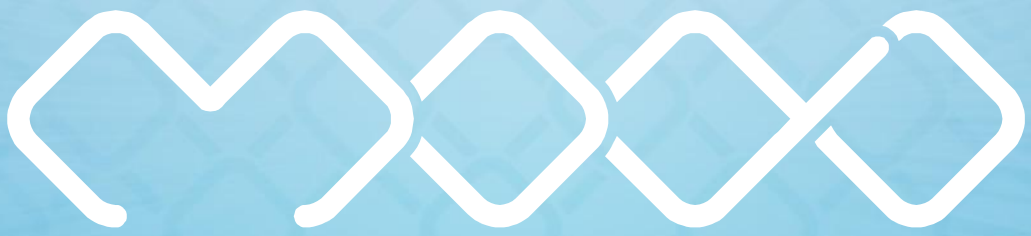
sanpark



2009

Modo Language Specification

specification

Sanpark IT and Design Co.,Ltd.
Tunalı Hilmi Cad. 50 / 18 06660 Kavaklidere / Ankara / TURKEY
Tel: +90 312 425 27 77 Fax: +90 312 425 27 78
info@sanpark.com
www.sanpark.com

June 2009

Table of contents

1. Introduction

By reducing the time and cost of the solutions of real life problems, information technologies had a deep impact on the every little aspect of both our professional and daily lives. Especially with the urge of the internet era, this impact is drastically increasing as the solutions now became time-free and location-free (Park and Lim, 1999; Calvary et al., 2003; Rau et al., 2004; Lam and Swayne, 2001; Kim 2001).

One of the largest challenges of the IT field is the gap between the real world systems and their representations in the computer, or to be more specific today's online world. Numerous professionals proposed notations, languages and methods to provide solutions to narrow this gap (Kernighan and Ritchie, 1978; Wirth, 1971; Gosling et al., 2005; Tom, 2001; Booch et al., 2000; Heng and Mackie, 2009; Laporti et al., 2009; Horsburgh et al., 2009; Reinhartz-Berger and Sturm, 2009; Schwabe and Rossi, 1998). Most of those solutions are being used by many of the IT and related professionals today.

However, with the increasing shift of systems to the online world there still resides a need to have a representation tool to model and convert real life systems to the world of computers.

Considering the online world of today and the predicted integrated and semantic, structure of tomorrow, one can easily say that this new tool

- should allow faster modeling,
- should be easy to use,
- should be able to represent every element of the real life systems,
- should be able to represent the relations between the elements of the real life systems,
- should easily be used for modeling different systems,
- should be able to produce a representation which can be interpreted not only by humans but also by machines,
- should be able to produce a representation which can easily be converted to the existing or the future technologies.

Considering the above reasons and more, we propose the Modo language.

1.1. About the document

This document contains the specification of the Modo language. In this first chapter, Introduction, the acknowledgement and the general information about the document is mentioned.

The rest of the document contains the following chapters:

Motivation chapter covers the underlying reasons, which led us to develop Modo.

Definitions chapter covers the technical terminology and jargon that we find compulsory to make the most use of this document.

Structure and Syntax chapter gives detailed information about the syntax of the language using Extended Backus-Naur Form, Extended BNF (ISO/IEC 14977, 1996). Along with the syntax, lexical sequence, constraints and rules are also described.

Elements chapter covers the element declarations and the uses of the elements in Modo definitions.

Selectors chapter introduces selectors conceptually. Additionally selector syntax and the uses of the selectors are described in this chapter.

Expressions chapter gives information about the expressions for operators, constants, strings and elements.

1.2. Scope

This document specifies/contains structure and syntax of the Modo language.

This document does not specify/contain

- use cases
- system dependent elements/methods
- technology dependent elements/methods
- conversions and interpretations of Modo to other technologies

1.3. Conformance

This document is intended for developers, designers, system designers and modelers and other related professionals whose activities contain modeling of real life systems for computer technologies.

For better understanding of this document, the audience must have experience in the following:

- Object oriented system modeling
- Computer programming languages
- EBNF notation (ISO/IEC 14977, 1996)

1.4. Credits and Acknowledgement

The authors thank Sanpark IT & Design Co. Ltd., for maintaining the professional environment and encouragement that resulted and catalyzed the development of Modo. They also thank Recep Kütük from Sanpark for the Modo identity and the document formatting.

Modo language is shaped by the ideas, practice and experience of Aykut Aydınlı and the Sanpark crew; starting from 2005. The latest version of Modo is shaped by Aykut Aydınlı and Doruk Eker.

This document is authored and edited by Aykut Aydınlı and Doruk Eker.

2. Motivation

The basic idea that initiated the effort for the development of Modo was to create a modeling tool, which is able to model a real life system, its elements and the relations between them.

Modo focuses on the system to be modeled, as it exists. The technological boundaries and constraints, which are only obstacles of the computer systems, are neglected during the modeling process. The model then can be converted to the required technologies. This conversion can be designed to catalyze any software development process, reducing the cycle time of development/maintenance and increasing the quality of the software.

Moreover, Modo is aimed to be a contribution for the ongoing studies for the Semantic Web, which is considered to be the next era of online information delivery (Berners-Lee et al., 2001). Modo models represent entities of real life systems and the relations between them. Thus the system is modeled at the semantic level, and more importantly this model is completely machine-readable. With the help of this property, Modo can act as a common definition language at semantic level and help different systems (online/offline) to be integrated.

All those properties of the language led us to show the effort to develop Modo.

3. Definitions

This document contains technical details about Modo language. These technical details necessitate the use of technical terminology. This chapter gives the definitions of some of these technical terms for making the most use of this document.

System is the real life system that is modeled using Modo.

Element is the single member of a system.

Member Element is the sub element of an element.

Modo Generator/Compiler/Interpreter is the application that can compile and convert the Modo declarations into different technologies or formats.

Syntax is a piece of code that is sequenced together to form a meaningful Modo sentences.

Grammar is the set of rules that are used to code Modo declarations.

4. Structure and syntax

This chapter describes the basic structure of the Modo language. Additionally Modo syntax is introduced in this chapter.

4.1. Syntax

This section defines the Modo syntax. Besides some fundamental differences, Modo syntax shows similarities to Cascading Style Sheets (CSS) language specified in Bos et al. (Bos et al., 2009). The reasons for these similarities are 1) CSS provides great ease for the definition of the elements and their properties and 2) common use of CSS language among different disciplines, especially designers and developers.

Modo syntax is defined using Extended BNF metalanguage specified in ISO/IEC 14977 (ISO/IEC 14977, 1996). By using the Extended BNF metalanguage, syntactic sequence of the Modo is separated into lexical units. This helps to identify every lexical unit in a meaningful Modo sentence.

These definitions include some of the main syntactic parts of the Modo language. For the complete list of definitions, please refer to Appendix A. Grammar.

Following example shows the declaration of an element in Modo syntax:

```
1  /* Company Element Declaration */
2  Company {
3      Name = String;
4      $Name = "Doe Software Inc.";
5      Founder = String;
6      $Founder = "John Doe";
7      TaxNumber = String;
8      $TaxNumber = "123456789987654321";
9  }
```

The example above shows the declaration of `Company` element. On line 1, a descriptive comment is added to the declaration. On line 2, the declaration of the `Company` element starts.

On lines 3, 5 and 7 the member elements, `Name`, `Founder` and `TaxNumber` of the `Company` element are declared respectively. On lines 4, 6 and 8 initial values of those member elements are assigned. Please note that the `dollar symbol "$"` preceding the member element name refers to the value of that member element.

Finally on line 9, `Company` element declaration ends with `end declaration symbol "}"`.

Note: The above example assumes that the element `String` has already been defined.

4.1.1. Tokenization

For better understanding, Modo syntax is separated into tokens at lexical level. A Modo declaration consists of a sequence of these tokens.

Following is the general Extended BNF notation of the Modo:

```
modo = {
    directive
    | rule
    | declaration
    | comment
};
```

As you can see above, Modo definition consists of directives, rules, declarations and comments. Followings are the Extended BNF notations of each token:

```
(* Directive Definition *)
directive = {blank},
        at symbol,
        identifier,
        {blank}-,
        expression,
        {blank},
        end sentence symbol;

(* Rule Definition *)
rule = {blank},
        at symbol,
        {blank},
        start group symbol,
        {blank},
        expression,
        {blank},
        end group symbol,
        {blank},
        start declaration symbol,
        {blank},
        modo,
        {blank},
        end declaration symbol;
```

```
(* Comment Definition *)
comment = delimited comment
        | single line comment;

(* Declaration Definition *)
declaration = {
        assignment
        | basic declaration
        | extended declaration
};
```

Characters (The Unicode Standard, 2003), letters, numerals, symbols and operators used in Modo syntax are defined as follows:

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
        | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P"
        | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
        | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f"
        | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
        | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
        | "w" | "x" | "y" | "z";

character = any Unicode character;

whitespace = {
    space character
    | tab character
    | line feed character
    | carriage return character
    | form feed character
};

new line = carriage return character
        | line feed character
        | next line character
        | line separator character
        | paragraph separator character;

space character = " ";

tab character =
        ? The Unicode Standard Tab Character (U+0009) ? ;

line feed character =
        ? The Unicode Standard Line Feed Character (U+000A) ? ;

carriage return character =
        ? The Unicode Standard
        Carriage Return Character (U+000D) ? ;

form feed character =
        ? The Unicode Standard Form Feed Character (U+000C) ? ;

next line character =
        ? The Unicode Standard Next Line Character (U+0085) ? ;

line separator character =
        ? The Unicode Standard
        Line Separator Character (U+2028) ? ;
```

```
paragraph separator character =
        ? The Unicode Standard
        Paragraph Separator Character (U+2029) ? ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
        | "8" | "9";

at symbol = "@";

assignment symbol = "=";

end sentence symbol = ";";

start group symbol = "(";

end group symbol = ")";

start declaration symbol = "{";

end declaration symbol = "}";

backslash symbol = "\";

start attribute selector symbol = "[";

end attribute selector symbol = "]";

start delimited comment symbol = "/*";

end delimited comment symbol = "*/";

start single line comment symbol = "//";

dot symbol = ".";

comma symbol = ",";

underscore symbol = "_";

hyphens symbol = "-";

escaped single quote symbol = "\'";

escaped double quote symbol = '\"';

single quote symbol = "'";

double quote symbol = '"';

increment operator = "++";

decrement operator = "--";

unary operator = "+" | "-" | "!" | "~"
        | increment operator
        | decrement operator;
```

```
operator = "*" | "/" | "%" | "+" | "-" | "<<" | ">>" | "<"
         | ">" | "<=" | ">=" | "==" | "!=" | "&" | "^" | "|"
         | "&&" | "||";

unary selector operator = "$", ".", "#", "?", " ";

selector operator = ">", "<", "+", "-", "~", "^", ":",
        "::", "%", "&", "|", "\", "/";

selector expression operator = "*" | "/" | "%" | "+" | "-"
         | "<<" | ">>" | "<" | ">" | "=" | "&" | "^" | "|"
         | "<=" | ">=" | "==" | "!=" | "*=" | "/=" | "%="
         | "+=" | "-=" | "&=" | "^=" | "|=" | "$=" | "~="
         | ".=" | "#=" | "?=" | ":=" | "&&" | "||";

universal selector = "*";
```

Additionally there are some extra tokens used in Modo syntax. For the sake of continuity, only main syntactic parts are shown here. Please refer to Appendix A. Grammar for complete list of tokens.

In Modo syntax, characters of any kind including `whitespace` are not allowed between syntactic parts.

Only the `space character`, `tab character`, `line feed character`, `carriage return character` and `form feed character` can occur in `whitespace`.

4.1.2. Characters and case

Modo syntax is case-insensitive. Elements and member elements declared in Modo can be considered as case-insensitive.

Following is the Extended BNF notation of `identifier` and `lhs identifier`:

```
identifier = {letter | underscore symbol}-,
         {
             digit
             | letter
             | underscore symbol
         };
```

The `identifier` definition above shows that the `identifier` starts with `letter` or `underscore symbol "_"` and can continue with `digit`, `letter` or `underscore symbol "_"`.

```
lhs identifier = {
             letter
             | underscore symbol
             | hyphens symbol
         }-,
         {
             digit
             | letter
             | underscore symbol
             | hyphens symbol
         };
```

The left-hand side selector (lhs) identifier shows that `lhs identifier` starts with either `letter`, `underscore symbol "_"` or `hyphens symbol "-"`.

```
/* Company Element Declaration */
Company {
    Name = String;
    $Name = "Doe\'s Software Inc.";
    Founder = String;
    $Founder = "John Doe";
    TaxNumber = String;
    $TaxNumber = "123456789987654321";
}
```

The example above shows the declaration of `Company` element. In this declaration `Company`, `Name`, `Founder` and `TaxNumber` are the left-hand side identifiers.

Additionally, there is an alternative use of `backslash symbol "\"` in strings. Backslash symbol is used to print single and double quote symbols in string values. Also for printing backslash symbol in string values, one must specify two backslash symbols `"\\"` consecutively.

Following is the Extended BNF notation of `escaped single quote symbol` and `escaped double quote symbol`:

```
escaped single quote symbol = "\'";

escaped double quote symbol = '\"';
```

In addition to that, `string`, `single quote escaped string` and `double quote escaped string` definitions are as follows:

```
string = (
            single quote symbol,
            single quote escaped string,
            single quote symbol
        )
        |
        (
            double quote symbol,
            double quote escaped string,
            double quote symbol
        );

single quote escaped string = {
        (any Unicode character - single quote symbol)
        | escaped single quote symbol
};

double quote escaped string = {
        (any Unicode character - double quote symbol)
        | escaped double quote symbol
};
```

4.1.3. Directives

Directives are the instructions directly related with the Modo generators/compilers/interpreters. They are not a part of the system declared in Modo. Directives start with `at symbol "@"` and continue with `identifier` and `expression`. Finally, directives end with `end sentence symbol ";"`.

```
(* Directive Definition *)
directive = {blank},
        at symbol,
        identifier,
        {blank}-,
        expression,
        {blank},
        end sentence symbol;
```

4.1.4. Rules

In Modo syntax, `Rules` play an important role of representing the different states of a system. In real life, the elements of a system might contain different properties or values for different states of the system. When modeling the system, those different states, values and properties must be included in the model. In Modo syntax, those different properties and values are reflected using the Rules.

Declarations, types and the values of the elements can be specified depending on rules. With the help of rules, values and properties of the system can be changed or set depending on one or more conditions. Rules start with `at symbol "@"`.

There are two types of rules: 1) Rule blocks and 2) inline rules. Following is the Extended BNF notation of rules (rule blocks) and inline rules:

```
rule = {blank},
        at symbol,
        {blank},
        start group symbol,
        {blank},
        expression,
        {blank},
        end group symbol,
        {blank},
        start declaration symbol,
        {blank},
        modo,
        {blank},
        end declaration symbol;
```

The `rule` definition above shows that `rule` starts with `at symbol "@"`, and continues with `start group symbol`, `expression` and `end group symbol`. After rule expression is specified, rule specific declaration starts with `start declaration symbol`. Any meaningful Modo declaration can be placed between declaration symbols. Finally, `rule` ends with `end declaration symbol`.

```
inline rule = {blank},
        at symbol,
        {blank},
        start group symbol,
        {blank},
        expression,
        {blank},
        end group symbol;
```

Similarly, the `inline rule` starts with `at symbol "@"` and continues with `start group symbol`, `expression` and `end group symbol`.

Following example shows the use of rules in Modo declarations:

```
1  Company {
2      Name = String;
3      $Name = "Doe Software Inc.";
4      $Name = "DOE Corp." @ ($CurrentYear > 2008);
5      Founder = String;
6      $Founder = "John Doe";
7      TaxNumber = String;
8      $TaxNumber = "123456789987654321";
9  }
```

The example above shows the use of inline rules. On line 2, the `Name` member element of the `Company` element is defined and on line 3, it is assigned an initial value. On line 4, another value is assigned to the `Name` element, but this time depending on the rule `($CurrentYear > 2008)`.

Imagine a scenario where the company's name was changed to `"DOE Corp."` in the year 2008. Therefore, once the value of the `CurrentYear` is greater than 2008 the value of the `Name` element is `"DOE Corp."`. In other cases, the value of the `Name` element is `"Doe Software Co.,Ltd."`.

```
1  Company {
2      Name = String;
3      Founder = String;
4      TaxNumber = String;
5      $Name = "Doe Software Inc.";
6      @ ($CurrentYear > 2008) {
7          $Name = "DOE Corp.";
8      }
9      $Founder = "John Doe";
10     $TaxNumber = "123456789987654321";
11 }
```

The example above shows the use of rule blocks. On line 6, rule block starts with `at symbol "@"`. On line 7, a new value is assigned to the `Name` element. Finally, rule block ends with `end declaration symbol` on line 8.

One possible implementation of `Rules` in Modo syntax is to define a value or property of elements that does not change in any state of the system.

For the previous scenario, assume that the tax number of the company is always the same. Therefore, value of the `TaxNumber` element never changes. The following example shows the Modo syntax, which defines an everlasting rule.

```
1  Company {
2      Name = String;
3      Founder = String;
4      TaxNumber = String;
5      $Name = "Doe Software Inc.";
6      $Founder = "John Doe";
7      $TaxNumber = "123456789987654321" @ (1);
8  }
```

In the above example, the elements are defined similarly to the previous example. However, when assigning values a rule is defined for the value of the `TaxNumber` element. On line 7, after the value is assigned to the `TaxNumber` element the rule `(1)` is defined. This rule indicates that, for all states of the system the value of the `TaxNumber` element is `"123456789987654321"`.

Note: The above examples assume that the member element `CurrentYear` and element `String` have already been declared.

4.1.5. Blocks

Blocks are used for element and rule declarations. In Modo syntax, blocks start with `start declaration symbol "{"` and end with `end declaration symbol "}"` (Appendix A. Grammar). Every `start declaration symbol` must end with `end declaration symbol`.

Similarly, in string values if a string starts with `single quote symbol`, it must end with `single quote symbol`. In addition, if a string starts with `double quote symbol`, it must end with `double quote symbol`. Every `expression` that starts with `start group symbol "("` likewise ends with `end group symbol ")"`.

4.1.6. Selectors

Modo syntax provides some pattern-matching rules for element and member element declarations. These pattern-matching rules are called “Selectors”.

Modo syntax introduces selectors conceptually. For further information, please refer to 6. Selectors.

4.1.7. Elements

Elements are the basic Modo structures that define real life entities. Elements can be declared by single line definitions or block declarations.

Following example shows the declaration of `Company` element.

```
1  /* Company Element Declaration */
2  Company {
3      Name = String;
4      $Name = "Doe Software Inc.";
5      Founder = String;
```

```
6       $Founder = "John Doe";
7       TaxNumber = String;
8       $TaxNumber = "12345678998765432l";
9   }
```

Elements can be declared by extending built-in or previously declared elements. Following example shows the declaration of `Worker` and `Developer` elements:

```
1   Worker {
2       FirstName = String;
3       LastName = String;
4       Department = String;
5   }
```

The example above shows the declaration of `Worker` element. On line 1, `"Worker"` identifier is specified and declaration of the `Worker` element starts with `start declaration symbol`. On line 2, `FirstName` member element is defined. On line 3, `LastName` member element is defined and on line 4, `Department` member element is defined. Finally, element declaration ends with `end declaration symbol`.

Below is the declaration of the `Developer` element, which is an extension of the `Worker` element.

```
1   Developer = Worker {
2       $Department = "Software Development";
3   }
```

The example above extends the `Worker` declaration. On line 1, `Developer` element is declared extending the `Worker` element. Additionally on line 2, a specific value is assigned to the `Department` element.

4.1.8. Declarations

There are three types of Modo element declarations:

- Assignments
- Basic Declarations
- Extended Declarations

Assignments are one-line declarations where the `assignment symbol "="` is used and values are assigned to the elements and member elements.

Basic declarations are the typical element declarations as blocks.

Extended declarations are the declarations that extend built-in or previously declared elements.

For further information about declarations, please refer to 5.1. Declarations.

4.1.9. Expressions

Modo syntax allows using expressions for assignments. Following example shows the use of expression in assignments:

```
1  Worker {
2      $FirstName = "John";
3      $LastName = "Doe";
4      $FullName = ($FirstName + " " + $LastName);
5  }
```

The example above shows the declaration of `Worker` element. On line 4, the value of the `FullName` member element is specified by an expression. According to this expression, value of the `FullName` member element is composed of the value of the `FirstName` member element, the `space character` `" "` and the value of the `LastName` member element.

Modo declarations specify the state of the system for the given time periods. This nature of the Modo necessitates that the assignments must be valid for the given time periods. Therefore, for the above example, when the `FirstName` attribute value changes, `FullName` attribute value must be recalculated and updated by the generators/compilers/interpreters using this Modo model.

For further information, please refer to 7.Expressions chapter.

4.1.10. Comments

There are two types of comments:

- Delimited Comments
- Single Line Comments

```
comment = delimited comment
        | single line comment;
```

Delimited comments start with `start delimited comment symbol` `"/*"` and end with `end delimited comment symbol` `"*/"`.

```
delimited comment = {blank},
        start delimited comment symbol,
        ({character} - end delimited comment symbol),
        end delimited comment symbol;
```

Single line comments start with `start single line comment symbol` `"//"` and end with any kind of `new line` character.

```
single line comment = {blank},
        start single line comment symbol,
        ({character} - new line),
        new line;
```

Comments include descriptions and notes about Modo declarations. Modo generators/compilers/interpreters ignore comments during generation, compiling and interpreting process. Comments are useful for making code more readable.

```
1  /* Beginning of Company element declaration */
2  Company {
3      /* Name member element definition */
4      Name = String;
5      /* Value assigned to Name element */
6      $Name = "Doe Software Inc.";
7      /* Founder member element definition */
```

```
8       Founder = String;
9       /* Value assigned to Founder element */
10      $Founder = "John Doe";
11      /* TaxNumber member element definition */
12      TaxNumber = String;
13      /* Value assigned to TaxNumber element */
14      $TaxNumber = "123456789987654321";
15  }
16  /* End of Company element declaration */
```

In the example above, comments are used on lines 1, 3, 5, 7, 9, 11, 13 and 16.

4.2. Values

There are two types of values in Modo syntax.

- Numeric values (integers and real numbers)
- Strings

All the element or member element values must be specified using either numeric values or strings.

4.2.1. Integers and real numbers

Integers and real numbers are formed by bringing digits `[0-9]` together.

Following is the Extended BNF notation of `integer` and `constant` (real number) syntax:

```
integer = "0" | (digit - "0", {digit});

constant = integer,
        [
            dot symbol,
            {digit}-
        ];
```

The syntax above shows that constant (real number) values include integer values. Decimal digits are separated using `dot symbol "."`.

4.2.2. Strings

Except numerals, all the values in Modo declarations must be specified as strings. Strings start with `single quote symbol` or `double quote symbol` and end with `single quote symbol` or `double quote symbol` respectively. Following examples show the different use of strings:

```
1  A = "This is double quote string.";
2  A = 'This is single quote string.';
3  A = "This is double quote \" escaped string";
4  A = 'This is single quote \' escaped string';
5  A = 'This is a \'string\'';
6  A = "This is a \"string\"";
```

5. Elements

Modo is used for describing systems. For better understanding, the system, its subsystems, its super system (the system that encompasses the system that is in focus), its environment and context can be separated into meaningful pieces. This approach has different names for different disciplines (e.g. Systems approach, object-oriented design). This approach is handy for analyzing and modeling systems. Sharing the same point of view, Modo offers separation of systems into signicifant unique pieces for modeling. These pieces are called “Elements” in Modo syntax. This chapter gives information about the declaration and usage of elements in Modo syntax.

5.1. Declarations

Declarations are used for representing the real-life system entities as Modo elements. There are three types of declarations: 1) Assignments, 2) basic declarations and 3) extended declarations.

Following sections give detailed information about declarations.

5.1.1. Assignments

Real-Life entities can have properties and sub elements. In Modo, these properties and sub elements are called “Member Elements”.

Declaration of the elements can be made by assigning values to the member elements. In the following example assignments are used to declare the `Company` element.

```
1 Company Name = String;
2 Company Founder = String;
3 Company TaxNumber = String;
4 Company $Name = "Doe Software Inc.";
5 Company $Founder = "John Doe";
6 Company $TaxNumber = "1234567899876543 21";
```

The example above shows the declaration of `Company` element. On line 1, `Name` attribute is defined and on line 4 the value `"Doe Software Co.,Ltd."` is assigned. On line 2, `Founder` attribute is defined and on line 5, the value `"John`

Doe" is assigned. Finally, on line 3, TaxNumber attribute is defined and on line 6, the value "123456789987654321" is assigned.

5.1.2. Basic declarations

Basic declarations are the typical element declarations in Modo syntax.

Following example shows the basic declaration of Company element:

```
1  Company {
2      Name = String;
3      $Name = "Doe Software Inc.";
4      Founder = String;
5      $Founder = "John Doe";
6      TaxNumber = String;
7      $TaxNumber = "123456789987654321";
8  }
```

The example above shows the declaration of Company element. Declaration of the Company element starts on line 1. Company element has three attributes called Name, Founder and TaxNumber. These attributes are defined on lines 2, 4 and 6 respectively; and their values are assigned on lines 3, 5, and 7. Finally, Company element declaration ends with end declaration symbol on line 8.

5.1.3. Extended declarations

Element declarations can be made by extending built-in or previously defined elements. The extended element carries all the properties of the parent element. Additionally, extended element can have its original member elements.

Following are the basic declarations of the Worker element and the Developer element which extends the Worker element:

```
1  Worker {
2      Name = String;
3      Surname = String;
4      Department = String;
5  }
```

The above example is the basic declaration of the Worker element. On line 1, the declaration of the element starts. On lines 2, 3 and 4, the member elements Name, Surname and Department are defined respectively.

```
1  Developer = Worker {
2      $Department = "Software Development";
3      Expertise = String;
4  }
```

On line 1, Developer element is declared extending the Worker element. With this assignment, extended declaration of the Developer element starts. On line 2, a value, "Software Development", is assigned to the Department element. In the above example the Department element belongs to the Worker element. So in line 2, the Developer element extends the Worker element via assigning a value to an existing member element. However in line 3, a new member element

`Expertise`, which belongs only to `Developer` element, is defined. So this time, the `Worker` element is extended via defining a new member element.

5.2. Using elements

Declaration process of the Modo elements is not limited to specifying the member elements, but it also includes relating them with each other and other Modo elements. This section gives details about using elements.

5.2.1. Member elements

In Modo syntax, all the properties/attributes of an element are also elements, which are called member elements. Member element declaration starting without any preceding characters, defines the type of the member element. The type of the member elements can be a built-in or previously declared element.

When the definition of the member element starts with the `dollar symbol "$"`, this assignment specifies the initial value of this element in the system. If the initial value is assigned with a rule, then that value will be condition dependent.

Following is the Modo declaration of the `Company` element:

```
1  Company {
2      Name = String;
3      $Name = "Doe Software Inc.";
4      $Name = "DOE Corp." @ ($CurrentYear > 2008);
5      Founder = String;
6      $Founder = "John Doe";
7      TaxNumber = String;
8      $TaxNumber = "123456789987654321";
9  }
```

In the above example, on lines 2, 5 and 7, the member elements `Name`, `Founder` and `TaxNumber` are declared respectively. Their types are also specified as `String`, which is another built-in or previously declared element. On lines, 3, 6 and 8 initial values are assigned with the help of `dollar symbol "$"`. On line 4, a different value is assigned to the `Name` member element, but this time depending on a rule `($CurrentYear > 2008)`.

Modo syntax also allows nested declarations where the member elements can extend an existing element. This nested declaration is one of the ways to define the relationship between the elements.

Following examples shows the nested declaration of member elements:

```
1  Worker {
2      Name = String;
3      Surname = String;
4      Department = String;
5  }
```

The example above shows the declaration of the `Worker` element. On lines 2, 3 and 4, the member elements `Name`, `Surname` and `Department` are declared respectively.

```
1  Department {
2      DepartmentName = String;
3      Manager = Worker{
4          $Department = $DepartmentName;
5      }
6  }
```

The above example shows the declaration of the `Department` element. On line 2, `DepartmentName` member element is declared. On line 3, member element `Manager` is declared extending the element `Worker`. On line 4, the value of the `Department` member element of the `Manager` element is assigned to the value of the `DepartmentName` member element.

5.2.2. Accessing member elements

The value of built-in elements, previously declared elements or member elements could be used while declaring the member elements.

Following examples give details about accessing member elements:

```
1  Worker {
2      FirstName = String;
3      LastName = String;
4      FullName = String;
5      $FullName = ($FirstName + " " + $LastName);
6  }
```

The example above shows the declaration of `Worker` element. On line 5, the value of the `FullName` member element is assigned using an expression that accesses the values of the `FirstName` and `LastName` member elements of the `Worker` element.

```
1  Worker {
2      $FirstName = "";
3      $LastName = "";
4      $FullName = ($FirstName + " " + $LastName);
5  }
6
7  Department {
8      DepartmentName = String;
9      Manager = Worker{
10         Title = String;
11         $Title = "Manager " + $FullName;
12     }
13 }
```

Above is an example of accessing element attributes in extended declaration. On line 11, `Title` attribute is assigned a value using an expression which accesses `FullName` attribute of the `Worker` element.

5.3. Functions and member methods

Modo syntax does not allow the definition of user defined functions and member methods. However, Modo generators/compilers/interpreters can provide some built-in functions and member methods.

This section gives information about using built-in functions and member methods in Modo declarations.

Following example shows the use of member methods in Modo declarations:

```
1  Worker {
2      FirstName = String;
3      LastName = String;
4      FullName = String;
5      $FullName = ($FirstName + " " + $LastName);
6
7      /* FormatCurrency function takes 2 parameters
8       * param1: Amount
9       * param2: Decimal Count
10      */
11     Salary = String;
12     $Salary = FormatCurrency(1500, 2);
13 }
```

The example above shows the use of member methods in Modo declarations. On line 12, the value of the `Salary` member element is specified by the return value of `FormatCurrency` function. `FormatCurrency` function takes two parameters: 1) Currency amount and 2) decimal count. The value of the `Salary` member element becomes `"1500.00"`.

Warning: `FormatCurrency` function used in the example above only shows the use of member methods. This function is not a part of Modo syntax.

Note: The above examples assume that the element `String` has already been declared.

6. Selectors

Selectors provide control over elements. Selectors can define the scope of the declaration.

This chapter gives information about the uses of selectors over elements.

6.1. Selector syntax

There are two types of selectors:

- Left-hand side selectors
- Right-hand side selectors

Following is the Extended BNF notation of left-hand and right-hand side selectors:

```
lhs selector = (
            {blank},
            universal selector | selector operand,
            {blank},
            {
                selector operator,
                {blank},
                selector operand
            }
        ),
        {
            {blank},
            comma symbol,
            {blank},
            lhs selector
        };
```

Left-hand side selectors have more complex structure than right-hand side selectors. The Extended BNF notation above shows that `lhs selector` consists of a `selector operand` or a `universal selector`, and a `selector operator`. Additionally, it is possible to specify more than one `lhs selector` by separating with `comma symbol ","`.

```
rhs selector = {blank},
        [unary selector operator],
        identifier,
        {rhs selector};
```

The Extended BNF notation above shows that `rhs selector` consists of optional `unary selector operator` and an `identifier`. In addition to that, it is possible to specify more than one `rhs selector`.

6.2. Universal Selector

In Modo selector syntax, `"*"` symbol plays an important role. This symbol is called `universal selector` and used to match any single element. As specified in Appendix A. Grammar, `universal selector` can only be used in left-hand side selectors. The main reason for this convention is, right-hand side selectors are not allowed to address more than one element.

6.3. Grouping

Modo syntax provides grouping for left-hand side selectors. Left-hand side selectors may specify more than one element at a time. This is useful for declaring identical elements.

Warning: Grouping can only be used on left-hand side selectors. Right-hand side selectors have no grouping support.

Following example shows the use of grouping:

```
1  Developer, Designer = Worker {
2      FirstName = String;
3      LastName = String;
4  }
```

The example above shows the use of grouping. Both `Developer` and `Designer` elements are of type the `Worker` element. These elements are separated with `comma symbol`.

6.4. Selector operators

There are two types of selector operators:

- Unary selector operators
- Binary selector operators

This section gives information about the use of these selector operators.

6.4.1. Unary selector operators

Unary selector operators are the selector operators that take one selector operand.

Following is the Extended BNF notation of the unary selector operators:

```
unary selector operator = "$", ".", "#", "?", " ";
```

Following examples shows the use of unary selector operators:

```
.Developer { }
#Developer { }
?Developer { }
Developer $Name = "";
```

Warning: The meaning and the use of these selector operators may be interpreted differently by different Modo generators/compilers/interpreters.

6.4.2. Binary selector operators

Binary selector operators are the selector operators that take two selector operands.

Following is the Extended BNF notation of the binary selector operators:

```
selector operator = ">", "<", "+", "-", "~", "^", ":",
        "::", "%", "&", "|", "\", "/";
```

Following examples shows the use of binary selector operators:

```
Developer > $Name = "";
Developer < $Name = "";
Developer + Designer { }
Developer - Designer { }
Developer ~ Designer { }
Developer ^ Designer { }
Developer % Designer { }
Developer & Designer { }
Developer | Designer { }
Developer / Designer { }
```

Warning: The meaning and the use of these selector operators may be interpreted differently by different Modo generators/compilers/interpreters.

6.5. Selector operands

Selector operands are the components of the selection operation affected by binary and unary selector operators.

Following is the Extended BNF notation of `selector operand` in Modo syntax:

```
selector operand = {blank},
        [
            [unary selector operator],
            lhs identifier
        ],
        {blank},
        [
            (
                start group symbol,
                [
                    selector expression,
                    {
                        {blank},
                        comma symbol,
                        {blank},
                        selector expression
                    }
                ],
                {blank},
                end group symbol
            )
            |
            (
                start attribute selector symbol,
                selector expression,
                {
                    {blank},
                    comma symbol,
                    {blank},
                    selector expression
                },
                {blank},
                end attribute selector symbol
            )
        ],
        {selector operand};
```

The notation above shows that selector operands start with optional `unary selector operator` and `lhs identifier`. It is possible to specify a selector start with either `start group symbol "("` or `start attribute selector symbol "["`. If a selector starts with `start group symbol`, it must end with `end`

`group symbol`. Similarly, if a selector starts with `start attribute selector symbol`, it must end with `end attribute selector symbol`.

6.6. Selector expressions

By the help of the selector operand expressions, not only the elements but also its member elements, member element values and member methods can be used in the left-hand side selectors.

Following is the Extended BNF notation of `selector expression` in Modo syntax:

```
selector expression = {blank},
        {start group symbol},
        {blank},
        operand | expression,
        {
            {blank},
            selector expression operator,
            {blank},
            {start group symbol},
            {blank},
            operand | expression,
            {blank},
            {end group symbol},
        },
        {blank},
        {end group symbol};
```

The `selector expression` notation above shows that `selector expression` and `expression` notation have several tokens in common except the `selector expression operator`.

Following is the Extended BNF notation of `selector expression operator`:

```
selector expression operator = "*" | "/" | "%" | "+" | "-"
        | "<<" | ">>" | "<" | ">" | "=" | "&" | "^" | "|"
        | "<=" | ">=" | "==" | "!=" | "*=" | "/=" | "%="
        | "+=" | "-=" | "&=" | "^=" | "|=" | "$=" | "~="
        | ".=" | "#=" | "?=" | ":=" | "&&" | "||";
```

7. Expressions

An expression can be defined as a sequence of operands and operators (ECMA-334, 2006). In Modo syntax, expressions are used for the definition of elements and member elements.

Following is the Extended BNF notation of the `expression` in Modo syntax:

```
expression = {blank},
        {start group symbol},
        {blank},
        operand | expression,
        {
            {blank},
            operator,
            {blank},
            {start group symbol},
            {blank},
            operand | expression,
            {blank},
            {end group symbol},
        },
        {blank},
        {end group symbol};
```

The `expression` definition above shows that expressions start with optional `start group symbol, "("`. An `operand` or `expression` follows this symbol. Then optional `operator` and `operand/expression` follow. Finally expression syntax ends with optional `end group symbol, ")"`.

7.1. Operators

As mentioned in `expression` syntax, expressions are composed of operands and operators. Operators specify the operations that will be applied to the operands (ECMA-334, 2006).

In Modo syntax, there are two types of operators: 1) Binary operators and 2) unary operators.

Unary operators take one operand (e.g. `-A`, `A++`).

Binary operators take two operands (e.g. `A + B`).

Following is the Extended BNF notation of `unary operator` and `operator` (Binary operator) in Modo syntax:

```
unary operator = "+" | "-" | "!" | "~"
        | increment operator
        | decrement operator;

operator = "*" | "/" | "%" | "+" | "-" | "<<" | ">>" | "<"
        | ">" | "<=" | ">=" | "==" | "!=" | "&" | "^" | "|"
        | "&&" | "||";
```

Expressions are evaluated from left to right. Operator precedence is the key factor that determines which operations will be applied to which operands.

Following table lists the precedence of the operators:

Operator Precedence

	Category	Operators
1	Unary operators	+ - ! ~ ++ --
2	Multiplication operators	* / %
3	Addition operators	+ -
4	Shifting operators	<< >>
5	Relational operators	< > <= =>
6	Equality operators	== !=
7	Logical AND operator	&
8	Logical XOR operator	^
9	Logical OR operator	|
10	Conditional AND operator	&&
11	Conditional OR operator	||

First column specifies the priority of the operator. Unary operators have the highest priority (1) and conditional OR operator has the lowest (11) one. Second column specifies the category of the operator. Finally, third column lists the operators.

7.1.1. Unary operators

Unary operators have the highest priority. These operators consist of 1) unary plus operator, 2) unary minus operator, 3) negation operator, 4) bitwise complement operator and 5) increment and decrement operator:

Unary plus operator is used to form an operation like `+A`. This operation results simply the value of the operand.

Unary minus operator is used to form an operation like `-A`. This operation results the multiplication of the operand by (`-1`).

Negation operator is used to form an operation like `!A`. This operation results the logical negative value of the operand.

Bitwise complement operator is used to form an operation like `~A`. This operation results the complement of the operand.

Increment and decrement operators is used to form an operation like `++A`, `--A`, `A++` or `A--`. Increment operator (`++A` or `A++`) increases the operand value by one. Similarly decrement operator (`--A` or `A--`) decreases the operand value by one.
Evaluating expressions from left to right can yield different results for `++A` and `A++`. One must remember that, the operation `++A` is calculated before the `A++` operation.

7.1.2. Multiplication operators

Multiplication operators have second highest priority. Multiplication operators consist of 1) multiplication operator, 2) division operator and 3) remainder operator:

Multiplication operator is used to form an operation like `A * B`. This operation results the multiplication of the operand `A` by `B`.

Division operator is used to form an operation like `A / B`. This operation results the division of the operand `A` by `B`.

Remainder operator is used to form an operation like `A % B`. This operation results the remainder of the division of the operand `A` by `B`.

7.1.3. Addition operators

Addition operators consist of 1) addition operator and 2) subtraction operator:

Addition operator is used to form an operation like `A + B`. This operation results the addition of the operands `A` and `B`.

Subtraction operator is used to form an operation like `A - B`. This operation results the subtraction of the operand `B` from `A`.

7.1.4. Shifting operators

`"<<"` and `">>"` symbols are used for bit shifting operations. Shifting operators consists of 1) left-shifting operator and 2) right shifting operator:

Left-shifting operator is used to form an operation like `A << 5`. The `A << 5` operation yields the `5` bits left-shifted value of the operand `A`.

Right-shifting operator is used to form an operation like `A >> 5`. The `A >> 5` operation yields the `5` bits right-shifted value of the operand `A`.

7.1.5. Relational operators

`"<"`, `">"`, `"<="` ve `">="` symbols are used for relational operations. Relational operators consist of 1) “less-than” operator, 2) “greater than” operator, 3) “less than or equal to” operator and 4) “greater than or equal to” operator:

"Less than" operator is used to form an operation like `A < B`. If the value of the operand `A`, is less than the value of the operand `B`, `A < B` operation yields the logical `true` value. If the value of the operand `A`, is greater than or equal to the value of the operand `B`, `A < B` operation yields the logical `false` value.

"Greater than" operator is used to form an operation like `A > B`. If the value of the operand `A`, is greater than the value of the operand `B`, `A > B` operation yields the logical `true` value. If the value of the operand `A`, is less than or equal to the value of the operand `B`, `A > B` operation yields the logical `false` value.

"Less than or equal to" operator is used to form an operation like `A <= B`. If the value of the operand `A`, is less than or equal to the value of the operand `B`, `A <= B` operation yields the logical `true` value. If the value of the operand `A` is greater than the value of the operand `B`, `A <= B` operation yields the logical `false` value.

"Greater than or equal to" operator is used to form an operation like `A >= B`. If the value of the operand `A` is greater than or equal to the value of the operand `B`, `A >= B` operation yields the logical `true` value. If the value of the operand `A` is less than the value of the operand `B`, `A >= B` operation yields the logical `false` value.

7.1.6. Equality operators

`"=="` and `"!="` operators are used for equality operations. Equality operators consists of 1) "equal to" operator and 2) "not equal to" operator.

"Equal to" operator is used to form an operation like `A == B`. If the value of the operand `A` is equal to the value of the operand `B`, `A == B` operation yields the logical `true` value.

"Not equal to" operator is used to form an operation like `A != B`. The `A != B` operation is logically negative to the operation `A == B`. If the value of the operand `A` is not equal to the value of the operand `B`, `A != B` operation yields the logical `true` value.

7.1.7. Logical AND operator

Logical AND operator is used to form an operation like `A & B`. This operation results the bitwise logical AND of the operands `A` and `B`.

7.1.8. Logical XOR operator

Logical XOR operator is used to form an operation like `A ^ B`. This operation results the bitwise logical exclusive OR of the operands `A` and `B`.

7.1.9. Logical OR operator

Logical OR operator is used to form an operation like `A | B`. This operation results the bitwise logical OR of the operands `A` and `B`.

7.1.10. Conditional AND operator

Conditional AND operator is used to form an operation like `A && B`. This operation results the coditional AND of the operands `A` and `B`.

7.1.11. Conditional OR operator

Conditional OR operator is used to form an operation like `A || B`. This operation results the conditional OR of the operands `A` and `B`.

7.2. Constants and strings

Constant numeric values and strings can be used as operands in expressions.

Following is the Extended BNF notation of `constant` and `string` in Modo syntax:

```
constant = integer,
        [
            dot symbol,
            {digit}-
        ];
```

The `constant` definition above shows that only the numeric values are considered as constants. A `constant` can be integer or real number. Decimal part of the number starts with `dot symbol "."`. Just like integer and real number which are numeric constants, `string` is the alphanumeric constant.

```
string = (
            single quote symbol,
            single quote escaped string,
            single quote symbol
        )
        |
        (
            double quote symbol,
            double quote escaped string,
            double quote symbol
        );
```

The `string` definition above shows that strings can start with `single quote symbol '` or `double quote symbol "`.

Warning: If a `string` starts with `single quote symbol`, it must end with `single quote symbol`. In addition, if a `string` starts with `double quote symbol`, it must end with `double quote symbol`.

7.3. Elements and member elements

Elements and member elements can be considered as operands in Modo expressions.

For further information about elements and member elements, please refer to 5.2. Using elements.

7.4. Functions and member methods

Unlike C, C++, C#, Java etc., function and/or method definition is not allowed in Modo syntax. Modo represents and forms the system and its states for the given time periods and for the given set of rules. Nevertheless, there can be built-in elements, functions and member methods defined in Modo language references specific for certain generators/compilers/interpreters. These built-in elements, functions and member methods can also be considered as operands in Modo expressions.

Appendix A. Grammar


```
(**
 * MODO LANGUAGE DEFINITION
 * ------------------------
 * This chapter defines Modo language syntax using
 * Extended BNF metalanguage.
 * There are three parts in this definition:
 * (1) Letters, characters, digits, symbols and operators
 * (2) Identifier, expression and other lexical units
 * (3) Final Modo definition
 *
 *)

(**
 * MODO LANGUAGE DEFINITION Part I
 * -------------------------------
 * Letters, characters, digits, symbols and operators
 *)

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
       | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P"
       | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
       | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f"
       | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
       | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
       | "w" | "x" | "y" | "z";

character = any Unicode character;

whitespace = {
    space character
    | tab character
    | line feed character
    | carriage return character
    | form feed character
};

new line = carriage return character
         | line feed character
         | next line character
         | line separator character
         | paragraph separator character;
```

```
space character = " ";

tab character =
        ? The Unicode Standard Tab Character (U+0009) ? ;

line feed character =
        ? The Unicode Standard Line Feed Character (U+000A) ? ;

carriage return character =
        ? The Unicode Standard
        Carriage Return Character (U+000D) ? ;

form feed character =
        ? The Unicode Standard Form Feed Character (U+000C) ? ;

next line character =
        ? The Unicode Standard Next Line Character (U+0085) ? ;

line separator character =
        ? The Unicode Standard
        Line Separator Character (U+2028) ? ;

paragraph separator character =
        ? The Unicode Standard
        Paragraph Separator Character (U+2029) ? ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
        | "8" | "9";

at symbol = "@";

assignment symbol = "=";

end sentence symbol = ";";

start group symbol = "(";

end group symbol = ")";

start declaration symbol = "{";

end declaration symbol = "}";

backslash symbol = "\";

start attribute selector symbol = "[";

end attribute selector symbol = "]";

start delimited comment symbol = "/*";

end delimited comment symbol = "*/";

start single line comment symbol = "//";

dot symbol = ".";

comma symbol = ",";
```

```
underscore symbol = "_";

hyphens symbol = "-";

escaped single quote symbol = "\'";

escaped double quote symbol = '\"';

single quote symbol = "'";

double quote symbol = '"';

increment operator = "++";

decrement operator = "--";

unary operator = "+" | "-" | "!" | "~"
        | increment operator
        | decrement operator;

operator = "*" | "/" | "%" | "+" | "-" | "<<" | ">>" | "<"
        | ">" | "<=" | ">=" | "==" | "!=" | "&" | "^" | "|"
        | "&&" | "||";

unary selector operator = "$", ".", "#", "?", " ";

selector operator = ">", "<", "+", "-", "~", "^", ":",
        "::", "%", "&", "|", "\", "/";

selector expression operator = "*" | "/" | "%" | "+" | "-"
        | "<<" | ">>" | "<" | ">" | "=" | "&" | "^" | "|"
        | "<=" | ">=" | "==" | "!=" | "*=" | "/=" | "%="
        | "+=" | "-=" | "&=" | "^=" | "|=" | "$=" | "~="
        | ".=" | "#=" | "?=" | ":=" | "&&" | "||";

universal selector = "*";

(**
 * MODO LANGUAGE DEFINITION Part II
 * --------------------------------
 * Identifier, expression and other lexical units
 *)

(**
 * Directive Definition
 *)

directive = {blank},
        at symbol,
        identifier,
        {blank}-,
        expression,
        {blank},
        end sentence symbol;
```

```
(**
 * Rule Definition
 *)

rule = {blank},
        at symbol,
        {blank},
        start group symbol,
        {blank},
        expression,
        {blank},
        end group symbol,
        {blank},
        start declaration symbol,
        {blank},
        modo,
        {blank},
        end declaration symbol;


inline rule = {blank},
        at symbol,
        {blank},
        start group symbol,
        {blank},
        expression,
        {blank},
        end group symbol;

(**
 * Declaration Definition
 *)

declaration = {
        assignment
        | basic declaration
        | extended declaration
};

(**
 * Comment Definition
 *)

comment = delimited comment
        | single line comment;

delimited comment = {blank},
        start delimited comment symbol,
        ({character} - end delimited comment symbol),
        end delimited comment symbol;

single line comment = {blank},
        start single line comment symbol,
        ({character} - new line),
        new line;

blank = whitespace | comment;
```

```
(**
 * Other lexical units
 *)

string = (
            single quote symbol,
            single quote escaped string,
            single quote symbol
        )
        |
        (
            double quote symbol,
            double quote escaped string,
            double quote symbol
        );

single quote escaped string = {
        (any Unicode character - single quote symbol)
        | escaped single quote symbol
};

double quote escaped string = {
        (any Unicode character - double quote symbol)
        | escaped double quote symbol
};

integer = "0" | (digit - "0", {digit});

constant = integer,
        [
            dot symbol,
            {digit}-
        ];

expression = {blank},
        {start group symbol},
        {blank},
        operand | expression,
        {
            {blank},
            operator,
            {blank},
            {start group symbol},
            {blank},
            operand | expression,
            {blank},
            {end group symbol},
        },
        {blank},
        {end group symbol};

member method name = identifier;

identifier = {letter | underscore symbol}-,
        {
            digit
            | letter
            | underscore symbol
        };
```

```
operand = {start group symbol},
          {blank},
          (
              string
          )
          |
          (
              [
                  unary operator
                  - (increment operator | decrement operator)
              ],
              {blank},
              constant
          )
          |
          (
              [unary operator],
              {blank},
              rhs selector
          )
          |
          (
              rhs selector,
              {blank},
              increment operator | decrement operator
          )
          |
          (
              [
                  unary operator
                  - (increment operator | decrement operator)
              ],
              {blank},
              member method name,
              {blank},
              start group symbol,
              {blank},
              [
                  expression,
                  {
                      {blank},
                      comma symbol,
                      expression
                  }
              ],
              {blank},
              end group symbol
          ),
          {blank},
          {end group symbol};
```

```
assignment = {blank},
        lhs selector,
        {blank},
        assignment symbol,
        {blank},
        expression,
        {blank},
        [inline rule],
        {blank},
        end sentence symbol;

basic declaration = {blank},
        lhs selector,
        {blank},
        start declaration symbol,
        {blank},
        modo,
        {blank},
        end declaration symbol;

extended declaration = {blank},
        lhs selector,
        {blank},
        assignment symbol,
        {blank},
        rhs selector,
        {blank},
        start declaration symbol,
        {blank},
        modo,
        {blank},
        end declaration symbol;

lhs identifier = {
            letter
            | underscore symbol
            | hyphens symbol
        }-,
        {
            digit
            | letter
            | underscore symbol
            | hyphens symbol
        };
```

```
lhs selector = (
            {blank},
            universal selector | selector operand,
            {blank},
            {
                selector operator,
                {blank},
                selector operand
            }
        ),
        {
            {blank},
            comma symbol,
            {blank},
            lhs selector
        };

rhs selector = {blank},
        [unary selector operator],
        identifier,
        {rhs selector};

selector operand = {blank},
        [
            [unary selector operator],
            lhs identifier
        ],
        {blank},
        [
            (
                start group symbol,
                [
                    selector expression,
                    {
                        {blank},
                        comma symbol,
                        {blank},
                        selector expression
                    }
                ],
                {blank},
                end group symbol
            )
            |
            (
                start attribute selector symbol,
                selector expression,
                {
                    {blank},
                    comma symbol,
                    {blank},
                    selector expression
                },
                {blank},
                end attribute selector symbol
            )
        ],
        {selector operand};
```

```
selector expression = {blank},
        {start group symbol},
        {blank},
        operand | expression,
        {
            {blank},
            selector expression operator,
            {blank},
            {start group symbol},
            {blank},
            operand | expression,
            {blank},
            {end group symbol},
        },
        {blank},
        {end group symbol};

(**
 * MODO LANGUAGE DEFINITION Part III
 * ---------------------------------
 * Final Modo definition
 *)

modo = {
    directive
    | rule
    | declaration
    | comment
};
```

Appendix B. Bibliography

Berners-Lee T., Hendler J., Lasilla O., *The Semantic Web*, Scientific American 284, 34 – 43, 2001

Booch, G., Jacobson, I. and Rumbaugh, J., *OMG Unified Modeling Language Specification*, *Version 1.3 1st Edition*, Object Management Group (OMG), 2000.

Bos, B., Çelik, T., Hickson, I. and Lie, H. W. (Eds.), *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, W3C Candidate Recommendation 23 April 2009*, World Wide Web Consortium (W3C), http://www.w3.org/TR/CSS2/, Last Access: 02.06.2009.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L. and Vanderdonckt, J., *A Unifying Reference Framework for multi-target user interfaces*, Interacting with Computers 15, 289 – 308, 2003.

ECMA-334, *C# Language Specification*, *4th Edition*, ECMA International, 2006.

Heng, B.C.P. and Mackie, R.I., *Using design patterns in object-oriented finite element programming*, Computers and Structures 87, 952 – 961, 2009.

Horsburgh, J.S., Tarboton, D.G., Piasecki, M., Maidment, D.R., Zaslavsky, I., Valentine, D. and Whitenack, T., *An integrated system for publishing environmental observations data*, Environmental Modelling & Software 24, 879 – 888, 2009.

Gosling, J., Joy, B., Steele, G. and Bracha, G., *The Java language specification*, *3rd Edition*, Addison-Wesley, 2005.

ISO/IEC 14977, *Information technology – Syntactic metalanguage – Extended BNF*, *1st Edition*, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 1996.

Kernighan, B., and Ritchie, D., *The C Programming Language*, *1st Edition*, Prentice Hall, 1978.

Kim, K.S., *Information seeking on the Web: Effects of user and task variables*, Library & Information Science Research 23, 233 – 255, 2001.

Lam, D. and Swayne, D., *Issues of EIS software design: some lessons learned in the past decade*, Environmental Modeling & Software 16, 419 – 425, 2001.

Laporti, V., Borges, M.R.S. and Braganholo, V., *Athena: A collaborative approach to requirements elicitation*, Computers in Industry 60, 367 – 380, 2009.

Park, K.S. and Lim and C.H., *A structured methodology for comparative evaluation of user interface designs using usability criteria and measures*, International Journal of Industrial Ergonomics 23, 379 – 389, 1999.

Rau, P.P., Choong, Y. and Salvendy, G., *A cross-cultural study on knowledge representation and structure in human computer interfaces*, International Journal of Industrial Ergonomics 34, 117 – 129, 2004.

Reinhartz-Berger, I. and Sturm, A., *Utilizing domain models for application design and validation*, Information and Software Technology 51, 1275 – 1289, 2009.

Schwabe D., Rossi G., *An Object Oriented Approach to Web-Based Applications Design*, Theory and Practice of Object Systems 4, 207 – 225, 1998

The Unicode Standard, Version 4.1.0, The Unicode Consortium, Boston, MA, Addison-Wesley, Available at http://www.unicode.org/versions/Unicode4.1.0/, Last Access: 02.06.2009, 2003.

Tom, A., Inside C#, Microsoft Press, 2001.

Wirth, N., *The Programming Language Pascal*, Acta Informatica 1, 35 – 63, 1971.