aML – a Macro Language

revised formal definition

Zsolt G. Hernáth

Technical Report
DOI:10.21862/2021.aML
Faculty of Informatics
Eötvös Loránd University
Budapest

January 13, 2021

About aML

aML is a context sensitive language in which macro names and invocations of macros together with their embedding context are freely electable so far as they can be characterized by regular languages. Due to the above features, aML is not simply for extending programming languages, but much rather for singling particular phrases of formerly issued text and make them become macro invocations of subsequently implemented macro definitions. The basis of aML is FORMAL [5, 6][1] defined in Eötvös Loránd University Computer Centre in the middle of 80s. FORMAL in many aspects broke away the philosophy of early time macro languages and the expansion methods applied with them.

The first formal definition of aML [1, 2], – from now on referred to as prototype definition has got ready in May 2010 for the STDL schema transformation language defined and implemeted in SZOMIN08 poroject (2009-2011). In the prototype definition the above mentioned regular languages were proposed to be given a conveniently modified minimal subset of POSIX regular expression language[2]. To the minimal subset selected we did not take over the $ end-of-line meta-symbol, but, at the same time, there was a particular need of the character representations of keys **ENTER** and **TAB** which are not supported by POSIX. For them, therefore, we took over escapes **\n** and **\t** from the C programming language. Another deviation from POSIX was that matching patterns **a.*b** and **a.+b** aML needs lazy quantifications instead of greedy[3]. Alterations were accomplished by a C++ implementation of macro processor modules completed by the end of 2010.

A subset of the prototype definition following out STDL schema transformations was implemented also in C. The C implementation disapproved the regular expression language worked out for the prototype definition. Instead, it followed out a minimal language with unified syntax and pattern matching concept of what is called *semi-greedy* quantification.

[1] FORMAL has originally been defined as a macro language for FORTRAN IV in 1980. It aimed at introducing new data types, control flow constructions unavailable in FORTRAN IV, and also C-like operations ++ and −−.

[2] The POSIX based regular expression language for aML has been worked out by Peter Bauer. For lack of formal specification both the POSIX subset and the deviations from POSIX were only outlined by definitions [1, 2].

[3] In lazy quantification pattern **a.*b** matches all strings started with letter a and ending with the closest b fllowing a, opposit to greedy quantification, when the farthermost letter b ensuing letter a is matched.

Deficiencies and Contradictions in Prototype Definition

By tests made with C implementation light was thrown on a number of deficiencies e.g. in load and target directives, in macro parameter references and link-indicator and also on contradictions between name classes and the syntax of macro-directive. All those and also the C implementation of the aML regular expression language brought us to the decesion of putting down a revised formal definition of aML.

Role and Treatment of Backslash

Apart from escapes **\n** and **\t** backslash plays a joint and several escaping role in the prtotype definition. Outputting any escape but **\n** and **\t** the escaped character without the preceding backslash appears purely on the all-time target segment[4]. Escapes **\n** and **\t** on output appears as ENTER and TAB (ASCII-7 octal codes 012 and 011), respectively.

Such a treatment of backslash brings on trouble in all host languages where backslash plays particular role, e.g. when a host language is a markup language, or the C programming language because of C escapes. In order to aML could be used in as wide range of host languages as possible, aML had to be furnished with a dynamic escape treatment. The dynamic backslash handling is followed out by introducing a new directive – the escape-directive.

By escape-directive one can specify the set of characters which are outputted as escape sequences (i.e characters preceded by backslash), and also those characters which preceded by backslash appear on output according to their escape interpretation specified. For example, the

```
#escape{[A(a)-Z(z)r\\]}
```

specifies that each capitals preceded by backslash as smaller case letters, and \r and \\ without alteration as \r and \\ have to be outputted.

Introducing the escape-directive brought us to dissolve hidden inconsistencies coming from the inadequate syntax of macro-, load-, and target-directive proposed by the prototype definition.

Recognition of Macro Invocations

Developing the prototype definition we thought that the efficiency of regognizing macro invocations can be increased by specifying those representatitve contexts in which macro invocations are worth to be looked for. The prototype definition, therefore provides for

[4] A text file for output in the operating system's file system assigned by the latest target-directive.

the embed-directive. STDL transformation tests showed that the embed-directive did not fulfill the expectations hoped.

Efficiency tests made by the C implementation manifested that the best efficiency can be achieved by matching arbitrary character sequences closed by elements of name classes. Such text fragments as regular languages can, however, be easily and automatically generated for each declared name class, namely, the embed-directive subvention is unnecessary.

File Names in Operating Sytem Environment

The actual input and output of the aML macro processor are determined by the recently performed load-directive and target-directive, respectively. Any of them specifies a name of a file in the operating system environment. According to the prototype definition syntax, names of files are delimited by whitespaces. Since most of nowadays used operating systems allows whitespaces inside file names, the syntax of load and target directives was desired to be changed. Syntax changed allowes leading, trailing and intermediate whitespaces building file nemes. Composing file names, escapes are also allowed. Handling escapes used happens according to the escape-directive recently applied.

Name Classes and Macro Names

The prototype definition syntax of macro-directives brings forth a hidden inconsistency between declarable name classes[5] and macro names delimited by whitespaces: e.g. no macro named **ENTER** can be defined according to the prototype definition syntax, though name class with the (only) name **ENTER** can be declared as `#embedded{"\n"}`. Assuming no escape-directive changed the treatment of escapes **n** and **t**, the below macro definition

```
#macro [\n]
{
        #invoked as "^[ \t]*&"
        {
        }
}
```

is to eliminate empty lines of a text file.

The aML regular expression language

The development of a regular expression language based on the extended POSIX but significantly different from the extended POSIX syntax was primarily due to the symbol

[5]Name classes are regular languages. Elements of the set union of name classes can be used as macro names.

representations **n** and **t** and the need of aML macro parameter declaration and reference expressions.

Since C escape sequences **n** and **t** are character representations, there is a need to use them as elements of symbol classes (i.e inside POSIX bracket expressions) which violate POSIX regular expression syntax. As two other extension of symbol classes, expressions [ ] and [^] represent the empty set[6] and its counterpart i.e any character of the whole terminal alphabet, respectively. The latter is a synonym for the . POSIX meta-symbol.

As for macro parameter expressions, both declarations and references are headed and also terminated by the symbol **$**. A declaration does furthermore describe a codomain for the declared parameter.

The regular expression language developed for aML follows a precept of matching called *semi-greedy*. The appellation semi-greedy refers to the degree of avidity of quantifiers + and ∗. When concatenated quantified patterns in a sub-expression are disjoint, quantifiers are greedy, otherwise lazy or guarded greedy. Guarded greedy quantifiers are at most as much greedy only that the rest of expression could still be matchable, provided the entire expression can be matched at all.

Connection to Operating System

Planning the prototype definition of aML practical claims of XML schemata transformations were primarily in focus, and theoretical aspects were not considered at all. Among others we do not think about a direct connection from the macro processor to the all-time operating system, though STDL[7] transformation steps create temporary files which in lack of connection to all-time operating system can not be removed by themselves. By the newly introduced system-directive an operating system command can be carried out.

Macro Parameter References, Context and Link Indicators

A macro definition in aML is a (possible empty) sequence of context-sensitive rewriting rules. On the left-hand side of rewriting rules not only scalar (ordinary, unstructured) but also structured parameters can be declared. There are two sorts of structured parameters: record and vector. They are relations in relational algebraic sense and can be nested into each other in arbitrary depth.

Records are tuple of separated fields, which depending on their declarations may carry either scalar or structured valeues. A record parameter can be referred to the entire record as if were scalar parameter, or by fields. The latter corresponds to a relational algebraic projection.

[6] The empty set might occasionaly represent the empty symbol.

[7] Schema Transformation Description Language

Vectors are sets of binary relations between the set of their elements and the set of indexes of their elements. Vectors of vectors are called multi-dimensional vectors. Multi-dimensional vectors are not arrays. Elements of the innermost vector are scalars or records, while elements of higher dimensions are vectors containing occasionally different number of elements. Similarly to record parameters, a vector parameter can be referred to as it were scalar, or as to vector, where row or column continuous ordering of elements for each dimension has to be prescribed. A reference to a vector parameter as vector results in the series of values of all elements of the vector in a particular order. In case of multi-dimensional vectors the order of elements in the series depends on in which dimension which ordering of elements is specified. A column continuous ordering in any dimension distributes the series of element values over more mere lines.

The prototype definition does not clarify the influence of the context indicator when vector parameter references result in more than one line, and how vectors of different dimension are linked by the link-indicator.

The influence of the context indicator on vector parameter references, of course, must not depend on whether values are distributed over one or more single lines, ensured by the statement preservation principle which states:

one embedding statement read in remains one statement after its parameter references have been substituted by their actual values, independently of the value substistutions result in one or more single lines.

The link-indicator performs a relational algebraic projection to vector elements in the outer join of vectors of occasionally different dimension, and different number of elements in different dimensions. Scalars linked are considered as zero-dimensional vectors, but the outer join is controlled by at least 1-dimensional vector from left to right. The join condition is a partial equality between element indexes. Element indexes are ordered tuples. A partial equality between element indexes is hold, if walking down from the highest towards the lowest dimension the indexes are equal to each other. The appearance and the sequence of linked vector elements depend on whether row or column continuous order is specified for each dimension.

Prototype Definition Extensions

Associate Grammars

aML in use lives in symbiosis with two other languages. One of them is the host language, and the other one is a regular expression language. Though aML strongly depends on its all-time host language, and its regular expression language, it does not consider its own grammar either as a sub-grammar of that of the host language, or as the main-grammar of its regular expression language grammar

This paper describes all three garammars separately. Dependeces between separately defined grammars are denoted by applying **quasi-terminal** symbols. Any non-terminal symbol of a grammar may appear as a quasi-terminal symbol in a grammar that depends on it. Quasi-terminal symbols in the dependent grammar are not elements of the terminal alphabet. Their quasy-terminality is indicated by letter T in the superscript of the non-terminal symbol.

The escape-directive and the sytem-directive

By the escape directive the treatment of those escape sequences can be configured which appear on the output, or in macro names and in file names close to the operating system. This means that apart from the escapes **\n** and **\t** backslash keeps its joint and several escaping role in regular expressions and in escape settings. Escapes **\n** and **\t** used in regular expressions and in escape-settings are considered as reserved character representations for characters mapped to keys **ENTER** and **TAB**, respectively.

The primary motivation for introducing the sytem-directive was to allow aML embedding to be processed from aMl embedding. By the system-directive a connection with the operating system can be established and maintained for executing one operating system command from aML embedding.

Zsolt G. Hernáth

Contents

Chapter 1

Introduction

Macro languages, in general, are used to extend host languages by introducing new control flow constructions, data types and operations, originally not defined by them (RATFOR [3], FORMAL [4]). Another reason might be simply shorten host language texts by using parameterized abbreviations for host language text fragments. In the early period of their evolution, macro languages differentiated two categories of macros, namely, macro functions and macro statements. Differentiation was consequently made both in definitions and invocation.

- A definition of a macro function represents a particular fragment of a host language statement or instruction. Expounding an invocation of a macro function embedded by a statement, the invocation is replaced by the text specified in the definition.

- A definition of macro statement contains a sequence of statements that may contain host language statements with occurrent invocations of macro functions and also macro statements. An invocation of a macro statement shapes a statement, and represents the sequence of statements the definition contains. Expounding the invocation, the invocation is substituted by the pure host language text the statements in the definition expand.

aML in many aspects rejects the philosophy of early time macro languages and the expansion methods applied with them. In addition, in aML what are called primal macros[1]can also be defined. Invocations to primal macros expand macro definitions.

1.1 The aML-philosophy

In the aML-philosophy, notions of macro function and macro statement in the classical sense are not differentiated. In general, macro invocations within one statement form

[1]An appellation by Dean Zoltán Horváth

invocations of macro functions. Nevertheless, any of them may de facto expand as either a pure macro function, or a macro statement in the classical sense, or a mixture of those.

The expansion of macro invocations is context-sensitive. Context-sensitivity is based on, and also controlled by invocation contexts. In aML, a macro definition is a possible empty list of rewriting rules. The left-hand sides of rules declare contexts where macro invocations are acceptable, whilst the rigth-hand sides contain text lines the invocations shall be substituted with. Substituted text lines are allowed to take over the all-time invocation context from either side by applying context indicators. The latter is what one has to apply to implement classical sensed macro functions in aML, and called a context presev-ing.

Context preserving needs to specify how far back and ahead relevant text fragments are to be considered as invocation contexts. The range of invocation contexts for different host languages may, however, be different, so that aML introduces the notion of host language statement, and provides lingual means to specify what kinds of or which text fragments constitute statements for the host language considered.

aML supports declarations of parameterized rewriting rules for macro definitions. Formal parameters are considered as symbolic refrences to particular texts — fragments of invocation contexts. References are set when a macro invocation is detected. Neither the references, nor the referenced texts shall be altered during expansion. Rewriting rules can be furnished with unstructured and structured formal parameters. Unstructured formal parameters are called atoms or scalar parameters. Structured parameters are either records or vectors, both of them are relational algebraic relations.

Analogously with other macro languages, aML does primarily proffer directives for macro processor control. Directives are used e.g. to declare or alter macro definitions, or switch their visibility off and on. In addition to directives, aML offers three indicators which are low-level operations. They are in their precedencial order the comment indicator, the context indicator and the link indicator.

1.1.1 Embedding Text Layout

In aML-philosophy, a host language embedding is assumed to be a text stream generated by nesting text files called embedding data stream. Processing embedding data streams, each of them is considered as a sequence of context-free host language units called statements. Statements are the widest language units delimit the visible text context of their macro invocations.

1.1.2 Host Language Phrases as Macro Invocations

The aML breaks away the classical macro technique philosophy. Instead of embedding invocations to predefined or in advance user defined macros, particular phrases in host

language texts are considered and interpreted as invocations of subsequently established adequate macro definitions. This constitutes a particularly great significance when a reedition of formerly issued read-only text is to be made up.

1.1.3 Free-syntax Embedding and Limitations

aML provides free syntax for embedding macro invocations into host laguege texts at three levels:

- aML provides means to declare what or which host language constructions shall be considered as statements by assuming that they can be characterized by some sort of their initial and terminal sub-strings generated by regular grammars; furthermore, allows host statements to embed an arbitrary number of macro invocations.
- aML does not define the set or regular language of acceptable macro names as its own lexical elements, but rather provides means to specify name classes as symbols generated by regular grammars.
- aML does not fix macro name relative positions of actual parameters in macro invocations. Those can rather be defined freely on the left-hand side of rewriting rules in macro definitions. The only thing that aML postulates is that different macro invocations in the same host language statement can not share actual parameters either partly, or as a whole.

The degree of the applicable free syntax embedding is limited by how properly the above categories can be defined by using regular grammars.

1.1.4 Macro Definitions and Macro Expansion

A macro definition in aML assigns an arbitrary number of but at most 256 rewriting rules to a unique name called macro name. Each rewriting rule is context-sensitive in the sense that its left-hand side is an invocotation pattern, i.e a macro invocation together with an embedding context. Macro invocations are primarily identified by macro names.

The expansion of macro invocations is a recursive procedure. It lasts until generated texts are finally free from substituable macro invocations. The expansion of a macro invocation may be context preserving or context independent, and also the mixture of those. It is controlled in the right-hand side of the rewriting rule. The expansion of macro invocations within one statement takes place strictly from left to right, and in the case of a context preserving substitution neighbouring macro invocations are expounded in interaction with each other.

1.2 Terminology and Associate Grammars

Formalism used thorough this paper is in general the usual terminology of Formal Language Theory. The terminal alphabet of the descriptive language is always obtained from the terminal alphabet of the grammar to be described by adding the empty symbol (ε), if needed.

Elements of the non-terminal alphabet show themselves in **boldface** style. Polysyllabic **non-terminal-symbol**s are rather hyphened than underscored. Neighbouring non-terminal symbols on the right-hand side of production rules are separated by one space. Production rules are of form **left-hand-side** $\Longrightarrow \varrho$, where ϱ is an element of the free semi-group supported by the set union of the terminal and the non-terminal alphabets. Whenever production rules with the same left-hand side follow directly each other in consequtive lines, their left-hand side together with the relation symbol $\Longrightarrow$ appears only in the first line. If a serie of production rules enumerated blanks or larger subsets of the terminal alphabet, POSIX-like bracket expressions are used instead on the right-hand sides of production rules. POSIX-like bracket expressions carrying **non-terminal**s indicate optionality. They may be quantified. Square brackets and quantifiers show themselves in **bold**. Quantifiers from extended POSIX may also be used in **bold**.

When aML is in use two associate languages are also in use. One of them is the host language which carries aML embedding, the other one is a regular expression language by which name classes, embedding statements and macro invocations together with their embedding contexts can be described. Associate grammars closely depend on each other. To define them separately, any of them needs to quote production rules of some other.

aML neither stabilize nor recommend any particular host language, namely it does not consider its own grammar a sub-grammar of the host languege actually in use. aML integrates its regular expression grammar, but does not consider that as a sub-grammar of its own grammar. Considering all above, grammars of associate languages on the one hand are expeted to be defined separately, but to do that, dependencies between them have to be capable to be described in separately defined grammars, on the other hand.

Introducing the notion of **quasi-terminal** symbols, dependencies can adequately be specified in separately defined grammars as followes. A quasi-terminal is some non-terminal symbol of a grammar the dependent grammar depends on, but behaves as terminal symbol of the dependent grammar. If **non-terminal** is a non-terminal symbol of a grammar the dependent grammar depends on, the corrspondent quasi-terminal symbol in the dependent grammar appears as **non-terminal**^{T}.

1.2.1 Host Language By aML Eye

Let S denote the terminal alphabet of the host language, $C_{ASCII-8}$ and $S_{ASCII-8}$ in turn the closed decimal code interval $[1, 253]$ of ASCII-8 code table, and the set of symbols

coded by the elements of $C_{ASCII-8}$, respectively. aML assumes $S \subseteq S_{ASCII-8}$. Let I and J be finite index sets, $H_i (i \in I)$ and $T_j (j \in J)$ regular languages over S, and furthermore

$$\mathcal{H} = \bigcup_{i \in I} H_i \quad \text{and}$$
$$\mathcal{T} = \bigcup_{j \in J} T_j.$$

Let S_ε denote the terminal alphabet of the descriptive grammar. aML assumes the host language is given by the grammar $G(S_\varepsilon, N, \textbf{embedding-text}, \mathcal{H}, \mathcal{T}, \mathcal{P})$, where N is the non-terminal alphabet, symbol **embedding-text** $\in N$ is the start symbol of the grammar, and $\mathcal{P}$ is the set of production rules, as listed below:

embedding-text $\Longrightarrow$ **embedding-statement**
embedding-text embedding-statement
embedding-statement $\Longrightarrow (\mathcal{H} \ni) h$ **embedding-fragment** $t (\in \mathcal{T})$
embedding-fragment $\Longrightarrow \varepsilon$
host-symbol
embedding-fragment host-symbol
host-symbol $\Longrightarrow$ **embedding-symbol**
embedding-symbol-code-representation
embedding-symbol $\Longrightarrow s (\in S)$.
embedding-symbol-code-representation $\Longrightarrow c (\in C_{ASCII-8})$.

According to the grammar above, aML assumes that host language statements are identifiable by their prefix and suffix generated by regular languages.

1.2.2 Regular Expression Language for aML

The language defined here is derived from a minimal subset of extended POSIX regular expression language, and called aML regular expression language. The extended POSIX subset chosen containes the dot (`.`) that matches any symbol in the terminal alphabet, the hat (`^`) that matches the start position of the string, parentheses for composing captured (sub-)expressions of form `(expression)`, the choice operator `|` that matches either the expression before or the expression after the operator, the quantifiers `?`, `+`, `*` and the extended POSIX bracket expression.

In aML there is strong need for the character representations of the ENTER and the TAB keys, which are unavailable in POSIX regular expression language.[2]. For them,

[2]POSIX metacharacter `$` matches the position right before the end-of-line character.

therefore, aML takes over escapes `\n` and `\t` from the C programming language. Due to the above, the syntax of POSIX bracket expression has been in significant degree changed: in bracket expressions presence of escapes is allowed, and for position-based characters like hats (`^`) or closing brackets (`]`) and hyphens (`-`), their escaped occurence is acceptable. To document the dependence of aML regular expression language on aML the quasi-terminals **regular-expression-function**^{T}, **macro-parameter-declaration**^{T}, **macro-parameter-reference**^{T} and **parameter-reference**^{T} are introduced.

Grammar

The aML regular expression grammar is given as $G(T, N, \textbf{regular expression}, Q_r, P)$, where T and N are the terminal and the non-terminal alphabet, **regular-expression**$\in N$ is the start symbol, Q_r is the set of quasi-terminals introduced above, and P is the set of production rules. T is assumed be the set of symbols of ASCII-8 code table, and given as $T = T_M \bigcup T_O$, where T_M = {`\`, `^`, `-`, `(`, `)`, `[`, `]`, `.` `?`, `+`, `*`}, and $T_O = T - T_M$.

Symbols

symbol $\Longrightarrow$ **meta-symbol**
normal-symbol

meta-symbol $\Longrightarrow$ $m\ (\in T_M)$

normal-symbol $\Longrightarrow$ **single-symbol**
symbol-class

single-symbol $\Longrightarrow$ **common-symbol**
common-symbol
meta-symbol

common-symbol $\Longrightarrow$ $o\ (\in T_O)$

symbol-class $\Longrightarrow$.
[**single-symbols**]
[^**single-symbols**]
[]
[^]

single-symbols $\Longrightarrow$ **single-symbol**
single-symbol hyphen single-symbol
single-symbols single-symbol

hyphen $\Longrightarrow$ $-$

A **symbol** in regular expressions represents always an element of some set of symbols. Any **common-symbol** identifies the set the only element of which is the **common-symbol** itself. For any **symbol**, other than n and t, the sequence **symbol** is called escaping, which represents either the **symbol** itself or the **common-symbol**-reprezentation of a **meta-symbol**. Escapes \n and \t are used as reserved character representations of ENTER and TAB keys, respectively[3].

A **symbol-class** is either an aML bracket-expression, or the **meta-symbol** dot (.). The **symbol-class** `[]` is the empty set. The empty set is not identical with the empty symbol[4]. The **symbol-class** `[^]` is the complemnetary set of the empty set, which is the whole terminal alphabet and is equivalent with the **meta-symbol** . .

POSIX derived expressions

derived-expression $\Longrightarrow$ **normal-symbol**
| (**derived-expression**)
| **derived-expression quantifier**

POSIX-like-expression $\Longrightarrow$ ε
| **derived-expression**
| **POSIX-like-expression**|**POSIX-like-expression**
| (**POSIX-like-expression**)[**quantifier**]

quantifier $\Longrightarrow$ **at-most-one**
| **at-least-one**
| **zero-or-at-least-one**

at-most-one $\Longrightarrow$?

at-least-one $\Longrightarrow$ +

zero-or-at-least-one $\Longrightarrow$ *

Posix derived expressions defines the minimal subset of te extended POSIX regular expression language taken over and partly redefined.

[3]The escapes \n and \t are not **common-symbol**s. They are mere character representations for **common-symbol**s of key ENTER and TAB.

[4]The empty symbol is not an element of the terminal alphabet. The empty set is a possible representation of the empty symbol.

Function expressions

function-sub-expression $\Longrightarrow$ **POSIX-like-expression**
regular-expression-function^T
(**function-sub-expression**) [**quantifier**]

function-expression $\Longrightarrow$ **function-sub-expression**
function-expression | **function-expression**
(**function-expression**) [**quantifier**]

Regular expressions

macro-expression $\Longrightarrow$ **function-expression**
macro-parameter-declaration^T [**at-most-one**]
macro-parameter-reference^T
parameter-reference^T
(**parameter-reference**^{T}) [**quantifier**]

regular-expression $\Longrightarrow$ **macro-expression**
regular-expression | **regular-expression**
(**regular-expression**) [**quantifier**]
regular-expression regular-expression

Choice operator | is a left-to-right or right-to-left associative operator depending on the direction of matching patterns[5]. The left-to-right resp. right-to-left associativity law has a great importance when the empty symbol is matched. For a left-to-right comparision, patterns `(|A)` and `(A|)` both match the empty symbol in any case, independently of the character string to be matched. Any character string starting with capital `A` is matched by both pattern above, but the empty symbol is also matched. Due to left-to-right associativity law, pattern `(|A)` produce the empty match, whilst pattern `(A|)` results in matching character `A`. Matching zero number of characters in a sub-expression quantified by `?` or `*` results in matching the empty symbol.

Semi-Greedy Quantifcation and Pattern Matching

Matching regular expression patterns, generally two exaggerated kind of quantifications are in use: greedy and lazy. Greedy quantifiers match as many symbol as they can, while lazy quantifiers match the necessary least number of symbols. In POSIX regular expression language quantifiers are greedy. Our aML regular expression language quantifiers are

[5] In aML regular expression pattern matching may take place in any direction

semi-greedy. The semi-greedy quntification[6] is a combined balance between the greedy and lazy quantification. The combined balance between greed and laziness is not simply a golden mean, but pattern dependent. There are patterns by which quantifiers perform greedy match, and also others matched lazy, and in addition, there are such patterns as well which are matched *guarded greedy*. A guarded greedy pattern match is at most as much greedy only as the rest of the pattern could still be matched, provided the entire pattern can be matched at all. The actual behaviour of semi-greedy quantifiers, i.e when and where they follow out greedy, lazy, or guarded greedy pattern match can easily be characterized by neighburing quantified sub-expressions.

Let s and g be quantified **single-symbol**, or **symbol-class** or quantified parenthesized sub-expression composed from them like `(.)*` or `([A-Z0-9])+`. Let S and G denote the non-empty sets of symbols of s and g, respectively. A semi-greedy quantifocation is greedy, or lazy, if $S \bigcap G = \emptyset$, or $S \supset G$ is held, respectively. In any other cases semi-greedy quantifications are guarded greedy.

1.2.3 aML Grammar

Due to the dependency between the aML grammar and its co-grammars quasi-terminal symbols **embedding-symbol**^{T}, **embedding-symbol-code-representation**^{T}, **quatifier**^{T}, **regular-expression**^{T} and their set denoted as Q_a is now introduced.

Let T_a denote the terminal alphabet of aML, S, and $S_{ASCII-8}$ be the sets defined in (1.2.1). Let furthermore $T = T_a \bigcup S \bigcup \{\varepsilon\}$ be the terminal alphabet of the descriptive grammar, and assume $\mathcal{F}$ denote the set of names of files in the file system of the all-time operating system. The grammar which descibes aML is given as $G(T_\varepsilon, N, \textbf{embedding-data-stream}, Q_a, \mathcal{F}, P)$, where N is the non-terminal alphabet, **embedding-data-stream** is the start symbol and P is the set of production rules presented in subsequent chapters.

From now, throughout the formal definition a number of redundant non-terminals and production rules are introduced in order to give more accurate detailes via semantic annotations provided informally. Beyond commentaries the most complex syntactical and semantic aML constructions are illustrated by a number of examples.

[6] Hope, adjective semi-greedy is not engaged so far. If still, sorry for robbing it in this sense.

Chapter 2

aML Basics

2.1 Lexical Elements

The set of terminal symbols of aML is defined as the union of the persistent aML terminal alphabet and the set of terminal symbols of the all-time embedding language. The persistent terminal alphabet of aML consists of ordinary and special symbols. Special symbols are **indicator**s, **costume**s and **delimiter**s. All of them may clash symbols of the host language actually in use. Such conflicts are, however, context-dependent. aML postulates that in any context where **delimiter**s, **costume**s and **indicator**s occur, it interprets them according to their role in the context. aML has been planned that the above contexts with few exceptions are primarily the bodies of macro definitions. Resolving symbol conflicts can be done by using backslash.

2.1.1 Characters

symbol $\Longrightarrow$ **ordinary-symbol**
embedding-symbol^T
delimiter
costume
indicator

Ordinary Symbols

ordinary-symbol $\Longrightarrow$ [- _ * + ? ! / = | < > , ; : 0-9A-Za-z]
white-space

Delimiters

Delimiters are embedding rules for particular grammatical units by marking their initial and terminal positions.

delimiter $\Longrightarrow$ **[**[ ] ' " ( ) { } [: :] .**]**

Delimiters used to mark the initial and the terminal position of the same grammatical units, and with one exception have to constitute homogeneous pairs. The only odd delimiter is called the continuation symbol, and represented by dot (.). Homogeneous pairs are "", '', (), [], {} and [::].

The pair "" called quotation delimits regular expressions. So does the pair '' in particular contexts as well. (cf **Macro Parameter Declaration and Reference**(2.2.1), **Regular Expression Function**(2.2.2), **The embedded-directive**(3.1.1), **The invoked-directive**(3.1.5), **The statement-directive**(3.1.10) and **The term-directive**(3.1.13)).

Grammatical units delimited by pairs "" are line sensitive, i.e they are not allowed to contain **ENTER** character inside. The continuation symbol is used to split double quoted quotations longer than one line into more physical lines. Each line except the last shall contain a continuation symbol that directly follows the right-hand sided quotation mark. Between a continuation symbol and the quotation mark that opens the next quotation fragment, only **white-space**s shall occur.

The pair () separates function names from **parameter-list**s in term declarations and from **value-list**s in **regular-expression-function**s. It may also be used in **macro-parameter-declaration**s, where particular sub-expressions are delimited (cf **Macro Parameter Declaration and Reference**(2.2.1), **Regular Expresion Functions**(2.2.2) and **The term-directive**(3.1.13)).

Pairs [] are used to delimit macro names in macro definitions, and to declare and reference vector parameters (cf **Macro Parameter Declaration and Reference**(2.2.1) and **The macro-directive**(3.1.7)).

Delimiters {} marks off the beginning and the end of directive bodies (cf **Directives**(3.1)).

Finally the pair [: :] indicates the beginning and the end of long-lines which include occasionally more than one physical lines but which are still treated as a single line (cf **aML Comments**(2.1.2), **Lines**(2.4.1)).

Costumes

Similarly to delimiters, **costume**s are also embedding rules for particular lexical elements.

costume $\Longrightarrow$ **parameter-symbol**
directive-symbol
macro-symbol
card
function-symbol
back-slash

parameter-symbol $\Longrightarrow$ %
$

directive-symbol $\Longrightarrow$ #

macro-symbol $\Longrightarrow$ &

card $\Longrightarrow$ ~

function-symbol $\Longrightarrow$ @

back-slash $\Longrightarrow$ \

The **parameter-symbol** `$` is used to enclose **macro-parameter-declaration**s and **macro-parameter-reference**s.

Using the **parameter-symbol** `%` is more complex. To indicate **parameter-tag**s of record parameters, the name of the tag shall be headed by `%`, and to indicate references to function parameters, parameter-names shall be surrounded by `%` symbols (cf **Macro Parameter Declaration and Referenece**(2.2.1), **The term-directive**(3.1.13)).

The **directive-symbol** marks off directives. Though a directive symbol together with a valid directive identifier does only constitute a performable aML directive, checking a **directive-symbol**, the macro processor treats it, as if a performable aML directive were needed to be controlled (cf **Directives**(3.1), **Recognizing and Handling Directives**(4.2.3)).

A **macro-symbol** indicates the position of the **macro-name** on the left-hand side of rewriting rules. Exactly one **macro-symbol** shall occure in each rewriting rule (cf **The invoked-directive**(3.1.5)).

A **card** is a synonym for the **macro-symbol**, but used differently both syntactically and in semantic sense. It can be used in any side of rewriting rules and even any number of times. Its any occurance represents the macro name of the macro invocation the rewriting rule is just being applied for.

The **function-symbol** is an embedding rule for **regular-expression-function**s (cf **Regular Expresion Functions**(2.2.2)).

The **back-slash** plays a joint and several escaping role.

Indicators

Indicators represent low-level aML operations (cf **aML comments**(2.1.2), **Context Indicator**(3.2.2), **Comment Indicator**(3.2.1), **Link Indicator**(3.2.3)).

indicator $\Longrightarrow$ **comment-opener**
comment-terminator
context-indicator
left-link-indicator
right-link-indicator
comment-opener $\Longrightarrow$ {:
comment-terminator $\Longrightarrow$:}
context-indicator $\Longrightarrow$ ...
left-link-indicator $\Longrightarrow$ {*
right-link-indicator $\Longrightarrow$ *}

White Space Characters

gaps $\Longrightarrow$ ε
gap
white-space $\Longrightarrow$ **gap**
new-line
gap $\Longrightarrow$ **blank-or-horizontal-tab**
gap blank-or-horizontal-tab
blank-or-horizontal-tab $\Longrightarrow$ **blank**
horizontal-tab
blank $\Longrightarrow$ **[ ]**
horizontal-tab $\Longrightarrow$ \t
new-line $\Longrightarrow$ \n

In regular expressions derived from **regular-expression**^{T} and in **escape-setting**s of **escape-directive** **\n** and **\t** are reserved character representations for characters mapped to keys **ENTER** and **TAB**, respectively. Outside the above contexts they are real escapes. Their treatment can be prescribed by **escape-setting**s in **escape-directive** (cf **Regular Expression Language for aML**(1.2.2), **The escape-directive**(3.1.2)).

2.1.2 aML Comments

aML comments may stand anywhere in aMl embedding. Though the non-terminal symbol **aML-comment** may stand in more places on the right-hand side of production rules than anyone could imagine, from now on, for clarity of the rules, we will still omit it.

aml-comment $\Longrightarrow$ **comment-opener commentary comment-terminator**

commentary $\Longrightarrow \varepsilon$

commentary-symbol

commentary commentary-symbol

commentary-symbol $\Longrightarrow$ **ordinary-symbol**

embedding-symbol^T

delimiter

costume

comment-opener

context-indicator

left-link-indicator

right-link-indicator

Example for Comments

```
#embedded{"Example for Comments"}
#macro [{:Here a macro named "Example for Comments" will be defined.
          This comment does hopefully desmostrate well what "anywhere"
          really means.:}Example for Comments{:Please, realize, this
          squere bracket syntactical unit is to mark off the initial
          and final position of the macro name: Example for Comments.
          The name has already been specified tightly between this and
          the prceding comment.:}]
{
        #invoked as "&{:This invocation pattern states:
                        Any occurence of Example for Comments anywhere
                        in an embedding statement is an invocation of
                        the macro named 'Example for Comments', and
                        expands the below text with context preserving:}"
        {
                ...An example for aML comments and context preserving...
        }
}
```

2.2 Parameter and Function Expressions

Regular expressions play a basic role in aML. One of the most important features of the language is that the free syntax of aML macro calls is limited insofar as they can be described in regular expressions. Parameter and function expressions are particular aML phrases representing captured regular expressions. They are de facto macro parameter declarations and references together with regular expression functions.

2.2.1 Macro Parameter Declaration and Reference

Macro parameter declarations are formal parameter declarations for rewriting rules. They are typified in the sense that they specify the set of acceptable actual values called domain. Domains are regular sub-expressions derived from **regular-expression**^{T}. Macro parameter declarations are captured sub-expression of left-hand sides of rewriting rules which are in deed regular expressions derived from **regular-expression**^{T}. Macro parameter declarations may be qualified optional by declaring them with quantifier `?`.

Domains are allowed to be composed from domain segments. Such domains are called structured domains. Segments of structured domains can be further, i.e, recursively structured. A structured domain may constitute a record or a vector. Both of them are relational algebraic relations. Unstructured domains are also called scalar domains. Scalar domains declare sets of mere character string values.

Records are n-tuples for some fixed number $n(\geq 1)$. Items of tuples are separated by declared delimitations. Vectors are special records. Items of vectors are called elements. Elements have a common domain, and they are separated by the same delimitation. The number of elements (the size of the vector) is not in advance declared. It is determined dynamically when the value (list of elements) for the vector parameter is matched. Vectors are ordered sets of binary relations between their elements and the indexes of their elements.

Vectors whose elements are structured as vector are called multi-dimensional vectors. Multi-dimensional vectors are not arrays. For some $n(> 1)$ elements of an n-dimensional vector are indexed by n indexes. The first index is for the first, the n^{th} is for the n^{th} dimension. The elements of the innermost vector depending on their declaration are scalars or records. For some $1 < i <= n$, the elemnts of the vector in the i^{th} dimension are i-1 dimensional vectors, consisting of occasionally different number of elements.

Macro parameter declarations are only accepted on the left-hand sides of rewriting-rules defined for macro definitions. In any other contexts they in most cases are parsed but either cause an error or treated as **ordinary-text**s.

macro-parameter-declaration $\Longrightarrow$ $**parameter-name selector**$
parameter-name $\Longrightarrow$ **identifier**
identifier $\Longrightarrow$ **[**0-9A-Za-z_-**]**+
selector $\Longrightarrow$ **atom**
record
vector
atom $\Longrightarrow$ (**'domain'**)
record $\Longrightarrow$ { **gaps tag-selectors gaps** }
tag-selectors $\Longrightarrow$ **tag-selector**
tag-selectors gaps delimitation gaps tag-selector
tag-selector $\Longrightarrow$ **parameter-tag selector**
parameter-tag $\Longrightarrow$ %**identifier**
vector $\Longrightarrow$ [**gaps element gaps delimitation gaps** | **gaps vector-termination**]
element $\Longrightarrow$ **selector**
vector-terminaton $\Longrightarrow$ **delimitation**
delimitation $\Longrightarrow$ **'domain'**
domain $\Longrightarrow$ **regular-expression**^{T}

The **vector-termination** must not be an actual or derived empty symbol, furthermore neither **delimitation**s nor **vector-termination**s must clash **atoms**.

Macro parameters are not variables, rather symbolic references to **ordinary-text**s. Assignments of referenced values take place when invocations of macros are matched. A macro parameter declared on the left-hand side of a rewriting rule can only be referenced on its right-hand side.

A **macro-parameter-reference** may refer to its entire scalar value, or to its structured value according to the declaration. Badly formatted **macro-parameter-reference**s, or those which are undeclared, are considered as **ordinary-text**s. (cf **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7)).

Records can be referenced by fields. References to a field of structured domain may pertain to its entire scalar, or to its structured value. Vectors can not be referenced by indexed elements. A reference to a vector parameter results in the series of all elements in a particular order.

macro-parameter-reference $\Longrightarrow$ $**parameter-name reference-type**$

reference-type $\Longrightarrow$ **atom-reference**
record-reference
vector-reference
atom-reference $\Longrightarrow \varepsilon$
record-reference $\Longrightarrow$ **parameter-tag reference-type**
vector-reference $\Longrightarrow$ [**reference-type element-ordering**]
element-ordering $\Longrightarrow$ **row-ordering**
column-ordering
row-ordering $\Longrightarrow$ ||
column-ordering $\Longrightarrow$ =

Referring to vector parameters **element-ordering** has to be specified for each dimension. Prescribing **column-ordering** the order of elements may be varied, and elements are distributed over more mere lines. Neighbouring elements in one-dimensional vectors are separated by one **blank** in a **row-ordering** reference. In **column-ordering** references, separators are **new-line**s.

Example

```
#embedded{"A"}
#macro [A]
{
        #invoked as "& $v[[[('[a-z]')'-'|'='']':'|'!'']','|'\.']$"
        {
                ...Series of elements of vector v appears as:...
                ...=======================================...
                ...$v[[[||]=]||]$...
        }
}
(W) Empty string (sub-)match may cause undesired pattern match
        line 4 in segment STDIN
A a-b-c-d=:e-f-g=:h-i=!,j-k-l=:m-n=:o=!,p-q-r-s-t-u-v-x-y-z=!.
Series of elements of vector v appears as:
=======================================
a b c d j k l p q r s t u v x y z
e f g m n
h i o
```

Example shows a name class declaration, a macro definition named `A`, an invocation of macro `A`, and the result of the invocation. Macro `A` declares a three-dimensional vector

parameter named `v`. Processing the macro definition, the aML processor sends a warning onto stderr indicating an empty **left-shade-decl**. When the macro invocation matched, the values for the elements of parameter `v` together with their element indexes are mapped as follows:

```
a[0,0,0], b[1,0,0], c[2,0,0], d[3,0,0], e[0,1,0], f[1,1,0], g[2,1,0], h[0,2,0],
i[1,2,0], j[0,0,1], k[1,0,1], l[2,0,1], m[0,1,1], n[1,1,1], o[0,2,1], p[0,0,2],
q[1,0,2], r[2,0,2], s[3,0,2], t[4,0,2], u[5,0,2], v[6,0,2], x[7,0,2], y[8,0,2],
z[9,0,2].
```

Mapping of element indexes is controlled by **delimitation**s and **vector-termination**s, specified for dimensions.

In case of referring to multi-dimensional vectors the order of elements and their view can easily be determined inductively. Suppose, v is an n-dimensional vector and for some $1 < i <= n$ the order of the elements, and their view in the $(i-1)^{th}$ dimension is already known. Recall that the i^{th} dimension is a vector whose elements are $(i-1)$-dimensional vectors. Elements of the vector of i^{th} dimension will be listed in their view below each other or next to each other depending on whether **column-ordering** or **row-ordering** is specified for the i^{th} dimension.

In our example, the 3^d dimension is an 1-dimensional vector with 2-dimensional vector elements. The **delimitation** of elements is the comma (`,`) which occurs two times in the invocation, i.e the vector has three elements whose indexes are 0, 1 and 2. Since **element-ordering** in the ^{d} dimension is **row-ordering**, its elements are replaced beside each other separated with one **blank**.

The refernce to the 2^{nd} dimension is **column-ordering**. The **vector-termination** and **delimitation** are the characters `=` and `-`, respectively.

Referencing of the 1^{st} dimension is **row-ordering**. The **delimitation** is the hyphen, and **vector-termination** is the equality sign. Considering all of these, values of elements of the 3^d dimension are **row-ordering** row vectors in **column-ordering** views as follows:

```
0-indexed element:
a b c d
e f g
h i

1-indexed element:
j k l
m n
o

the only row vector in column-continuous view indexed by 2
p q r s t u v x y z
```

The **row-ordering** elements of the 3^d dimension are placed beside each other in such a way as to the **column-ordering** row vectors of those are sparated by one **blank**:

```
a b c d j k l p q r s t u v x y z
e f g m n
h i o
```

which is in deed the result of invocation of macro `A` in the example.

2.2.2 Regular Expression Function

regular-expresion-function $\Longrightarrow$ **nametag**(**gaps [value-list] gaps**)
nametag $\Longrightarrow$ **@identifier**
value-list $\Longrightarrow$ **value**
value gaps , gaps value-list
value $\Longrightarrow$ '**regular-expression**^{T}'

Regular expression functions are parametrized symbolic referenes to parameterized terms can be defined by the **term-directive**. Inside **left-hand-side**s of **rewriting-rule**s they result in regular (sub-)expressions, whilst in other contexts they are treated as **ordinary-text**s (cf **Regular Expression Language for aML**(1.2.2), **Parameter Evaluation**(2.2.3), **The invoked-directive**(3.1.5), **The term-directive**(3.1.13)).

2.2.3 Parameter Evaluation

Parameter evaluations are procedures when particular texts are made temporarily referable by declared identifiers. They take always place, whenever macro invocations are matched or regular expression functions are performed.

Parameter evaluation for a macro invocation are controlled by the regular expression which itself is the matched left-hand side of the rweriting rule in the macro definition. Parameters of regular expression functions are evaluated from left to right. Correspondence between parameters and function arguments are positional (cf **Regular Expression Language for aML**(1.2.2), **Macro Parameter Declaration and Reference**(2.2.1), **Regular Expression Function**(2.2.2), **The term-directive**(3.1.13)).

2.3 Name Classes, Macro Names and Rule Names

aML does not define by default the set of the acceptable macro names as its own lexical elements. The set of acceptable macro names are the union of **name-class**es. A

name-class is a regular language defined by a regular expression can be derived from **regular-expression**^{T}, and declared with the embedded-directive (cf i**Regular Expression Language for aML**(1.2.2), **The embedded-directive**(3.1.1)).

2.3.1 Name Classes

name-class $\Longrightarrow$ **name-class-reference**
name-class-quotation
name-class-reference $\Longrightarrow$ **regular-expression-function**
name-class-quotation $\Longrightarrow$ " **regular-expression**^{T} "

A **name-class-quotation** may be splitted into more than one line by using the continuation symbol (.). A **name-class** containing the empty symbol is unacceptable (cf **Delimiters**(2.1.1), **The embedded-directive**(3.1.1)).

2.3.2 Names

Names in aML are **macro-name**s and **rule-nme**s, i.e names of **rewriting-rule**s. A name is a **macro-name**, if it is declared by a macro-directive. Escapes in **macro-name**s are interpreted according to the last time established **escape-setting**.

A **macro-name** is recognisable, if it is an element of a declared **name-class**, unless it is splitted by **border**s of **declared-statement-class-statement**s. At any particular time unrecognisable **macro-name**s can be made recognisable by posteriorly declared adequate **name class** (cf **Statement Classes and Statements**(2.4), **The embedded-directive**(3.1.1), **The escape-directive**(3.1.2), **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7)).

2.4 Statement Classes and Statements

Statements are the widest text frames delimit the visible text context of their macro invocations. By default, aML defines three statement classes: mere line, **long-line** and **directive-line**. All of them are context-dependent. A mere line is a single line terminated by the `ENTER` character. A **long-line** is a sequence of single lines following each other, enclosed by the delimiter pair `[::]`. A **directive-line** is a mere line which apart from leading white space characters starts with the **directive-symbol** which is directly followed by an aML directive keyword.

Besides the default statement classes aML provides also the power of declaring additional what are called **declared-statement-class**es which are regular languages defined by

the statement-directive (cf **Host Language By aML Eye**(1.2.1), **Deleimiters**(2.1.1), **Costumes**(2.1.1) and **The statement-directive**(3.1.10)).

2.4.1 Lines

inner-single-line $\Longrightarrow$ **inner-directive-line**
ordinary-inner-embedment
outer-single-line $\Longrightarrow$ **outer-directive-line**
host-embedment
ordinary-inner-embedment $\Longrightarrow$ **incomplete-embedment**
embedment
host-embedment $\Longrightarrow$ **ordinary-text**
outer-macro-invocation
host-embedment host-embedment
incomplete-embedment $\Longrightarrow$ **left-embedment**
right-embedment
middle-embedment
left-embedment $\Longrightarrow$ **embedment context-indicator**
right-embedment $\Longrightarrow$ **context-indicator embedment**
middle-embedment $\Longrightarrow$ **context-indicator embedment context-indicator**
embedment $\Longrightarrow$ ε
ordinary-text
text-reference
inner-macro-invocation
link-indicator
embedment embedment
ordinary-text $\Longrightarrow$ **ordinary-symbol**
embedding-symbol^T
ordinary-text ordinary-text
text-reference $\Longrightarrow$ **card**
text-reference macro-parameter-reference
text-reference regular-expression-function
link-indicator $\Longrightarrow$ **left-link-indicator embedment right-link-indicator**
long-line $\Longrightarrow$ |: **[context-indicator] embedment* [context-indicator]** :|

Lines cover the three default statement classes.

2.4.2 Declared Statement Classes

A **declared-statement-class** is specified by two regular expressions. One for matching the head and the other matching of the tail of the statement. Any of them may be missing except both yet neither of them shall derive the empty symbol (cf **Host Language By aML Eye**(1.2.1), **The statement-directive**(3.1.10)).

declared-statement-class $\Longrightarrow$ **context-declaration**
context-declaration $\Longrightarrow$ **open-context**
closed-context
open-context $\Longrightarrow$ **left-context**
right-context
left-context $\Longrightarrow$ **border context-indicator**
right-context $\Longrightarrow$ **context-indicator border**
closed-context $\Longrightarrow$ **border context-indicator border**
border $\Longrightarrow$ **border-quotation**
border-reference
border-quotation $\Longrightarrow$ " **regular-expression**^{T} "
border-reference $\Longrightarrow$ **regular-expression-function**

A **border-quotation** may be splitted into more than one line following the rule for division of quotations.

2.5 Embedding and Embedding Contexts

The aML embedding are directives and macro invocations in **embedding-data-stream**s. The former control the input, and output, change the processing environment of the aML macro processor, and also ensures connection to the all time operating sytem. The latter in turn are to be expounded (cf **Macro Invocations**(2.5.2), **Directives**(3.1)).

Dierctives with few exceptions are context sensitive only in the sense that the embedding contexts for particular directives are limited differently. The **outer-directive**s are, for instance, applicable only outside macro definitions, whilst **inner-directive**s anywhere. Few of them, however, depend on their embeding context as well. The output target of the host-directive, for example, depends on if it has been performed by the expand-directive. In contrast, expounding macro invocations of a statement are context-sensitive

in any sense. The non-terminal **embedding-statement** is used only to symplify semantic explanations (cf **Directive Embedding**(2.5.1), **Macro Invocations**(2.5.2), **The expand-directive**(3.1.3), **The host-directive**(3.1.4), **Context Indicator**(3.2.2)).

embedding-statement $\Longrightarrow$ **inner-embedding**
outer-embedding
inner-embedding $\Longrightarrow$ **inner-single-line**
long-line
outer-embedding $\Longrightarrow$ **outer-single-line**
declared-statement-class-statement
declared-statement-class-statement $\Longrightarrow$ **host-embedment**+

aML distinguishes between the embedding that are carried on the **right-hand-side**s of the **rewriting-rule**s, and those which embedded outside them. The former is called **inner-embeddding**, the latter in turn **outer-embeddding** (cf **Lines**(2.4.1), **Declared Statement Classes**(2.4.2), **aML Embedding**(2.5.3).

2.5.1 Directive Embedding

In comparision with **inner-directive**s the permissible embedding contexts for the invoked-directive and expand-directive is strongly limited. The **invoked-directive** may only be adopted by the **macro-directive**. Thex **expand-directive** is accepted only in **right-hand-side**s of **rewriting-rule**s. Using **outer-directive**s inside macro definitions are forbidden.

inner-directive-line $\Longrightarrow$ **expand-directive**
invoked-directive
inner-directive
outer-directive-line $\Longrightarrow$ **outer-directive**
inner-directive
inner-directive $\Longrightarrow$ **host-directive**
load-directive
macro-directive
off-directive
on-directive
system-directive
target-directive

outer-directive $\Longrightarrow$ **embedded-directive**
escape-directive
statement-directive
term-directive

The expand-directive is sensible for free occurrences of symbol `}` in its **inner-embeding**. An occurrence of symbol `}` is called bound if it closes a directive body. Unbound instances are called free. The host-directive considers any non-escaped occurence of symbol `}` as closing of its **host-embedding**.

The expand-directive parses and performs directives and macro invocations included by its **inner-embeding**. Unless free occurrences of symbol `}` are escaped, the first one ends the processing of the **inner-embedding**.

The host-directive neither parses nor performs its **host-embedding**, so that the first occurence of symbol `}` which is not escaped ends the output of its **host-embedding** up.

2.5.2 Macro Invocations

macro-invocation $\Longrightarrow$ **outer-macro-invocation**
inner-macro-invocation
left-shade $\Longrightarrow$ **outer-left-shade**
inner-left-shade
right-shade $\Longrightarrow$ **outer-right-shade**
inner-right-shade
outer-macro-invocation $\Longrightarrow$ **outer-left-shade macro-name outer-right-shade**
outer-left-shade $\Longrightarrow$ **outer-left-context**
outer-left-context $\Longrightarrow$ ε
ordinary-text
outer-right-shade $\Longrightarrow$ **outer-left-shade**
inner-macro-invocation $\Longrightarrow$ **inner-left-shade macro-name inner-right-shade**
inner-left-shade $\Longrightarrow$ **inner-left-context**
inner-left-context $\Longrightarrow$ **outer-left-context**
text-reference
inner-right-shade $\Longrightarrow$ **inner-left-shade**

Nonterminals **macro-invocation**, **left-shade** and **right-shade** cannot be derived from the start symbol. They are introduced only to simplify semantic annotations.

Unless **left-shade-decl** resp. **right-shade-decl** are parameter-free, a **left-shade** resp. a **right-shade** carries the left-hand resp. the right-hand sided macro parameter values of an **outer-macro-invocation** or an **inner-macro-invocation**.

A **left-shade** resp. a **right-shade** of a **macro-invocation** shall be matched by a **left-shade-decl** resp. a **right-shade-decl** of a **rewriting-rule** of the **macro-definition** identified by the **macro-name**, and registered in the macro library (cf **Statement Classes and Statements**(2.4), **Lines**(2.4.1), **aML Embedding**(2.5.3)). **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7), **Recognizing and Handling Macro Invocations**(4.2.4), **Expounding Macro Embedding**(4.3.3)).

An **embedding-statement** may embed an arbitrary number of macro invocations. For the **right-hand-side**s of **rewriting-rule**s applied in expounding macro invocations embedded by one **embedding-statement**, the following shall be held

- The **right-hand-side** of the **rewriting-rule** applied for the left-most resp. right-most macro invocation may embed an arbitrary number of **line**s being imprefect from the left resp. from the right.
- In the case of neighbouring macro invocations, the **right-hand-side** of the **rewriting-rule** applied for the left-hand-sided neighbour shall contain exactly so many **lines** being imperfect from the right, as many **lines** being imperfect from the left are embedded by the right-hand-sided neghbour.

Example

Example illustrates the implementation of a macro definition when the **right-hand-side** of its only **rewriting-rule** is partly generated by applying the expand-directive. Suppose there is a file named `ZZZ` in the parent directory of the one where the aML processor has been started from. File `ZZZ` contains two macro definitions, each of them with one anonym rewriting rule. In the last line of the file both macros are invoked. Since both will be the **inner-macro-invocation**s of the macro definition named `A`, the **context-indicator**s specified on both sides are treated as **ordinary-text**s. The content of the file:

```
#macro [B]
{
        #invoked as "&"
        {
                ...Q...
        }
}
```

```
#macro [C]
{
        #invoked as "&"
        {
                ...W...
        }
}
                ...B-C...
```

Let us type in the following aMl text on the standard input:

```
#embedded{"B|C"}
#macro [A]
{
        #invoked as "&"
        {
                #expand
                {
                        #load{../ZZZ}
                }
                #system{date}
                ...W...
        }
}
```

Parsing macro definition named `A` the expand-directive loads file `ZZZ`. Closing of the standard input, the macro library is displayed as

```
aML Macro Library
=================
Macro A
Rules{anon[0.0]:visible }
Definition:
#macro [A]
{
        #invoked as "&"
        {
                ...Q-W...
                #system{date}
                ...W...
        }
}
```

```
Macro B
Rules{anon[0.0]:visible }
Definition:
#macro [B]
{
        #invoked as "&"
        {
                ...Q...
        }
}
Macro C
Rules{anon[0.0]:visible }
Definition:
#macro [C]
{
        #invoked as "&"
        {
                ...W...
        }
}
```

Loading file ZZZ macro definitions named `B` and `C` are registered in the macro library, their invocations in turn are loaded into the **right-hand-side** of the **rewriting-rule** of macro named `A`. Finally the macro definition named `A` is also put up in the macro library.

2.5.3 aML Embedding

The aML considers a host language embedding as **embedding-data-stream**s nested in an arbitrary depth. An **embedding-data-stream** is a data file in the all-time operating system environment, and named by an **embedding-data-straem-name** (cf **Host Language By aML Eye**(1.2.1), **The load-directive**(3.1.6)).

$$\textbf{embedding-data-stream} \Longrightarrow \textbf{outer-embedding}^*$$

The host language text by aML eye are possible empty sequence of **outer-single-line**s and **declared-statement-class-statement**s.

Chapter 3

Directives and Indicators

3.1 Directives

Dierectives are to control the macro processor, and access the all time operating system. Each directive consists of the **directive-symbol** followed by a directive keyword and a directive body enclosed by the **delimiter** pair `{}`.

A common syntactical rule is that the line carrying the directive symbol can only adopt blanks and horizontal tabs preceding the **directive-symbol**, and the body closing `}` symbol can only be followd by white space characters terminated by **new-line**.

Apart from the above, and apart from those **white-space**s which are obliged to use by expand-directives, invoked-directives, and host-directives, **white-space**s can be used freely in directives.

3.1.1 The embedded-directive

The embedded-directive is used to declare **name-class**es, and can be applied any number of times in any **embedding-data-stream**. A declared **name-class** can not be removed or swithched off (cf **Name Classes, Macro Names and Rule Names**(2.3), **Directive Embedding**(2.5.1)).

embedded-directive $\Longrightarrow$ #embedded **wsp** {**wsp name-classes wsp**}
name-classes $\Longrightarrow$ **name-class** [**wsp** , **wsp name-classes**]
wsp $\Longrightarrow$ **white-space***

Since declared **name-class**es may overlap each other, the order of their declarations is significant (cf **Recognizing and Handling Macro Invocations**(4.2.4)).

3.1.2 The escape-directive

The escape directive overrides escape rules currently being in force by defining new ones. An **escape-setting** is an aML extension of POSIX bracket expression (cf **Output**(4.1.2)).

escape-directive $\Longrightarrow$ #escape **wsp**{**wsp escape-settings wsp**}
escape-settings $\Longrightarrow$ ε
escape-setting [**wsp escape-settings**]
escape-setting $\Longrightarrow$ [**escaped-symbols**]
escaped-symbols $\Longrightarrow$ ε
escaped-symbol
escaped-symbol - **escaped-symbol**
escaped-symbols escaped-symbol
escaped-symbol $\Longrightarrow$ **embedding-symbol**^{T}
embedding-symbol^T (**interpretation**)
interpretation $\Longrightarrow$ **embedding-symbol**^{T}
scale embedding-symbol-code-representation^T
scale $\Longrightarrow$ ε
octal
decimal
hexadecimal
octal $\Longrightarrow$ O
o
decimal $\Longrightarrow$ D
d
hexadecimal $\Longrightarrow$ X
x

Escapes can be basically interpreted in three different ways:

(1) as the escaped character preceded by backslash,

(2) as an **embedding-symbol**^{T} occasionally given as a visible symbol or its ASCII-8 code representation,

(3) as the escaped character without the preceding backslash.

An **embedding-symbol**^{T} without **interpretation** interpreted according to (1). Each **embedding-symbol**^{T} furnished by **interpretation** interpreted according to (2), and

any other symbol (left out from **escape-setting**) are interpreted according to (3).

Preparing more than one **escape-setting**, escape rules are determined by the rightmost **escape-setting** specified.

When **escape-settings** is the empty symbol, i.e directive body is empty, the default escape rules are set. The default escape rules can also be achieved by applying `#escape{[n(o12)t(o11)().+-*?]`.

The escape-directive `#escape{[a(A)-z(Z)r(o15)]}` states that escaped smaller case letters are interpreted as their correspondent uppercase letters, and escape \r is interpreted as carriage return – the character coded by octal 15 in ASCII-8 code table.

3.1.3 The expand-directive

The macro-directive directs the **inner-embed**s on the **right-hand-side**s of their **rewiting-rule**s to the macro library without processing directives, indicators and macro-invocations, unless those are inside the **directive-body** of an enxpand-directive (cf **Embedding and Embedding Contexts**(2.5), **The macro-directive**(3.1.7)).

expand-directive $\Longrightarrow$ #expand **wsp directive-body**
directive-body $\Longrightarrow$ { **wsp inner-embedding* wsp** }

In the case of a non-empty[1] **directive-body** the following shall be taken care of.

- Lines carrying the symbols of **delimiter** pair {} shall contain no fragments of the text embedded by the **directive-body**;
- blanks and horizontal tabs preceding **right-embedment**s, **middle-embedment**s and **long-line**s, and those which follow **middle-embedment**s, **left-embedment**s and **long-line**s do not belong to the embedding text (cf **Lines**(2.4.1));
- with an eye on the above, the embedded text starts with the first (visible or invisible) character of the line which directly follows the one carrying the body opener `{` symbol, and ends with the newline character of the line directly followed by the one which contains the body terminator `}` symbol.

3.1.4 The host-directive

The host-directive is used to hide directives, macro invocations, **text-reference**s and **indicator**s.

Unless the host-directive is performed by the expand-directive, **host-embedding** is

[1] For an empty **directive-body**, the constraints are obviously carried by the syntatic rule.

directed onto the all-time target data stream without parsing and processing.

host-directive $\Longrightarrow$ #host **wsp** {**host-embedding wsp** }
host-embedding $\Longrightarrow$ ε
host-embedding symbol

If the host-directive is performed by an expand-directive, **host-embedding**, which shall be **inner-embedding**, is outputted onto the **right-hand-side** of the **rewriting-rule** which brings the expand-directive to effect.

The **host-embedding** enclosed by symbols `{` and `}` shall follow the rules stated for the **directive-body** of the expand-directive (cf **Embedding and Embedding Contexts**(2.5), **The expand-directive**(3.1.3), **The target-directive**(3.1.12), **Output**(4.1.2)).

3.1.5 The invoked-directive

The invoked-directive is used to define **rewriting-rule**s for macro definitions (cf **Macro Invocations**(2.5.2), **The macro-directive**(3.1.7)).

invoked-directive $\Longrightarrow$ #invoked **white-space+ rule-declaration**
rule-declaration $\Longrightarrow$ **left-hand-sides wsp right-hand-side**
left-hand-sides $\Longrightarrow$ [**rule-name white-space+**] as **white-space+ left-hand-side**
left-hand-sides white-space+ or **white-space+ left-hand-sides**
rule-name $\Longrightarrow$ **identifier**
left-hand-side $\Longrightarrow$ **invocation-context-pattern**
invocation-context-pattern $\Longrightarrow$ "**separator invocation-pattern separator**"
invocation-pattern $\Longrightarrow$ **left-shade-decl* macro-symbol right-shade-decl***
left-shade-decl $\Longrightarrow$ [**macro-parameter-declaration**] **separator**
right-shade-decl $\Longrightarrow$ **separator** [**macro-parameter-declaration**]
separator $\Longrightarrow$ ε
separator ordinary-text
separator text-reference
right-hand-side $\Longrightarrow$ **directive-body**

A **rule-name** is local to the **macro-definition** the invoked-directive is carried by. An **invocation-context-pattern** is a regular expression, specifying a context pattern not necessarily narrower than the associated statement where the macro invoked from.

Each **invocation-context-pattern** shall contain one **macro-symbol** (&) which symbolizes the macro name in the pattern. The **macro-symbol** is the base position of matching the **invocation-context-pattern**. The pattern may also contain an arbitrary number of **card**s; each is an occurence of the **macro-name** matched at the base posotion (cf **Costumes**(2.1.1), **Name Classes, Macro Names and Rule Names**(2.3), **Embedding and Embedding Contexts**(2.5), **Recognizing and Handling Macro Invocations**(4.2.4)).

An **invocation-context-pattern** may be splitted into more than one line following the rule of division of quotations (cf **Delimiters**(2.1.1)).

A **left-shade-decl** and **right-shade-decl** may declare formal parameters which may be referenced in the **directive-body**. Undeclared **macro-parameter-reference**s are treated as **ordinary-text**s. A **separator** is a possible empty, and in general also possible **macro-parameter-declaration** free regular subexpression (cf **Macro Parameter Declaration and Reference**(2.2.1), **Macro Invocations**(2.5.2)).

3.1.6 The load-directive

The load-directive is used to hang up reading the **embedding-data-stream** the macro processor is actually reading statements from. Processing goes on with reading statements from the **embedding-data-stream** named by **embedding-data-stream-name**. The nameless **embedding-data-stream** is the standard input.

$$\textbf{load-directive} \Longrightarrow \#\text{load}\ \textbf{wsp}\ \{\textbf{embedding-data-stream-name}\}$$
$$\textbf{embedding-data-stream-name} \Longrightarrow \varepsilon \qquad \text{file-name} \in \mathcal{F}$$

Syntactical rules of composing an **embedding-data-stream-name** depends only on the all-time operating system environment. In order not to restricting syntactical rules of composing file names, aML syntax allowes **white-space**es and escaped symbols anywhere within file names, even leading and trailing **white-space**es as well. Escaped symbols in file-names are interpreted according to the recently applied escape-directive, or by the default **escape-setting** (cf **aML Grammar**(1.2.3), **Input**(4.1.1)).

3.1.7 The macro-directive

A macro-directive is used to create or modify **macro-definition**s. A **macro-definition** declares a **macro-name** or more, and a sequence of **rewriting-rule**s associated with the **macro-name**, or **macro name**s declared. The **right-hand-side** of a **rule-declaration** in a **rewriting-rule** specifies what the matched macro invocation will be replaced with.

$$\textbf{macro-directive} \Longrightarrow \#\text{macro}\ \textbf{white-space}+\ \textbf{macro-definition}$$

macro-definition $\Longrightarrow$ **names white-space* macro-body**
names $\Longrightarrow$ [**macro-name**] [**white-space***|**white-space* names**]
macro-name $\Longrightarrow$ **symbol**+
macro-body $\Longrightarrow$ {**rewriting-rules***}
rewriting-rules $\Longrightarrow$ **rewriting-rule**[**new-line**+ **rewriting-rules**]
rewriting-rule $\Longrightarrow$ **invoked-direktiva**

Unless **names** are already registered, the **macro-definition** is put up in the macro library under the **names** declared, otherwise the **macro-body** resides in the macro library is updated. As for the **macro-body**, rules stated by expand-directive for **directive-body** shall be followed (cf **The expand-directive**(3.1.3).

The order of **rewriting-rule**s within a **macro-definition** is significant: in the order of declarations, the first matched is taken to expound the macro invocation (cf **Name Classes, Macro Names and Rule Names**(2.3), **The invoked-directive**(3.1.5), **Recognizing and Handling Macro Invocations**(4.2.4)).

3.1.8 The off-directive

The off-directive is used to make rewriting rules invisible. An off-directive can be applied anywhere in an **embedding-data-stream**.

off-directive $\Longrightarrow$ #off **wsp** {**rewriting-rule-name**}
rewriting-rule-name $\Longrightarrow$ **name-prefix name-suffix**
name-prefix $\Longrightarrow$ ε
card
macro-name
name-suffix $\Longrightarrow$ ε
. **rule-name**

A **rewriting-rule-name** is composed from a **macro-name** and a **rule-name** local to the **macro-definition** named by **macro-name**. A **rewriting-rule-name** unambiguously identifies a **rewriting-rule**, if both **name-prefix** and **name-suffix** are specified. If **name-prefix** is missing, the off-directive has to be carried by the **rewriting-rule** is actually being performed, and the **rewriting-rule** named as **rule-name** becomes invisible within the **macro-definition** the **rewriting-rule** being processed is adopted by. For an empty bodied off-directive (both **name prefix** and **name-suffix** are missing) the **rewriting-rule** made invisible is the one which performs the empty bodied off-directive (cf **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7)).

If only **name-prefix** is specified, each **rewriting-rule** of the **macro-definition** named by **name-prefix**, becomes invisible.

Making invisible names to invisible is ineffective.

3.1.9 The on-directive

The on-directive is used to make **rule-names** visible. It makes **rewriting-rule**s visible in the same way as the off-directive makes them invisible (cf **The off-directive**(3.1.8)).

on-directive $\Longrightarrow$ #on **wsp** {**rewriting-rule-names**}

Making visible names to visible is ineffective.

3.1.10 The statement-directive

The statement-directive is used to declare **statement-class**es for host languages (cf **Declared Statement Classes**(2.4.2)).

statement-directive $\Longrightarrow$ #statement **wsp** {**wsp statement-classes wsp**}
statement-classes $\Longrightarrow$ **declared-statement-class** [**wsp** , **wsp statement-classes**]

Declared **statement-class**es may overlap each other, and the order of their declarations are, therefore, significant (cf **Statement Classes and Statements**(2.4), **Input**(4.1.1)).

3.1.11 The system-directive

The system-directive is used to connect the all-time operating system to execute one **command-line** from an aML embedding.

system-directive $\Longrightarrow$ #system **wsp** {**command-line**}
command-line $\Longrightarrow$ **host-embedding**

3.1.12 The target-directive

The target-directive is used to assign a target data stream for macro processor output.

target-directive $\Longrightarrow$ #target **wsp** {**target-data-stream-name**}
target-data-stream-name $\Longrightarrow$ ε
STDERR
file-name $\in \mathcal{F}$

Unless the actual target data stream is the nameless standard output or the standard error output named STDERR, the target-directive closes the target data stream actually fed, and unless the **target-data-stream-name** is ε or STDERR, opens the one named **target-data-stream-name** (cf **Output**(4.1.2)).

3.1.13 The term-directive

The term-directive is used to declare parameterized terms which can later be invoked as regular expression functions (cf **Regular Expression Function**(2.2.2)).

term-directive $\Longrightarrow$ #term **wsp** {**term-declarations**}
term-declarations $\Longrightarrow$ **term-declaration** [**wsp new-line term-declarations**]
term-declaration $\Longrightarrow$ **term-head** = **term-body**
term-head $\Longrightarrow$ **identifier** (**wsp** [**parameter-list**] **wsp**)
parameter-list $\Longrightarrow$ **parameter-name** [**wsp** , **wsp parameter-list**]
term-body $\Longrightarrow$ "**regular-expression**^{T}"
parameter-reference $\Longrightarrow$ %**parameter-name**%

The **term-body** may be splitted into more than one line following the rule of division of quotations (cf **Delimiters**(2.1.1)).

The non-terminal **parameter-reference** is introduced only to indicate the dependency of the aML regular expression garammar from aML (cf **Regular Expression Language for aML**(1.2.2))

Each **parameter-name** in **term-head** should appear as **parameter-reference** in the **term-body**. Declared **parameter-name**s are local to the regular expression term (cf **Regular Expression Function**(2.2.2) and **Parameter Evaluation**(2.2.3)).

3.2 Indicators

Indicators are precedence-dependent and context-sensitive low-level operations. They are the comment indicators, the **context-indicator** and the **link-indicator** in descending order of their precedences. The latter two are only recognisable inside **right-hand-side**s of **rewriting-rule**s, whilst the former anywhere except inside **host-embedding** of host-directives (cf **The expand-directive**(3.1.3), **The host-directive**(3.1.4), **Performing Directives**(4.3.1)).

3.2.1 Comment Indicators

The comment indicators provide embedding of **aML-comment**s into aML texts. The **commit-opener** and **commit-terminator** together is a unary operation, resulting in

a 0-lengthed (i.e empty) string (cf **aML comments**(2.1.2), **The host-directive**(3.1.4), **Input**(4.1.1)).

3.2.2 Context Indicator

The **context-indicator** is the only indicator that may appear also in declarative cues. In declarative role they are used to declare **statement-class**es.

In operational cues, **context-indicator**s appeare in **incomplete-embedment**s or may appeare in **long-line**s, indicating a need of a completion process from the sides where **context-indicator**s stand. Expounding macro invocations, the left-hand sided embedding contexts can not be always staticly defined, but always computable (cf **Lines**(2.4.1), **Expounding Macro Embedding**(4.3.3)).

3.2.3 Link Indicator

The link indicator is a relational algebraic extension of the concatenation. Its operands may be **macro-parameter-reference**s of **reference-type vector-reference** (vectors) and **ordinary-text**s (scalars). Scalars are also considered as zero-dimensional vectors.

Recall that vectors are relational algebraic relations. The link indicator implements a projection to the vector elements in the relational agebraic outer join between vectors occasionally of different dimensons and different number of elements in different dimensions. The join condition is a partial equality between vector element indexes. The join is controlled by the at least one dimensional vectors from left to right.The partial index comparision takes place within a comparision range which is the smaller dimension of vectors of different dimensions. Index tuple items are compared for equality from the outermost dimension towards the innermost within the comparision range.

The order of the resulted element join is determined by the **element-ordering**s applied in the all-time leftmost non-null dimensional vector. The resulted element join is a mere line, iff **element-ordering**s in each dimension of each vector are **row-ordering** (cf **Macro Parameter Declaration and Reference**(2.2.1), **Macro Invocations**(2.5.2)).

Example

The example shows a statement and a name class declaration, a macro definition named Class, a macro invocation of macro Class, and the result outputted on to stdout. Registering the **macro-definition** in the macro library, the aML processor sends a warning on to stderr indicating the empty **left-shade-decl**.

The statement class defined is a **closed-context** statement class. Its left-hand sided border is `Class`, and the closing border is modelled as `};[ \t\n]?`. Its easy to see that the

five-line macro invocation is an element of the **closed-context** statement class declared.

The name class declared consists of the only name `Class`. The only anonym rule of the **macro-definition** places a **right-shade-decl separator** as `[ \t\n]*{[ \t\n]*`, and contains two **macro-parameter-declaration**s, one with name `v` and another one with name `p`. The latter is only for capturing the closer symbol ';' of the macro invocation lest that be part of the right-hand sided invocation context, which is finaly linked by the right-hand sided **context-inidicator** to the lines `<compound-object>`, `<compound name="f" type="float"/>`, and `</compound-object>`.

```
#statement{"Class"..."};[ \t\n]?"}
#embedded{"Class"}
#macro [Class]
{
        #invoked as "&[ \t\n]*{[ \t\n]*$v[{%X('[ \t]*')''".
        "%Y('int|float')' '%Z[('[a-f]')',|';']}".
        "'\n'|'([ \n\t]*})']$$p(';')$"
        {
                ...<compound-object>...
                ...{*   <compound name="$v[%Z[=]=]$" type="$v[%Y=]$"/>*}...
                ...</compound-object>...
        }
}
(W) Empty string (sub-)match may cause undesired pattern match
        line 11 in segment STDIN
Class
{
        int a,b,c,d;
        float e,f;
};
<compound-object>
   <compound name="a" type="int"/>
   <compound name="b" type="int"/>
   <compound name="c" type="int"/>
   <compound name="d" type="int"/>
   <compound name="e" type="float"/>
   <compound name="f" type="float"/>
</compound-object>
```

Parameter `v` is a one-dimensional vector with record elements. The **delimitation** of its elements is `\n` and its **vector-termination** is `([ \n\t]*})`. Tags of the record named `%X`, `%Y` and `%Z` are of **reference-type atom**, **atom** and **vector**, respectively, the elements

of the latter are of **reference-type atom**. The record tag named `%Z` is a one-dimensional-vector whose element **delimitation** and **vector-termination** are comma(`,`) and semi-colon (`;`), respectively.

When the macro invocation matches the invocation pattern, declared parameters map their values. Elements of vector parameters together with their index tuples are dynamically created and mapped. The value and element index mapping is controlled by ordered pairs of (`value`, **delimitaion**) and (`value`, **vector-termination**).

According to the above, vector parameter `v` has two elements: the value of v[0] is `\tint a,b,c,d;` and the value of v[1] is `\tfloat e,f;`. Since elements of vector `v` are records, and taking into account that **parameter-tag** named `%Z` is a one-dimensional vector, the value mapping distributed over the parameter-tags are as follows:

```
v[%X0] maps the value \t
v[%Y0] maps the value int
v[%Z[0]0] maps the value a
v[%Z[1]0] maps the value b
v[%Z[2]0] maps the value c
v[%Z[3]0] maps the value d
v[%X1] maps the value \t
v[%Y1] maps the value float
v[%Z[0]1] maps the value e
v[%Z[1]1] maps the value f
```

Apart from scalars the two-dimensional `v[%Z[=]=]` vector and the one-dimensional `v[%Y=]` vector are linked. The comparision range is 1. The outermost index of element values `a`, `b`, `c`, `d` in vector `v[%Z[=]=]`, and of element value `int` in vector `v[%Y=]` is 0. Simlarly, the outermost index of element values `e`, `f` in vector `v[%Z[=]=]`, and of element value `float` in vector `v[%Y=]` is 1. Thus, the result of the linking looks as:

```
<compound name="a" type="int"/>
<compound name="b" type="int"/>
<compound name="c" type="int"/>
<compound name="d" type="int"/>
<compound name="e" type="float"/>
<compound name="f" type="float"/>
```

Chapter 4

Process of aML Embedding

4.1 The aML Macro Processor

The aML macro processor is controlled by a dinamically changable processing environment. Processing a host language text that embeds aML directives and macro invocations, the macro processor may generate a sequence of output files in the all-time operating system environment producing a pure text, i.e a host language text that may contain aML directives and macro invocations according to the all-time processing environment of the macro processor.

The initially empty processing environment consists of the only augmentative dictionaries for regular expression functions, name and statement classes, and the augmentative, but also diminishable and alterable macro library. The all-time state of this processing environment is determined by host language text embedding being processed (cf **The embedded-directive**(3.1.1), **The macro-directive**(3.1.7), **The off-directive**(3.1.8), **The on-directive**(3.1.9), **The statement-directive**(3.1.10), **The term-directive**(3.1.13)).

4.1.1 Input

Host language text embedding shall be implemented as **embedding-data-stream**s nested in arbitrary depth. There is a LIFO maintained by the input handler of the macro processor to administer nested **embedding-data-stream**s, containing the one actually processed on the top. Operations push and pop are established by load-directive and the end of file symbol of the **embedding-data-stream** being processed.

The all-time host language text embedding is considered as a context-free sequence of host language statements. Host language statements are **outer-embedding**, and perhaps **inner-embedding**, the latter inside directive bodies. Between **outer-embedding** and **inner-embedding** there is a pretty strong analogy. What for **inner-embedding**

are **long-line**s, for **outer-embedding** are **declared-statement-class-statement**s of **statement-classes**. Similarly, **inner-single-line**s for **inner-embedding** correspond to **outer-single-line**s for **outer-embedding**.

Recognition of statements by matching statement classes is governed by a precedence rule. On the top of the prcedence hiararchy **directive-line**s stand, on the bottom of precedence hierarchy **host-embedment**s and **ordinary-inner-embedment**s. Between the two statement classes precedence of **declared-statement-class**es are splitted according to the order of their declarations (cf **Statement Classes and Statements**(2.4), **The statement-directive**(3.1.10)).

In case of **closed-context** statement classes, whenever a left-hand sided **border** detected the input handler keeps on reading in until the corresponding right-hand sided **border**, or a higher precedenced right-hand sided **border**, if any, is matched. Similarly, in case of **right-context** statement classes reading in is kept on until one of the right-hand sided **border**s of **right-context** statement classes, or a higher precedenced statement class, if any, is matched. In case of **left-context** statements, the end of the statement is determined by matching the next **left-context**, or **closed-context** statement or a **directive-line**-like statement[1].

The input handler recognizes comment indicators and escaped symbols. Whereas the **aML-comment**s are filtered out, backslashes are kept.

4.1.2 Output

Processing a host language text embedding, the aML macro processor generates a pure host language text. A pure host language text is a text that may contain unperformed directives, unexpounded macro invocations, and regular expression functions occasionally recognizable in the processing environment actually obtained. Escapes are interpreted according to the recently performed escape-directive or the default **escape-setting**. The generated pure host language text appeares as a sequence of files in the operating system environment. Any output file created and closed by target-directive can be treated as an **embedding-data-stream** by applying the load-directive (cf **The load-directive**(3.1.6), **The target-directive**(3.1.12), **Recognizing and Handling Regular Expression Functions**(4.2.2)).

[1] A **directive-line**-like statement is a line which apart from leading **blank**s and **horisontal-tab**s starts with a **directive-symbol**

4.2 Recognition of aML Embedding

4.2.1 Recognizing and Handling aML Comments

Comment indicators outside **host-embedding** of host-directives are recognized, and filtered out by the input handler (cf **Input**(4.1.1), **The host-directive**(3.1.4), **aML Comments**(2.1.2)).

4.2.2 Recognizing and Handling Regular Expression Functions

Unless regular expression functions are outside left-hand-sides of **rewriting-rule**s, their recognition takes place by matching the **function-symbol** (@).

Regular expression functions are symbolic references to parametrized terms. Their expansions (replacing them by their definitions) takes place recursively. During the definition replacement, **parameter-reference**s are replaced by the corresponding elements of the **value-list** cf **Regular Expresion Functions**(2.2.2, **Parameter Evaluation**(2.2.3), **The term-directive**(3.1.13)).

If the number of elements of a **value-list** is equal to the number of elements of the **parameter-list**, **value**s and **parameter-name**s of the same position within their own lists are assigned to each other (cf **Regular Expresion Functions**(2.2.2), **The term-directive**(3.1.13)).

If the **value-list** is shorter, than the **parameter-list**, **value**less parameters obtain the empty (0-lengthed) character string. A warning diagnostic message is also generated. In reverse case **value**s beyond the expected ones, are ignored in an accompaniment of a warning diagnostic message.

4.2.3 Recognizing and Handling Directives

The recognition of directives takes place by matching a **directive-line** statement. Recognized aML dierectives are brought into effect, by invoking associated directive interpreters. Each directive has its own interpreter controlled by the associated sub-grammar (cf **Lines**(2.4.1), **Directives**(3.1)).

4.2.4 Recognizing and Handling Macro Invocations

A host language statement may contain an arbitrary number of **macro-invocation**s. Some of them may be cought partly or as a whole by others as parameters, and therefore notions visibility and ken play an essentially important role in recognizing them. A **macro-invocation** is said being recognisable, if its definition contains a visible **rewriting-rule** the **left-hand-side** of which can be matched by it.

The visibility is a property of declared **rewriting-rule**s. The ken is the visible range within the statement from the position of a recognisable **macro-name** in both direction in which its **left-shade** and **right-shade** can be matched at all by the **left-shade-decl** and **right-shade-decl** declared in one of its visible **rewriting-rule**s.

Visibility

The visibility of declared **rewriting-rule**s may be obstructed only by the off-directive.

Ken

The ken of the leftmost recognisable **macro-name** whithin the statement is the statement boundary in both directions, inclusive.

The ken of the rightmost or an intermediate recognisable **macro-name** leftwards is determined by the closest preceding already matched **macro-invocation**, or in lack of that the left-hand sided statement boundary, and rightwards is the statement boundary, inclusive.

For neighbouring **macro-invocations**, the **left-shade** of its right-hand sided neighbour must not overlap the macro parameter values adopted by the **right-shade** of the lef-hand sided neighbour, i.e the ken of the right-hand sided neighbour leftwards is the macro parameter value free tail of the **right-shade** of the left-hand sided neighbour, inclusive.

Due to the above, a **macro-invocation** can be and is, in fact, detected, if

(1) the **macro-name** is recognisable, and its **macro-definition** contains at least one visible **rewriting-rule** (cf **The macro-directive**(3.1.7), **The off-directive**(3.1.8), **The on-directive**(3.1.9, **The statement-directive**(3.1.10), **Visibility**(4.2.4));

(2) there is a visible **rewriting-rule** in the **macro-definition** named by **macro-name**, the **left-hand-side** of which yields a **macro-symbol**-based match within the ken of the **macro-name** in question (cf **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7), **Ken**(4.2.4))).

Condition (1) is the requirement to explore those locations within statements where **macro-invocation**s are worth looking for. Hence an ordered set of what are called **name-class** search patterns is established, in order to locate the next leftmost place in the statement where a **macro-invocation** candidate resides. A **name-class** search pattern is the narrowest special closure of the **name-class** which matches any element belongs to the **name-class**, but steps over anything else. The ordering of **name-class** search patterns is determined by the order of the declarations of **name-class**es. Matching the set of **name-class** search patterns results in a list of all elements found from **name-class**es. If the list is empty, the statement is **macro-invocation** free (cf **The embedded-directive**(3.1.1), **Expounding Macro Embedding**(4.3.3)).

Condition (2) postulates that a **macro-symbol**-based match within the ken of the **macro-name** requires that the **left-shade-decl** and **right-shade-decl** shall match the **left-shade** and the **right-shade**, respectively. The match comparison is performed from the **macro-symbol** in both directions towards the statement boundaries. If the **left-shade-decl** or **right-shade-decl** or both contain **macro-parameter-declaration**s, values from **left-shade** and **right-shade** are assigned to them (cf **Macro Parameter Declaration and Refrence**(2.2.1), **Parameter Evaluation**(2.2.3), **Expounding Macro Embedding**(4.3.3)).

The following examples illustrates the above rules.

Example 1

```
#embedded{"A"}
#macro [A]
{
        #invoked as "$p('abc')$&$q('abc')$~"
        {
                ...p($p$)-q($q$)...
        }
}
abcAabcAabcAabcA
p(abc)-q(abc)Ap(abc)-q(abc)A
```

Example 1 defines the macro named `A`. The macro definition contains only one anonym rewriting rule, where `A` declares macro parameters named `p` left to, and `q` right from the **macro name**. The latter is followed by a **card**. The definition is followed by an **outer-single-line** carrying name `A` four times, and finally the substituded result of macro invocations on the standard output.

The leftmost matched invocation is constituted by the leftmost occurence of `A`. Its both declared macro parameters are assigned to the value `abc`. The invocation results in a **left-embedment**, namely `p(abc)-q(abc)...`. To complete the **left-embedment**, the macro processor looks for the next (leftmost) invocation in the **outer-single-line**. The next invocation is composed by the 3^d occurence of `A`. Its ken both leftward and rightward is started with letter `A`, namely, leftward the 2^{nd} and rightward the rightmost. Expounding the invocation produce the **embedment** `p(abc)-q(abc)Ap(abc)-q(abc)A` which is in turn identical with the result on stdout.

Example 2

```
#term{p() = "[A-Za-z][A-Za-z0-9]*"}
```

```
#embedded{"$|A"}
#macro [$]
{
        #invoked as "&"
        {
                ...Caught...
        }
}
#macro [A]
{
        #invoked as "&$p('@p()')$"
        {
                ...p($p$)...
        }
}
        A$
        p(Caught)
```

The example presents a term declaration, a name class declaration, two macro definitions, furthermore, a statement consisting of invocation of macros `A` and `$`, and finally the result of the statement.

Due to invocation contexts declared by invoke-directives, invocation of macro named `A` is unrecognisable, since its invocation context expects a character string as parameter value from the right-hand side which shall start with a letter and may contain letters an figures.

The invocation of macro named `$` is, however, recognised and expounded: its invocation context is the empty symbol from both sides.

The invocation of `$` results in the **middle-embedment**

```
...Caught...
```

which is completed from both sides by the context indicator resulting in the line

```
ACaught
```

Since macro expansions are recursive processes, the line resulted shall be again checked whether it contains macro invocations. Invocation A is now already recognised and even matched. Its expansion results in the line

```
p(Caught)
```

Example 3

```
#embedded{"A|$"}
#macro [$]
{
        #invoked as "&"
        {
                ...Caught...
        }
}
#macro [A]
{
        #invoked as "&$p('\$')$"
        {
                ...p($p$)...
        }
}
        A$
        p(Caught)
```

Example 3 produces the same result as Example 2. The main difference between them is the co-domain declaration of macro parameter `p`.

According to the declaration of macro parameter `p` the leftmost, otherwise the only recognized macro invocation is now `A`. The expansion results in the **middle-embedment**

```
...p($)....
```

The complementary context is \t from the left-hand side and \n from the right-hand side. After completion the resulted line is

```
p($)
```

Now macro invocation `$` in the above line is recognizable. Its expansion results in the **middle-embendent**

```
...Caught...,
```

which after completion produce looks as

```
p(Caught)
```

i.e produce the same output on stdout as did Example 2.

4.3 Processing aML Embedding

Processing of aML embeddigns means that statements read in are filtered out from comments and scanned through for directives and macro invocations. Directives are executed, and macro invocations are expounded. Statements different from the above are outputted on the all-time target data stream (cf **Input**(4.1.1), **Output**(4.1.2), **Recognizing and Handling Directives**(4.2.3), **Recognizing and Handling Macro Invocations**(4.2.4)).

Directives with the exception of expand-directive and host-directive augment or alter the aML processing environment. Indicators operate on the embedding context . Macro embedding, either independently of or dependently on, and eithter together with or without their context will be replaced by the associated **right-hand-side**s. Replacements are controlled by particular expansion rules, and may take place either independetly of or dependently on the invocation context, and also independently of or in interaction with each other (cf **Embedding and Embedding Contexts**(2.5), **Macro Invocations**(2.5.2), **Directive Embedding**(2.5.1), **Directives**(3.1), **Indicators**(3.2)).

Execution of directives takes palce in one step. Performing directives apart from the above exceptions produce no statements to be further processed. The expansion of macro embedding can, however, start a recursive process: statements on the right-hand sides of rewriting rules applied may embed directives, indicators and macro invocations to be further processed.

4.3.1 Performing Directives

For each aML directive, the macro processor maintains an interpreter that parse and execute the directive according to its sub-grammar and semantics. Each directive subgrammar is common in the sense that any directive starts with a directive name prefixed by the **directive-symbol** (`#`), and ends with the directive body delimited by the pair `{}`. Also a common syntactic rule for directives that the **directive-symbol** may only be preceded, and the symbol closing the derective body (}) may only be followed by spaces, tabs, or ENTER (cf **Directives**(3.1)).

The embedded-directive and statement-directive augment the dictionaries of name classes and statement classes, respectively (cf **Declared Statement Classes**(2.4.2), **The embedded-directive**(3.1.1), **The statement-directive**(3.1.10)).

The expand-directive may only be embedded by **right-hand-side**s of **rewriting rule**s in macro definitions (cf **Embedding and Embedding Contexts**(2.5), **The expand-directive**(3.1.3), **The invoked-directive**(3.1.5)).

The host-directive prohibits executon of directives, indicators and also expansion of macro invocations. Statements inside the **host-embedding** are outputted without parsing and

processing them. The target data stream is the **right-hand-side** of the **rule-declaration** whenever the directive is performed by an expand-directive, otherwise the all-time target data stream created and opened or just switched by the last time performed target-directive – the standard output by default (cf **The expand-directive**(3.1.3), **The host-directive**(3.1.4), **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7)).

The invoked-directive defines **rewriting-rule**s for macro definitions. Unless the **directive-body** carries expand-directives, **rewriting-rule**s are written without processing onto its associated macro definition in the macro library (cf **The expand-directive**(3.1.3), **The host-directive**(3.1.4), **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7)).

The macro-directive augments the macro library, or alters an already declared macro definition (cf **The expand-directive**(3.1.3), **The host-directive**(3.1.4), **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7)).

The off-directive and on-directive alter visibility of **rewriting-rule**s in the macro library. (cf **The invoked-directive**(3.1.5), **The macro-directive**(3.1.7), **The off-directive**(3.1.9), **The on-directive**(3.1.9), **Visibility**(4.2.4)).

The load-directive and target-directive change the all-time input and output data stream of the macro processor (cf **The load-directive**(3.1.6), **The target-directive**(3.1.12), **Input**(4.1.1), **Output**(4.1.2)).

The statement-directive augments the statement class dictionary (cf **Declared Statement Classes**(2.4.2), **The statement-directive**(3.1.10), **Input**(4.1.1)).

The term-directive augments the dictionary of regular expression terms (cf **Regular expresion functions**(2.2.2), **The term-directive**(3.1.13)).

4.3.2 Statement Preservation Principle

The statement preservaiton principle play a major role in expounding such macro invocations where **right-hand-side**s of **rewriting-rule**s contain **incomplete-embedment**s with **link-indicator**s inside[2]. It states that an **ordinary-inner-embedment** remains one statement after its **macro-parameter-reference**s, if any, are replaced by their matched values, even if the replacement results in more then one line. It is obvous, that according to the principle, **macro-parameter-reference**s with **vector-reference** as **reference-type** result in **long-line**s.

The principle allows the aML macro processor to map matched values and indexes for **macro-parameter-reference**s of **reference-type vector-reference**, and to perform **link-indicator**s immediatelly as soon as the macro invocation matched.

[2]The precedence of the **context-indicator** is higher than that of the **link-indicator.**

4.3.3 Expounding Macro Embedding

The expansion of macro invocations is a complex rewriting procedure carried out recursively. Here only statements from **statement-class**es different from **directive-line** are considered. Such statements are **ordinary-inner-embedment**s, **host-embedment**s, **declared-statement-class-statement**s, and **long-line**s.

Reading of **inner-single-line**s in from **right-hand-side**s of **rewriting-rule**s there are no **macro-parameter-reference**s for declared macro parameters, and no link-indicators have to be delt with. All value substitutions of all **macro-parameter-reference**s, and also performing link-indicators takes place immediately, when macro invocatios are matched (cf **Statement Preservation Principle**(4.3.2)).

Parsing **long-line**s and **ordinary-inner-embedment**s, the macro processor looks for **context-indicator**s and macro invocations. Completion of **incomplete-embedment**s precedes expansion of macro invocations included by them.

Parsing **declared-statement-class-statement**s and **host-embedment**s, only macro invocations are looked for (cf **Lines**(2.4.1), **Macro Invocations**(2.5.2), **Declared Satement Classes**(3.1.10), **Indicators**(3.2)).

Macro invocations carried by one statement are expounded strictly from left to right. Statements free from **context-indicator**s and macro invocations are outputted. The expansion of a statement starts always the expansion of the first matched, namely the leftmost macro invocation of the statement (cf **Macro Invocations**(2.5.2), **Input**(4.1.1), **Output**(4.1.2), **Recognizing and Handling Macro Invocations**(4.2.4), **Statement Preservation Principle**(4.3.2)).

Expansion of a macro invocation takes place by reading of statements in one by one from the **right-hand-side** of the **rewriting-rule**. All **incomplete-embedment**s and also **long-line**s in **right-hand-sides** of **rewriting-rule**s that contain **context-indicator**s are indexed separately according to the sides of incompleteness. A **middle-embedment** e.g. may be furnished with index n, and m from the left-hand and from the right-hand side, respectively, where n and m may be different non-negative whole numbers. Reading of an **incomplete-embedment** in starts a context preseving expansion process. A context preserving expansion process follows the below rules and constraints:

1. Statements which are free from **context-indicator**s and macro invocations are outputted.

2. For the leftmost invocation, an incomplete statements from the left is completed from the left-hand-side with the initial fragment of statements. The initial fragment starts with the first symbol of the statement and lasts till the leftmost symbol of the leftmost macro parameter value to the left from the macro name, exclusive, if any. Lacking left-hand-sided macro parameter values the initial fragment lasts till

the **macro-name**. The initial fragment might also be the empty (i.e 0-lengthed) character string.

3. For the rightmost invocation, an incomplete statements from the right is completed from the right-hand-side with the closing fragment of the statement. The closing fragment starts with the first symbol follows the rightmost symbol of the rightmost macro parameter value to the right from the macro name, if any, and lasts till the last symbol of the statement. Lacking right-hand-sided macro parameter values the closing fragment starts with the symbol immediately following the **macro-name**. The closing fragment might also be the empty (i.e 0-lengthed) character string.

4. For the leftmost or any intermediate macro invocation, an incomplete statement from the right indexed by j joins the incomplete statement from the left indexed by j of the right-hand sided neighbour together with the **separator** between them. Until the desired statement of the right-hand sided neighbour reached, all statements read in are processed according to 1 - 5. Lacking the statement in question result in an expansion error. A completed incomplete embedment constitutes one statement that may belong to any default or declared statement class, or a sequence of embedments[3] which is then imediatelly expounded according to 1 - 5.

5. Whenever the expansion of a macro invocation ends, the context preserving process goes on with expounding the right-hand sided neighbour according to 1 - 5 if that at all exists. The right-hand sided neighbour must not contain subsequent incomplete statements from the left, otherwise an expansion error comes about.

[3]It occures when the context preserving expansion process touches **long-line**s.

Bibliography

[1] Hernáth Zsolt, Bauer Péter: *aML – a Macro Language annotált formális definíció* https://plc.inf.elte.hu/szomin_odf/repos/doc/aML/definition_hu/aml.pdf

[2] Zsolt G. Hernáth, Péter Bauer: *aML – a Macro Language annotated formal definition* https://plc.inf.elte.hu/szomin_odf/repos/doc/aML/definition_en/aml.pdf

[3] Kernighan, Brian W.: *RATFOR — A Preprocessor for a Rational Fortran*, Software—Practice and Experience, vol. 5., pp. 395-406., October 1975, `https://doi.org/10.1002/spe.4380050408`

[4] Nagata, Hiroyasu: *FORMAL: A language with a macro-oriented extension facility*, Computer Languages Volume 5, Issue 2, pp. 65-76, 1980, `https://doi.org/10.1016/0096-0551(80)90048-X`

[5] Hernáth Zsolt: *FORMAL: nyelvi környezettől független magas szintű makronyelv*, Információ elektronika 1984/3. 134.-143. oldal

[6] Hernáth, Zsolt: *FORMAL: a high-level and extensible macro-language for extending several basic languages*, 4th Hungarian Computer Science Conference Abstracts, Győr 1985. Conference abstract, pp 40

[7] Wikipedia: *M4 (computer language)*, http://en.wikipedia.org/wiki/M4_(computer_language)

[8] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation 26 November 2008, http://www.w3.org/TR/REC-xml/

[9] *Extensible Markup Language (XML) 1.1 (Second Edition)*, W3C Recommendation 16 August 2006, edited in place 29 September 2006, http://www.w3.org/TR/xml11/