Journal of Biomedical Informatics 83 (2018) 196–203

Contents lists available at ScienceDirect

Journal of Biomedical Informatics

journal homepage: www.elsevier.com/locate/yjbin

A detailed analysis of the Arden Syntax expression grammar

Stefan Kraus[a,*], Marc Rosenbauer[a], Lutz Schröder[b], Thomas Bürkle[c], Klaus-Peter Adlassnig[d,e], Dennis Toddenroth[a]

[a] *Department of Medical Informatics, Biometrics and Epidemiology, Chair of Medical Informatics, Friedrich-Alexander-University Erlangen-Nuremberg, Wetterkreuz 13, 91058 Erlangen, Germany*
[b] *Department of Computer Science, Chair of Theoretical Computer Science, Friedrich-Alexander-University Erlangen-Nuremberg, Martensstrasse 3, 91058 Erlangen, Germany*
[c] *Bern University of Applied Sciences, Institute for Medical Informatics, Höheweg 80, CH-2502 Biel, Switzerland*
[d] *Section for Artificial Intelligence and Decision Support, Center for Medical Statistics, Informatics, and Intelligent Systems, Medical University of Vienna, Spitalgasse 23, A-1090 Vienna, Austria*
[e] *Medexter Healthcare GmbH, Borschkegasse 7/5, A-1090 Vienna, Austria*

A R T I C L E I N F O

Keywords:
Arden Syntax
Medical Logic Modules
Domain-specific language

A B S T R A C T

Objective: The Arden Syntax for Medical Logic Systems is a standard for encoding and sharing medical knowledge in the form of Medical Logic Modules. To improve accessibility for clinicians, the originators of the standard deliberately designed Arden Syntax expressions to resemble natural language, and parentheses around operands are not generally required. For certain patterns of nested expressions, however, the use of parentheses is mandatory, otherwise they are not accepted by an Arden Syntax environment. In this study, we refer to such patterns as anomalies. The purpose of this paper is to investigate the extent and the circumstances of such anomalies, and to outline a solution based on an alternative grammar encoding approach.
Methods: To analyze the distribution of anomalies in nested expressions, we developed two custom-made complementary utilities. The first utility, termed parser, checks a single expression pattern against the specification-compliant grammar for syntactic correctness. The second utility, termed composer, automatically creates an extensive amount of expression patterns by permuting and nesting operators without the use of parentheses, and stores these together with the expected syntactic correctness. By means of these utilities we conducted a comprehensive analysis of anomalies by comparing the expected correctness with the actual correctness. Any detected anomalies are stored into a set of files, grouped by the respective top-level operator, for a subsequent analysis.
Results: The composer utility nested 165 unary, binary, or ternary operators of Arden Syntax version 2.8 to a depth of two, resulting in a set of 76,533 expression patterns, of which 18,978 (24.8%) have been identified as anomalies. An automated assessment of their practical relevance for medical knowledge encoding is infeasible. Manual screening of selected samples indicated that only a small proportion of the detected anomalies would be relevant. The cause of the anomalies lies in the encoding of the grammar. A change of the basic encoding approach with some additional customizations eliminates the anomalies. A working expression parser is included in the supplementary material.
Conclusion: Arden Syntax expressions are affected by anomalies. Since only a small proportion of them have practical relevance and they cannot cause false calculations or clinical decisions, their practical impact is likely limited. However, they may be potential points of confusion for knowledge engineers. An alternative expression grammar, based on a different encoding approach, would not only eliminate the anomalies, but could considerably facilitate both maintenance and further development of the standard.

1. Introduction

The Arden Syntax for Medical Logic Systems is a standard for encoding and sharing medical knowledge in the form of Medical Logic Modules (MLMs) [1], maintained by the Health Level 7 (HL7) organization. A single MLM typically contains sufficient knowledge to support a specific clinical decision, such as whether a hypoglycemic patient requires therapy in a particular situation. The Arden Syntax is a highly

* Corresponding author.
E-mail address: stefan.kraus@uk-erlangen.de (S. Kraus).

https://doi.org/10.1016/j.jbi.2018.05.008
Received 21 January 2018; Received in revised form 11 May 2018; Accepted 13 May 2018
Available online 27 June 2018

specialized language, tailored to the requirements of data-driven clinical event monitoring [2]. During its use in clinical routine at the University Hospital Erlangen, Germany [3], we repeatedly observed that this standard is useful not only for event monitoring functions, but for many other programming tasks, such as user-driven information retrieval at the bedside. Our experiences were so promising that the Arden Syntax appeared as a suitable starting point for a domain-specific language (DSL) for medical informatics in general. However, we also detected some minor issues in the Arden Syntax specification that may require revision, especially with regard to the potential use of MLMs beyond their originally intended scope of application. This study is part of a research project to generalize the Arden Syntax to an easy-to-use DSL for various clinical applications [4]. The prototype of this DSL has been termed PLAIN (**P**rogramming **L**anguage, **A**rden-**IN**spired) to point out its origin. In contrast to the highly specialized Arden Syntax, PLAIN is intended to be generic within the medical informatics domain. This disparate nature of both languages requires multiple adjustments to the language constructs, the data type system, and the interactions with the surrounding clinical environment. This study concentrates on the encoding of operators and expressions, the most cumbersome part of both languages.

The main motivation for this investigation of the Arden Syntax grammar was to systematically delineate the circumstances of several cases, reported by Arden Syntax users, where specific expressions resulted in unexpected syntax error messages. An example has been communicated by a participant of the HL7 Arden Syntax working group in the mailing list in October 2015. It refers to the first argument of the SUBLIST operator (see specification [5], section 9.14.6).

```
(1A)    SUBLIST 2 ELEMENTS FROM crp               → OK
(1B)    SUBLIST -2 ELEMENTS FROM crp              → SYNTAX ERROR
(1C)    SUBSTRING 2 CHARACTERS FROM germcode      → OK
(1D)    SUBSTRING -2 CHARACTERS FROM germcode     → OK
```

Expression (1A) extracts the first two elements from a list of laboratory results. Expression (1B) is intended to likewise extract the last two elements from a list (the minus sign indicates extraction from the end of a list), but standard-compliant compilers fail to parse a negative first argument here, and instead report a syntax error. The Arden Syntax also provides a SUBSTRING operator ([5], 9.8.10), which extracts a sub-part from a specific string. While both the SUBLIST and the SUBSTRING operator functionally support a negative first operand, only in case of SUBLIST will its invocation produce a syntax error, unless the first operand is parenthesized. Thus, counterintuitively, expression (1B) is not processed, in contrast to expression (1D).

```
(1E)    SUBSTRING 2 CHARACTERS FROM FIRST matcodes    → OK
(1F)    ATTRIBUTE "material" FROM FIRST findings      → SYNTAX ERROR
```

Earlier research on the mapping of complex patient data [6] highlighted another example related to the ATTRIBUTE FROM operator ([5], 9.18.4). In both expressions (1E) and (1F), the second operands (underlined for accentuation) were expected to be valid. While in example (1E) the underlined expression is a valid operand, the very same expression results in a syntax error in case of (1F).

```
(1G)    SORT ROUND values     → OK
(1H)    ROUND SORT values     → SYNTAX ERROR
(1I)    ROUND LAST values     → OK
```

Another significant example refers to sorting and rounding of lists with numeric values, such as a list of calculated clinical score values. Example (1G) rounds the values first, and then sorts them. Example (1H) should sort the values first, and then round them, but results in a syntax error. Example (1I) is similar to (1H), except that SORT has been replaced with LAST, but is accepted by a compiler.

Such cases of counterintuitive syntax errors, which we will further refer to as anomalies, may be considered a minor problem insofar as they can be solved in a simple way. In example (1B), the workaround would be to parenthesize the first operand "-2" of the SUBLIST operator. The solution for example (1F) would be to parenthesize the second operand "FIRST findings" of the ATTRIBUTE FROM operator. The solution for example (1H) would be to parenthesize "SORT values". Moreover, such anomalies appear to be infrequent and cannot lead to false decisions, because they are reported at compile time and thus not processed at all. However, they may impede the knowledge engineering process, insofar as MLM authors may be faced with incomprehensible compile errors; this may result in a decreasing readiness to write MLMs. In the above examples, the error messages will likely be inexplicable for most Arden Syntax users. Moreover, in case of such an unexpected syntax error, a user can hardly determine which of the operands needs to be parenthesized, or may not even be aware of such a solution. The purpose of this study is to perform a detailed analysis of the Arden Syntax grammar in order to determine the extent of anomalies in Arden Syntax expressions, and the reasons for their occurrence. In addition, we discuss an alternative encoding approach that fixes this issue and might also considerably simplify the maintenance of the standard.

2. Background

2.1. Medical Logic Modules

An MLM corresponds to a production rule (CONDITION-ACTION) insofar as it typically evaluates a predefined condition in conjunction with patient-specific input data, and then possibly performs an action. Such a condition might be the occurrence of a low glucose value; the corresponding action might be to dispatch an alert message to a physician. The decision logic of an MLM is not limited to basic operations such as comparisons of individual patient attributes, but can be a procedural program of arbitrary complexity. MLMs have a frame-like structure, consisting of sections termed categories, which are themselves subdivided into multiple slots. Four specific slots within the KNOWLEDGE category ("structured slots") contain programming language constructs that resemble those used in common all-purpose programming languages. For an introduction into writing MLMs see Hripcsak [7].

To convert MLMs into an executable format, Arden Syntax compilers described in the literature usually either transform code into intermediate all-purpose programming languages such as Java [8–10] or C++ [11], or into specific virtual machine code [12]. The construction of such a compiler by means of a parser generator presupposes a computable language description in the form of a grammar, usually provided in Backus-Naur form. The Arden Syntax specification provides such a grammar for common LALR(1) parsers. This abbreviation stands for "look ahead, left to right, rightmost derivation", with a single token look ahead, whereby a token is a meaningful chunk of input, such as a keyword, a variable name, or an arithmetic operator. Based on a grammar, a compiler can decide whether Arden Syntax code is syntactically correct or not. For an introduction on this matter see for example [13] or [14].

2.2. Statements and expressions

A structured slot of an MLM, such as the LOGIC slot for the actual clinical decision, contains statements, such as variable assignments, conditional branching, and loops. These statements process expressions composed of operators tailored to the requirements of clinical decision support functions.

```
(1J)  WRITE glucose WHERE IT OCCURRED WITHIN THE PAST
      24 HOURS;
(1K)  WRITE ((glucose) WHERE ((IT) OCCURRED WITHIN THE
      PAST (24 HOURS)));
(1L)  WRITE (WHERE (glucose, OCCURRED_WITHIN_PAST
      (glucose, HOURS (24))));
```

Example (1J) shows a valid Arden Syntax WRITE statement ([5], 12.2.1). It outputs the glucose readings of a patient that have been documented within the past 24 h to the standard destination of the specific environment. The keyword WRITE must be followed by an expression and a semicolon. An expression is a combination of operators and their operands, which may be constants, variables, or sub-expressions. The nested expression in example (1J) (underlined for accentuation) is composed of a WHERE operator ([5], 9.3.1), an OCCURRED WITHIN PAST operator ([5], 9.7.7), and an HOUR ([5], 9.11.5) operator. This example also contains the variable glucose, the keyword IT (which, in this example, refers to the value of glucose), and the numeric constant 24. Expressions are evaluated at runtime by an Arden Syntax environment and result in a value. The underlined expression in example (1J) results in a list of numeric values. A so-called atom constitutes a special case of an expression that is not built from sub-expressions, and thus cannot be decomposed to sub-expressions. Examples for atoms include the variable name glucose and the numeric literal 24. The expression "`24 HOURS`", in contrast, is not an atom, since it is formed by the unary HOUR operator and the numeric operand 24. To determine the evaluation order of a nested expression at runtime, any operator of the Arden Syntax is associated with a precedence level, described in annex A4 of the specification.

As in most common all-purpose programming languages, Arden Syntax operators may be left-associative, right-associative, or non-associative. In the latter case, consecutive operator use is treated by the compiler as a syntax error (this is an intended behavior and has no relation to anomalies, see [5], 9.1.7.3). For example, the expression "`x < y < z`" is invalid, because the operator "`<`" is non-associative. The same applies to e.g. "`x <= y < z`", because all Arden Syntax comparison operators, including the IS ([5], 9.6) and OCCUR ([5], 9.7) comparison operators, are non-associative and share the same precedence level.

The error handling of the Arden Syntax with respect to the evaluation of expressions at runtime is tailored to the requirements of data-driven event monitoring. MLM execution continues in case of a situation that would create an exception or error message in common all-purpose languages, and the expression evaluates to a NULL value instead. This handling is reasonable for the medical domain, and PLAIN adopted it. For example, the expressions "`2 + true`" (adding a number and a Boolean value) and "`1 / 0`" (division by zero) are accepted by an Arden Syntax compiler, but are evaluated to NULL at runtime. Consequently, it may occur that a nested expression always returns NULL. To give an example, the expression "`UPPERCASE SINE <arg>`" returns NULL for any elementary argument; SINE returns NULL in case the argument is not a number; UPPERCASE returns NULL in case the argument is not a string.

2.3. Use of parentheses

The designers of the Arden Syntax intended to build on a pronounced resemblance to natural language so "*that clinicians will be able to read and understand Arden Syntax knowledge bases with little training*" [1]. Thus, keywords may have an operator-individual arrangement, and parentheses around operands are not generally required. This enables easy-to-understand expressions such as in example (1I), which resemble natural language more closely than alternative definitions predicated on nested parentheses. For comparison, example (1K) outlines how Example (1J) would look like if parentheses were mandatory. Example (1L) illustrates what expressions would look like in a conventional approach with a function-like operator syntax.

The specification is slightly vague regarding the use of parentheses, insofar as it merely states that "*Parentheses are not required*" in case of aggregation operators, and that "*Multiple aggregation and transformation operators may be placed in an expression without parentheses*" ([5], 9.12.1). The specification provides, however, several examples based on other operator types where expressions are nested without the use of parentheses. The only case in which parentheses are explicitly required refers to consecutive arithmetic operators. For example, the expression "`3 + -4`" must be transformed to "`3 + (-4)`" ([5], 9.9.4).

```
(1M)          MAXIMUM OF crea_lab, crea_bga
(1N)          MAXIMUM OF (crea_lab, crea_bga)
```

Although valid syntax does not generally require parentheses around operands, they can be used in order "*to force a different order of execution*" ([5], 9.1.8). For example, if a maximum creatinine value is required from two complementary data sources, such as a central laboratory and a blood gas analyzer, expression (1M) would be inappropriate, because the MAXIMUM operator ([5], 9.12.10) has a higher precedence than the COMMA operator ([5], 9.2.1), which joins both lists of creatinine values. The intended behavior of joining both lists first, then extracting the maximum value, can be enforced via parentheses as shown in expression (1N). This illustrates that an Arden Syntax knowledge engineer must be aware of operator precedence.

3. Methods

Our study started with a detailed investigation of the grammar provided in annex A1 of the specification, to investigate its encoding approach and internal structure. We focused on the expression level and analyzed the hierarchy of precedence levels and corresponding symbols. We used version 2.8 of the Arden Syntax [5], because the PLAIN prototype does not include fuzzy logic constructs introduced in the subsequent version 2.9 [15–17]. However, the parts of the grammar related to the subject of this investigation remained the same in subsequent versions. To examine the extent of and the reasons for the unexpected syntax errors, we laid down a definition of the effect we refer to. As described above, some specific patterns of nested expressions result in a syntax error unless certain operands are parenthesized. We refer to such cases as *anomalies*.

Definition. An anomaly is a specific expression pattern that results in a syntax error and fulfills the following conditions: the operands are not parenthesized, there is no consecutive use of non-associative operators, and there is no consecutive use of arithmetic operators.

```
(2A)        SUBLIST - <expr> ELEMENTS FROM <expr>
(2B)        ATTRIBUTE <expr> FROM FIRST <expr>
(2C)        ROUND SORT <expr>
```

An anomaly can be seen as an expression pattern that results in a counterintuitive syntax error. It is a pattern insofar as an anomaly does not depend on the specific value of an operand. In example (1B) any first operand of SUBLIST that starts with a minus sign would result in a syntax error, such as "`-2`", "`-3`", or "`-(log x + sine y)`". In this specific example, the first operand can be abstracted to "`- <expr>`". Note that the minus sign is, just like in most programming languages, a separate operator ([5], 9.9.3). Consequently, any expression that underlies the pattern in example (2A) is an anomaly. Analogously, example (1F) can be abstracted to the anomaly (2B), and example (1H) can be abstracted to the anomaly (2C). Anomalies always result in syntax errors, for any valid sub-expression `<expr>`.

Based on this definition, we created two complementary utilities to analyze the distribution of anomalies in an extensive set of nested Arden Syntax expressions. The first utility, called the *parser*, determines the actual syntactic correctness of an expression pattern. It was implemented by means of an open source parser generator, using the Arden Syntax grammar from the specification. This required some minor adjustments to the input format of the grammar, albeit without any alteration of its inherent tree structure. We restricted parsing to the expression level of the grammar and designed the utility to process only one single expression at a time, returning "OK" for syntactic correctness, "ERROR" in case of a syntax error, and "UNKNOWN TOKEN" in case of undefined lexical inputs. On the syntactic level of the Arden Syntax all atomic expressions are equivalent, thus we abstracted from variable names, specific literals such as numbers or string constants, and their specific data types. Consequently, a specific expression such as "`glucose IS LESS THAN 50`" can be abstracted to "`<atom> IS LESS THAN <atom>`". For automatic testing, we pragmatically used the single character token "`a`" as a replacement for atoms. To simplify interactive expression testing via the parser utility for human users, we introduced four additional tokens: "`num`" for any numeric literal, "`str`" for any string literal, "`var`" for any variable name, and "`atom`" for any atomic expression, including numbers, strings, and variable names. We integrated a minimalist tokenizer, which expects all tokens (case insensitive) to be separated by blanks. In this approach, example (1B) can be checked for syntactic correctness via the following invocations of the parsing tool (both invocations are equivalent; the second one is the user-friendly version):

```
PARSE "SUBLIST - a ELEMENTS FROM a"
PARSE "SUBLIST - num ELEMENTS FROM var"
```

The second utility, called the *composer*, creates an extensive set of nested Arden Syntax expression patterns, together with the expected syntactic correctness. This utility can be compared to an MLM author who is familiar with non-associativity but who is unaware of the anomalies. The composer permutes operators and hierarchically nests them, without parenthesizing any argument. To give an example, processing the logical OR operator ([5], 9.4.1) on the top-level and the logical NOT operator ([5], 9.4.3) on the sub-level creates the following patterns:

```
NOT a OR a
a OR NOT a
NOT a OR NOT a
```

We specified a straightforward input file format providing a syntax description of the 165 operators in version 2.8, together with their specific precedence level and associativity, separated by a semicolon. For example, the entry

```
21; SUBLIST %s ELEMENTS{ STARTING AT %s} FROM %s;
   RIGHT;
```

refers to the SUBLIST operator, which features an optional STARTING AT infix to specify a start position, as indicated by the curly braces part. The number 21 denotes the precedence level, the token "%s" is a placeholder for sub-level expressions, the term RIGHT marks this operator as right-associative. If a non-associative operator is nested within itself, or if different non-associative comparison operators that share the same precedence level are nested, the composer determines that a syntax error is expected from the parser. To do this correctly, the composer must consider operator precedence, since the consecutive use of operators in nested expressions depends on the order of evaluation. This can be illustrated by example (2D), which shows an expression composed of PLUS and WHERE. The precedence of PLUS is higher than that of WHERE, so the evaluation order of this expression is as shown in example (2E), constituting a case of consecutive use of the WHERE operator. Because WHERE is non-associative, the composer assigns an expected syntax error. Example (2F) shows an expression similar to (2D), but with a COMMA operator instead of PLUS. COMMA has a lower precedence than WHERE, so the evaluation order is as shown in (2G), not constituting a case of consecutive use of WHERE. Thus, a syntax error is not expected. When defining the expected correctness, the composer also accounts for the special case of consecutive arithmetic operators as described above. In all other cases, the composer assigns that an error is not expected.

```
(2D)  a WHERE a + a WHERE a
(2E)  a WHERE (a + a) WHERE a  → SYNTAX ERROR EXPECTED
(2F)  a WHERE a, a WHERE a
(2G)  (a WHERE a), (a WHERE    → SYNTAX ERROR NOT
      a)                         EXPECTED
```

Based on both utilities, we performed an extensive analysis by automatically processing the complete set of expression patterns with a depth of two for the Arden Syntax versions 1, 2, 2.8, and the PLAIN prototype. For each pattern, we automatically compared the actual correctness, determined by the parser, to the expected correctness, assigned by the composer. We stored any detected anomalies into a set of text files, grouped by the specific operator on the top level of an expression pattern.

4. Results

The Arden Syntax grammar, provided in annex A1 of the specification in Backus-Naur form, comprises four distinct levels that correspond to the structure of MLMs.

- **Frame-like structure:** The uppermost level addresses the frame-like MLM structure consisting of categories and slots.
- **Statements:** The next lower level describes the programming language statements within the structured slots EVOKE, DATA, LOGIC, and ACTION. Each of them contains a block of statements, whereas some statements are only allowed within specific slots. A READ statement, to access data items from the patient record, is only allowed within the DATA SLOT; a WRITE statement, to send an alert to a clinical user, is only allowed within the ACTION slot. Thus, the statement level has four different start symbols (`<evoke_block>`, `<data_block>`, `<logic_block>`, `<action_block>`).
- **Expressions:** In contrast to the statement level, the expression level of the Arden Syntax is uniform for all slots and statements (the EVOKE slot, however, does not refer to this level). Operator associativity and precedence are encoded using 54 distinct nonterminal symbols and 23 precedence levels.
- **Lexical constructs:** The lowest level addresses the lexical constructs, such as the requirements for a valid variable name.

Table 1 shows the results of the analysis performed by means of the two complementary utilities. For Arden Syntax version 2.8, permuting and nesting 165 operators resulted in 76,533 expression patterns. 18,978 (24.8%) were identified as anomalies. The percentage of anomalies has been growing across the examined Arden Syntax versions, from 6.4% in Version 1 to 24.8% in version 2.8. PLAIN was

Table 1
Results from the analysis of anomalies.

Arden Syntax Version	1	2	2.8	PLAIN
Operators	100	132	165	156
Expression patterns	22,657	36,783	76,533	54,021
Anomalies	1445	4581	18,978	0
Percentage of anomalies	6.4	12.5	24.8	0
Grammar conflicts	0	0	114	0

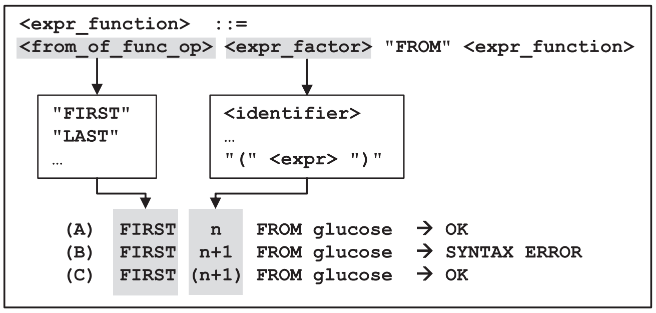
shown to be free from anomalies. The distribution of the recorded anomalies across the 165 operators was very heterogeneous. The minimum was the unary COMMA ([5], 9.2.2) with three anomalies. The maximum was the SUBLIST operator with 1664 anomalies (8.8% of all detected anomalies). 11,785 anomalies (62.1%) were related to operators that make use of the keywords FROM and/or USING.

The vast majority of the detected anomalies have no practical relevance insofar as they are never used in actual knowledge encoding (see discussion). The proportion of those anomalies that have practical relevance cannot be calculated automatically, because this would presuppose an automatic decision whether an expression pattern is relevant or not, which is not possible. A very rough estimate can be provided based on a manual screening of some samples from the files created by the composer. For example, the ATTRIBUTE FROM operator, used in Example (3B), is affected by 411 anomalies, but only 14 have potential practical relevance. Multiple other samples did not contain any relevant patterns at all. We assume that the proportion of anomalies with any practical relevance is in the lower double-digit range, considerably below one percent of the 18,978 recorded anomalies.

A manual analysis of the grammar rules related to a selection of anomalies revealed their reason. Operator precedence and associativity are implicitly encoded by means of separate nonterminal symbols for each precedence level. Examples of such nonterminal symbols are `<expr_or>` (precendece level 7), `<expr_and>` (level 8), and `<expr_not>` (level 9) for the logical operators ([5], 9.4). An anomaly is created whenever the precedence level of an actual operand is lower than that of the corresponding nonterminal symbol in the grammar. To refer to the initial examples (1G) and (1H): The SORT operator and its operand (`<expr_sort>`) refer to level 2. The ROUND operator and its operand refer to level 21. Thus, in case of example (1H), the operand, formed by the sub-expression "`SORT <expr>`" refers to a lower level than the ROUND operator. This results in a syntax error, unless "`SORT <expr>`" is parenthesized. The precedence level of a specific nonterminal symbol is not always obvious and may require a deeper understanding of the grammar.

The explanation of example (1B) is that the first operand of SUBLIST refers to `<expr_factor>` (level 22), while the operand "`-2`" as a case of unary minus refers to `<expr_plus>` (level 13). Fig. 1 shows another illustrative example in the form of a grammar rule for transformation operators, such as FIRST FROM (see 9.14.4 of the specification), and three example expressions (A), (B), and (C). The first operand of the FIRST FROM operator refers to `<expr_factor>` (level 22). In (A), the first operand "`n`" is an atom, referring to the highest precedence level `<expr_factor_atom>` (level 23). Thus, parentheses are not required. In (B), the first operand is the expression "`n + 1`". The binary "`+`" operator ([5], 9.9.1) refers to `<expr_plus>` (level 13), which has a lower precedence than `<expr_factor>`. Thus, a syntax error is reported, unless the operand is parenthesized as shown in (C).

Parenthesized expressions are on the highest possible precedence level `<expr_factor_atom>`. This explains why parenthesizing a specific operand unfailingly avoids a syntax error in case of an anomaly. Anomalies occur if at least one operand is a non-atomic expression and



<expr_function> ::=
<from_of_func_op> <expr_factor> "FROM" <expr_function>
"FIRST"
"LAST"
…
<identifier>
…
"(" <expr> ")"
(A) FIRST n FROM glucose → OK
(B) FIRST n+1 FROM glucose → SYNTAX ERROR
(C) FIRST (n+1) FROM glucose → OK

Fig. 1. Conditions for the occurrence of an anomaly.

the above conditions with respect to the precedence levels are fulfilled. Another contribution to the occurrence of anomalies is the existence of grammar conflicts (parsing ambiguities) in the expression grammar of later versions. The parser generator reported 114 conflicts on the expression level of version 2.8, many of them related to optional suffixes (USING, AT, STARTING AT) supported by several operators.

5. Discussion

The Arden Syntax standard originated in the late 1980s [18] and has been constantly evolving during a series of reviews [19]. Changes in the standard have been carefully designed, and the Arden Syntax constitutes a proven technology that has been integrated into a number of commercial clinical information systems. A calculated overall proportion of 24.8% anomalies may create a highly misleading impression and thus requires a careful discussion. The Arden Syntax has been tailored to data-driven event monitoring functions. The typical expression patterns in this specific use case do not include any anomalies. Moreover, anomalies generally presuppose expression nesting, and since the decision logic of a typical MLM is relatively straightforward, nesting is not necessarily required. Consequently, a knowledge engineer may create large sets of MLMs without ever facing this issue. Moreover, hardly any of the detected anomalies have practical relevance with respect to knowledge representation. For example, "`SINE UPPERCASE <expr>`" has no practical relevance at all, because it always results in a NULL value. An MLM author would hardly ever intend to calculate the sine of an uppercase string. Another point is that recent Arden Syntax versions, which are more strongly affected by anomalies, may not have been widely distributed yet.

Anomalies are not accepted by an Arden Syntax compiler, thus they cannot result in false calculations or even wrong clinical decisions. To an experienced Arden Syntax knowledge engineer, if he ever faces them at all, they are therefore little more than a cosmetic problem. They can easily be sidestepped by parenthesizing the specific operand. Thus, there is no urgent need to systematically eliminate anomalies from the standard. A more important question may be whether they impede knowledge engineering, which would undermine its design goal of being easily accessible for clinicians. If so, a partial adjustment of the grammar may facilitate the acceptance and dissemination of Arden Syntax in clinical settings. At a workshop held by the German Society of Medical Informatics in April 2016, the implications of anomalies have been discussed. Most participants pointed out that their occurrence may confuse potential knowledge engineers during their initial experiences with creating MLMs. It has been long established that the difficulties of finding capable domain experts that are willing to encode medical knowledge constitutes "*the biggest bottleneck in the development of expert systems*" [20], so beginners should not be confused by counterintuitive syntax error messages. From this point of view, even those anomalies that have no practical relevance may be seen as problematic, insofar as a beginner might try irrelevant expressions to gain experience with the language constructs. For example, it may be confusing that an expression such as "`UPPERCASE SINE <expr>`" is processed, while "`SINE UPPERCASE <expr>`" (same operators, different order) causes an error.

In our opinion, the Arden Syntax should be free from any anomalies, and a revision of the grammar should be considered. Throughout the evolution of the Arden Syntax, great importance was laid on backward compatibility, even when fundamentally new functionality such as objects [21] or fuzzy logic [17] was introduced. Removing anomalies from the standard would necessarily break with backward compatibility. The actual extent of a revision, however, would be limited insofar as the majority of the operators would remain unchanged. Moreover, the standard might considerably benefit from an easier maintenance, as explained below.

5.1. Anomalies in contrast to unexpected evaluation order

The anomalies described in this paper are to be strictly distinguished from unexpected behavior in terms of the order of evaluation. An anomaly is never processed by an Arden Syntax environment, but rejected as a syntax error at compile time, despite valid sub-expressions. Anomalies may seem rather unpredictable to a knowledge engineer, unless he is familiar with specific details from the expression grammar of the Arden Syntax, which will rarely be the case. This issue appears to be yet unaddressed in the specification and the literature.

An unexpected evaluation order, in contrast, never results in a syntax error, but in an unexpected calculation result during runtime. In case of nested expressions without parentheses, the evaluation order is determined by the operator precedence as stated in annex A4 of the specification. In some cases, this order may be difficult to understand for a less experienced knowledge engineer who is not entirely familiar with the operator precedence of the Arden Syntax. Hripcsak et al. had already addressed a potentially unexpected evaluation order in Section 6.6 of the rationale for the Arden Syntax [1]. More recently, Jung et al. had referred to this matter in the context of ArdenML [22], where an XML structure explicitly determines the evaluation order, thus avoiding an unintended order. On the one hand, a programming language that does not enforce parentheses around arguments enables easy-to-understand expressions that are close to natural language. On the other hand, such a language presupposes adequate skills and attention on the side of the knowledge engineer if expressions are nested. In case he should be unsure of the operator precedence, he can make use of parentheses. A potentially unexpected evaluation order can hardly be avoided, because it depends on the skills of the knowledge engineer. Anomalies, in contrast, could be completely avoided, provided that the grammar is revised, because they result from technical aspects of the design.

The Arden Syntax supports expressions of arbitrary complexity that do "*not require parentheses to force the obvious meaning*" [1]. Hripcsak et al. critically discuss their own design, declaring that keeping an overview of multiple levels of operator precedence may be difficult, and that "*keeping only the most obvious precedences - simple arithmetic and Boolean operations and requiring parentheses elsewhere - might have been a better choice*" [1]. In our opinion, the current design of the Arden Syntax realizes the intended advantages in that it does enable easy-to-understand expressions as in example (1I). Anomalies may impede the easy comprehensibility to a certain extent, and thus should be eliminated.

5.2. Recommendations for grammar adjustments

Our recommendations for an alternative grammar are based on the experimental PLAIN language. Although it is strongly influenced by the Arden Syntax and implements more than 80 percent of its operators, PLAIN is not intended primarily for knowledge representation, but for clinical algorithms in general. The greater complexity of program code, which is reflected, among other things, in more frequent and deeper nesting of expressions, requires stricter formalization. This contributed to the decision to replace the specification-compliant grammar by a complete rewriting. The PLAIN expression grammar itself is strongly influenced by the experimental, not fully specification-compliant Arden Syntax implementation ARSEN/IC, which has been used for some previous projects [6,23,24], and has been conceived for research on possible extensions and alternative implementation approaches.

The main difference between the standard-compliant grammar and the experimental grammar lies in the fundamental encoding approach. The Arden Syntax grammar encodes operator precedence and associativity by means of distinct nonterminal symbols, such as <expr_or>, as shown in Fig. 2(A). Fig. 2(B1) and (B2) show the corresponding details from the PLAIN grammar, using the input grammar format of the specific open source parser generator. It uses a single nonterminal symbol <ex> for any operand and controls precedence and associativity solely by precedence declarations such as %left, as illustrated in Fig. 2(B2). The use of nonterminal symbols is thus reduced to a minimum. This encoding approach points out some problem areas related to anomalies. For example, in this approach there is a conflict between the FIRST and the FIRST FROM operator. The Arden Syntax grammar circumvents this conflict by the operand of FIRST referring to <expr_function>, while the first operand of FIRST FROM refers to <expr_factor> (see [5], page 124). This avoids the conflict, but simultaneously creates the anomaly illustrated in Fig. 1. With the alternative approach, the conflict cannot be avoided by choosing different non-terminal symbols, since there is only one symbol for any precedence level. The conflict can only be resolved by a minor modification of the language, which also eliminates the anomaly (see also the recommendations in the supplementary material).

The results in Table 1 show that PLAIN is not affected by anomalies. Besides changing the encoding approach as outlined in Fig. 2(B1) and (B2), this also required adjusting several operators, such as moving optional suffixes to an infix position, if possible. PLAIN does not support the FROM operator ([5] 9.10.4), as the token FROM is also used in 15 other operators, contributing to multiple issues. The FROM operator can be dispensed with anyway, because it is merely an alias for the AFTER operator, which has exactly the same semantics. Moreover, a few operators, such as SORT and ADD, have been transformed to statements. Such language adjustments necessarily result in a loss of backward compatibility with earlier Arden Syntax versions. However, the planned next version 3.0 of the Arden Syntax will also break with backward compatibility.

A further recommendation refers to the Arden Syntax specification. The instructions for the use of parentheses should be clarified. We would recommend that parentheses can generally be omitted and that they are used exclusively to control the evaluation order Consequently, PLAIN does not enforce parentheses in case of consecutive arithmetic operators. In addition, all non-associative operators have been changed to left-associativity or right-associativity (see the example grammar in the supplementary material).

5.3. Migration of existing MLMs

Only minor syntactical changes are required for migrating existing MLMs to the proposed experimental grammar. The annex of specification 2.8 contains eight sample MLMs X4.1 to X4.8. The only changes required in these MLMs are illustrated in Fig. 3 using the example MLM X4.6. Fig. 3(A) shows a detail from this MLM. Fig. 3(B) shows the required adjustments to this MLM in the form of three additional OF keywords. None of the eight example MLMs requires further changes beyond this use of the OF keyword. In the current version of the

```
(A) <expr_or>  ::= <expr_or> "OR" <expr_and>    | <expr_and>
    <expr_and> ::= <expr_and> "AND" <expr_not> | <expr_not>
    <expr_not> ::= "NOT" <expr_comparison>     | <expr_comparison>
```

```
(B1)  ex ::= ex OR ex .
      ex ::= ex AND ex .
      ex ::= NOT ex .
```

```
(B2)  %left  OR .
      %left  AND .
      %right NOT .
```

Fig. 2. Details from the grammars. (A) shows a detail from the Arden Syntax grammar. (B1) and (B2) show details from the PLAIN grammar; (B1) shows some rules, while (B2) shows the corresponding precedence declarations.

```
(A)  IF pt_is_taking_tms
     AND THE LAST anc IS LESS THAN 1000
     AND THE LAST anc IS LESS THAN THE FIRST anc

(B)  IF pt_is_taking_tms
     AND THE LAST OF anc IS LESS THAN 1000
     AND THE LAST OF anc IS LESS THAN THE FIRST OF anc
```

Fig. 3. Examples of adjustments that would be necessary for migrating existing MLMs. (A) Detail from the example MLM X4.6 in Arden Syntax specification 2.8. (B) Necessary adjustments are limited to the addition of the keyword OF to the operators FIRST and LAST.

experimental grammar, the aggregation operators FIRST ([5], 9.12.12) and LAST ([5], 9.12.11) must use the otherwise optional OF keyword to avoid an ambiguity with regard to the transformation operators FIRST FROM ([5], 9.14.4) and LAST FROM ([5], 9.14.5) which would result in an anomaly. The aggregation operators EARLIEST, LATEST, MINIMUM, and MAXIMUM also have to use the OF keyword to avoid ambiguities with the related transformation operators. However, making the OF keyword mandatory may be only a temporary solution insofar as any other adjustment that prevents the grammar rules of two different operators from starting with two identical symbols eliminates the ambiguity. The eventual solution is a matter of discussion with the clinical users.

The only noteworthy change refers to sorting data structures. Version 2.8 of the standard introduced an optional USING modifier. This feature is well-designed, nevertheless optional suffixes on the expression level result in some anomalies. PLAIN circumvents this problem by shifting the sorting of data structures to the statement level.

```
(4A)    patients := SORT patients USING IT.bednumber;
(4B)    SORT patients USING IT.bednumber;
```

Example (4A) shows a complete Arden Syntax statement that sorts a list of patient objects by their bed number. The sort operator returns a sorted list that is in this specific example assigned to the same variable name. Example (4B) shows the corresponding PLAIN statement, that performs an in-place sorting of the list. Both approaches have their specific advantages and are basically equivalent. One of the advantages of (4B) is that there are no anomalies, since statements cannot be nested arbitrarily, unlike expressions. However, there are several other potential solutions, such as transforming the unary SORT operator to a binary operator like ORDER BY, so the eventual solution is a matter of discussion.

5.4. Advantages regarding maintenance

In addition to eliminating the anomalies, an alternative expression grammar such as our proposed approach could simplify maintenance and further development, as it substantially reduces complexity. The standard-compliant expression grammar has grown complex over time and now utilizes 54 different nonterminal symbols, 23 of them to encode precedence and associativity. The experimental expression grammar, in contrast, is a clearly arranged flat list, with one rule for each operator, and only six different nonterminal symbols. For extensive expression grammars this is a more comfortable approach, because it avoids complex coding of precedence levels using a multitude of different non-terminal symbols, which can make it difficult to keep an overview. The alternative approach allows for a clearer coding and an easier resolution of potential ambiguities. Moreover, the grammar does not have to be adapted in its structure for a possible change of the operator's precedence (see chapter three in [14]). Besides that, the experimental approach could facilitate the implementation of operator-specific characteristics, which may further increase the desired resemblance to natural language. This aspect may be discussed in another publication. However, a disadvantage of our proposed approach is that it is less flexible in terms of implementation. While the original Arden Syntax grammar can be implemented with LALR(1) as well as LL(1) parser generators [1], our suggested alternative is limited to LALR(1).

6. Conclusion

The Arden Syntax standard is affected by anomalies on the expression level. Most of them have no practical significance with respect to knowledge engineering. Anomalies cannot result in false calculations or decisions, simply because they are not processed at all. Thus, one may deem them merely a cosmetic problem. However they may be potential points of confusion for MLM authors. We therefore recommend an adjustment of the grammar similar to the alternative approach described above. Such a revision would not only eliminate the anomalies, but could also facilitate maintenance. A prototype of the experimental PLAIN language for clinical algorithms already uses such a revised Arden Syntax grammar.

Conflict of interest statement

The authors declared that there is no conflict of interest.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.jbi.2018.05.008.

References

[1] G. Hripcsak, P. Ludemann, T.A. Pryor, O.B. Wigertz, P.D. Clayton, Rationale for the Arden Syntax, Comput. Biomed. Res. 27 (1994) 291–324.
[2] G. Hripcsak, P.D. Clayton, R.A. Jenders, J.J. Cimino, S.B. Johnson, Design of a clinical event monitor, Comput. Biomed. Res. 29 (1996) 194–221.
[3] S. Kraus, I. Castellanos, D. Toddenroth, H.-U. Prokosch, T. Burkle, Integrating Arden-Syntax-based clinical decision support with extended presentation formats into a commercial patient data management system, J. Clin. Monit. Comput. 28 (2014) 465–473, http://dx.doi.org/10.1007/s10877-013-9430-0.
[4] S. Kraus, Generalizing the Arden Syntax to a common clinical application language, Stud. Health Technol. Inform. 247 (2018) 675–679.
[5] Health Level Seven International: The Arden Syntax for Medical Logic Systems, Version 2.8, 2012.
[6] S. Kraus, M. Enders, H.-U. Prokosch, I. Castellanos, R. Lenz, M. Sedlmayr, Accessing complex patient data from Arden Syntax Medical Logic Modules, Artif. Intell. Med. (2015), http://dx.doi.org/10.1016/j.artmed.2015.09.003.
[7] G. Hripcsak, Writing Arden Syntax medical logic modules, Comput. Biol. Med. 24 (1994) 331–363.
[8] M. Gietzelt, U. Goltz, D. Grunwald, M. Lochau, M. Marschollek, B. Song, K.-H. Wolf, ARDEN2BYTECODE: a one-pass Arden Syntax compiler for service-oriented decision support systems based on the OSGi platform, Comput. Methods Programs Biomed. 106 (2012) 114–125, http://dx.doi.org/10.1016/j.cmpb.2011.11.003.
[9] H.C. Karadimas, C. Chailloleau, F. Hemery, J. Simonnet, E. Lepage, Arden/J: an architecture for MLM execution on the Java platform, J. Am. Med. Inform. Assoc. 9 (2002) 359–368.
[10] K. Fehre, H. Mandl, K.-P. Adlassnig, A fuzzy Arden syntax compiler, Austrian Computer Society, Vienna, 2010, pp. 207–212.
[11] R.A. Kuhn, R.S. Reider, A C++ framework for developing Medical Logic Modules and an Arden Syntax compiler, Comput. Biol. Med. 24 (1994) 365–370.
[12] G. Hripcsak, J.J. Cimino, S.B. Johnson, P.D. Clayton, The Columbia-Presbyterian Medical Center decision-support system as a model for implementing the Arden Syntax, Proc. Annu. Symp. Comput. Appl. Med. Care 248–52 (1991).
[13] T.A. Mogensen, Introduction to Compiler Design, Springer-Verlag, London Limited, London, 2011.
[14] J. Levine, Flex & Bison, O'Reilly Media Inc, Sebastopol, 2009.
[15] K.-P. Adlassnig, K. Fehre, A. Rappelsberger, Fuzzy-Arden-Syntax-based, vendor-agnostic, scalable clinical decision support and monitoring platform, Stud. Health Technol. Inform. 216 (2015) 1111.
[16] T. Vetterlein, H. Mandl, K.P. Adlassnig, Processing gradual information with Fuzzy Arden syntax, Stud. Health Technol. Inform. 160 (2010) 831–835.
[17] T. Vetterlein, H. Mandl, K.-P. Adlassnig, Fuzzy Arden Syntax: a fuzzy programming language for medicine, Artif. Intell. Med. 49 (2010) 1–10, http://dx.doi.org/10.1016/j.artmed.2010.01.003.

[18] G. Hripcsak, O.B. Wigertz, P.D. Clayton, Origins of the Arden Syntax, Artif. Intell. Med. (2015), http://dx.doi.org/10.1016/j.artmed.2015.05.006.
[19] R.A. Jenders, K.-P. Adlassnig, K. Fehre, P. Haug, Evolution of the Arden Syntax: key technical issues from the standards development organization perspective, Artif. Intell. Med. (2016), http://dx.doi.org/10.1016/j.artmed.2016.08.001.
[20] J.R. Olson, H.H. Rueter, Extracting expertise from experts: methods for knowledge acquisition, Expert Syst. 4 (1987) 152–168, http://dx.doi.org/10.1111/j.1468-0394.1987.tb00139.x.
[21] R.A. Jenders, R. Corman, B. Dasgupta, Making the standard more standard: a data and query model for knowledge representation in the Arden syntax, AMIA Annu. Symp. Proc. (2003) 323–330.
[22] C.Y. Jung, J.-Y. Choi, S.J. Jeong, K. Cho, Y.D. Koo, J.H. Bae, S. Kim, Transformation of Arden Syntax's medical logic modules into ArdenML for a business rules management system, Artif. Intell. Med. (2016), http://dx.doi.org/10.1016/j.artmed.2016.03.005.
[23] S. Kraus, I. Castellanos, M. Albermann, C. Schuettler, H.-U. Prokosch, M. Staudigel, D. Toddenroth, Using Arden Syntax for the generation of intelligent intensive care discharge letters, Stud. Health Technol. Inform. 228 (2016) 471–475.
[24] S. Kraus, C. Drescher, M. Sedlmayr, I. Castellanos, H.-U. Prokosch, D. Toddenroth, Using Arden Syntax for the creation of a multi-patient surveillance dashboard, Artif. Intell. Med. (2015), http://dx.doi.org/10.1016/j.artmed.2015.09.009.