., annot_field_decl> ;

annot_field_decl := variable_decl ";" ;
     :annot_field_decl<variable_decl>
annot_field_decl := annotation_method
;
     :annot_field_decl<annotation_method>
annot_field_decl := type_decl1 ;
     :annot_field_decl<type_decl1>

annotation_method := modifier_list type identifier "(" ")"
default_value;
     :annotation_method<modifier_list, type, identifier,
default_value> annotation_method := type identifier "(" ")"
default_value;
     :annotation_method<:modifier_list<>, type, identifier,
default_value>

default_value := ;
     :default_value<:none>
default_value := "default"
value ;
     :default_value<value>;

end chapter;


chapter ENUMS
rules

enum_declaration := enum_head interface_list enum_definition ;
```

```
     :enum_decl<enum_head, interface_list, enum_definition>

enum_head := modifier_list 'enum' identifier ;
     :enum_head<modifier_list,identifier>
enum_head := 'enum' identifier ;
     :enum_head<:modifier_list<>,identifier>

enum_definition := "{" "}" ;
```

```
     :enum_field_decl_list<>
enum_definition := "{" enum_field_decl_list "}" ;
     enum_field_decl_list

enum_field_decl_list := enum_field_decl ;
     :enum_field_decl_list<enum_field_decl>
enum_field_decl_list := enum_field_decl_list
enum_field_decl ;
     enum_field_decl_list:<.., enum_field_decl>

enum_field_decl := enum_const_list;
     :enum_field_decl<enum_const_list, :none>
enum_field_decl := enum_const_list ";" ;
     :enum_field_decl<enum_const_list, :none>
enum_field_decl := enum_const_list field_decl_list;
     :enum_field_decl<enum_const_list,
     field_decl_list>

enum_const_list := enum_const ;
     :enum_const_list<enum_const>
enum_const_list := enum_const_list "," enum_const ;
     enum_const_list:<.., enum_const>

enum_const := modifier_list identifier "(" expr_list ")"
anonymous_body ;
     :enum_const<modifier_list, identifier, expr_list,
anonymous_body> enum_const := modifier_list identifier
anonymous_body ;
     :enum_const<modifier_list, identifier, :expr_list<>,
anonymous_body>
enum_const := modifier_list identifier "(" expr_list ")";
     :enum_const<modifier_list, identifier, expr_list, :none>
enum_const := modifier_list identifier;
     :enum_const<modifier_list, identifier, :expr_list<>, :none>

enum_const := identifier "(" expr_list ")" anonymous_body ;
     :enum_const<:modifier_list<>, identifier, expr_list,
anonymous_body>
enum_const := identifier "(" expr_list ")" ;
     :enum_const<:modifier_list<>, identifier, expr_list, :none>
enum_const := identifier anonymous_body ;
     :enum_const<:modifier_list<>, identifier, :expr_list<>,
anonymous_body>
enum_const := identifier ;
     :enum_const<:modifier_list<>, identifier, :expr_list<>,
```

```
:none> end chapter ;


end chapter ;
end definition
```

Notices

The licensed program described in this document and all licensed material available for it are provided by Kalimetrix under terms of the Kalimetrix Customer Agreement, Kalimetrix International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-Kalimetrix products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Kalimetrix has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Kalimetrix products. Questions on the capabilities of non-Kalimetrix products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Copyright license

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Kalimetrix, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Kalimetrix, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from Kalimetrix Corp. Sample Programs. © Copyright Kalimetrix Corp. _enter the year or years_.

Trademarks

Kalimetrix, the Kalimetrix logo, Kalimetrix.com are trademarks or registered trademarks of Kalimetrix, registered in many jurisdictions worldwide. Other product and services names might be trademarks of Kalimetrix or other companies.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Macrovision and FLEXnet are registered trademarks or trademarks of Macrovision Corporation.

Microsoft, Windows, Windows 2003, Windows XP, Windows Vista and/or other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corporation.

Netscape and Netscape Enterprise Server are registered trademarks of Netscape Communications Corporation in the United States and other countries.

Sun, Sun Microsystems, Solaris, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Pentium is a trademark of Intel Corporation.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be trademarks or service marks of others.