EGADSS: A Clinical Decision Support System for use in a Service-oriented Architecture

by

Glen McCallum

Bachelor of Science, University of Victoria, British Columbia, 2004

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCES

in the Department of Computer Science

© Glen McCallum, 2006

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
EGADSS: A Clinical Decision Support System for use in a Service-oriented Architecture

by
Glen McCallum
Bachelor of Science, University of Victoria, British Columbia, 2004

Supervisory Committee

Dr. Jens H. Weber (Department of Computer Science)
Supervisor

Dr. Daniel German (Department of Computer Science)
Department Member

Dr. Yvonne Coady (Department of Computer Science)
Department Member
Abstract

Almost one in ten of every hospital admission in Canada is caused by our own healthcare system. Steps must be taken to reduce medical errors. One proven way to reduce medical errors is through the use of Clinical Decision Support (CDS) Systems. CDS systems provide point of care alerts to clinicians that are based on pre-defined rules (e.g. if patient allergic to drug then do not administer drug). A significant issue with CDS Systems is that they are very expensive for a hospital to develop and maintain, thereby preventing widespread adoption.

We have designed and implemented EGADSS, the Evidence-based Guideline And Decision Support System. EGADSS is designed for use as part of a service-oriented architecture. By centralising the development and maintenance of a CDS system (possibly by a governing authority), more providers can afford to adopt them. Indeed we have interfaced EGADSS with two clients systems. In addition, EGADSS can easily be adopted by other EMRs.

We also evaluated the suitability of Arden Syntax, HL7's Clinical Guideline Encoding standard, for use in a service-oriented CDS environment. We identify several language features that should be standardised.

Most importantly, we show how to build a CDS system for use in a service-oriented architecture. EGADSS has been tested with 5 test guidelines and several patient profiles to verify that it behaves correctly.
Table of Contents

Abstract ...................................................................................................................... iii
Table of Contents ..................................................................................................... iv
List of Tables ........................................................................................................... vi
List of Figures ......................................................................................................... vii
Acknowledgments ................................................................................................... viii
Dedication ................................................................................................................. ix
Chapter 1 Introduction ............................................................................................ 1
  1.1 Service-Oriented Architecture and Service-based Decision Support ......... 2
  1.2 Traditional CDS Architecture ....................................................................... 3
    1.2.1 Implementation of the CDS System ...................................................... 4
  1.3 Challenges of Service-based Decision Support ........................................... 6
    1.3.1 Non-Standardized Language Features ................................................ 6
    1.3.2 Lack of Formal Semantic Description ................................................... 7
    1.3.3 Implementing the Service-based Decision Support System .................. 9
Chapter 2 Clinical Background .............................................................................. 10
  2.1 Adverse Events: Something needs to be done .......................................... 10
  2.2 Clinical Decision Support: Improving Care .............................................. 11
  2.3 Generic CDS System Architecture ............................................................. 12
  2.4 The HL7 Guideline Representation Standard: Arden Syntax .................... 13
  2.5 Non-Standard Guideline Languages – Related Work ................................ 17
    2.5.1 PRODIGY .................................................................................... 17
    2.5.2 PROforma ...................................................................................... 17
    2.5.3 GUIDE ....................................................................................... 18
    2.5.4 EON/SAGE ................................................................................... 18
    2.5.5 GLIF ............................................................................................ 18
Chapter 3 Non-Standardized Language Features ................................................. 21
  3.1 Curly Braces ............................................................................................... 21
    3.1.1 EGADSS Patient Summary Document .............................................. 21
    3.1.2 Standards-based Queries ................................................................. 22
    3.1.3 EGADSS Recommendations Document .......................................... 23
  3.2 Evoke Section ............................................................................................. 24
  3.3 Non-isolated Query Logic ............................................................................ 25
  3.4 Implicit Primary Times ................................................................................. 26
  3.5 C-style Output Formatting .......................................................................... 26
Chapter 4 Formal Semantic Description ................................................................ 28
  4.1 Specifying Computer Languages .................................................................. 28
    4.1.1 Syntax ............................................................................................. 28
    4.1.2 Semantics ........................................................................................ 29
  4.2 Requirements on a Guideline Language Semantic Description Formalism (GLDSF) ......................................................................................................................... 30
  4.3 Advantages of a Formal Semantic Description ............................................ 32
    4.3.1 GLDSF should facilitate precise and unambiguous semantic descriptions... 32
List of Tables

Table 1. Guideline Language Comparison Matrix .......................................................... 20
Table 2. EGADSS Patient Summary Document Sections and Codes ............................... 22
Table 3. EGADSS Recommendation Document and Element Descriptions .................. 23
Table 4. Comparison of Formal Semantic Description Approaches ............................... 46
Table 5. BNF Grammar Transformation Rules .............................................................. 47
Table 6. Example Test Generator Input ........................................................................ 77
Table 7. Syntactically Valid Expressions Generated from Example Input ....................... 77
Table 8. New Input Expressions for next Test Generation Cycle .................................... 77
Table 9. Testing Diabetes A1C CIG with Patient Profiles ............................................. 80
List of Figures

Figure 1. Generic CDS System Architecture.......................................................... 13
Figure 2. Flowchart Representation of Diabetes A1C Guideline................................ 15
Figure 3. MLM Representation of Diabetes A1C Guideline.................................... 16
Figure 4. XPath Query Embedded in Arden Syntax MLM..................................... 23
Figure 5. Montages for "AND" expression with the production rule included........... 49
Figure 6. "AND" expression Montages with Synthetic Attributes......................... 50
Figure 7. Example "AND" Expression Montages with Static Semantics................... 51
Figure 8. "And" Expression Montage with Control Flow ASM................................ 52
Figure 9. Complete Montages Description of "And" Expression............................. 53
Figure 10. EGADSS Component Architecture...................................................... 57
Figure 11. UML description of IKnowledgeMgt Interface...................................... 58
Figure 12. UML description of IAudit Interface................................................... 58
Figure 13. UML description of ISysConfig Interface............................................ 58
Figure 14. UML description of ITrans Interface.................................................. 59
Figure 15. GetRecommendations Sequence Diagram.......................................... 60
Figure 16. Sample Arden Syntax to CLIPS Translation......................................... 64
Figure 17. 2-Step Service Interaction Protocol.................................................... 67
Figure 18. N-Step Service Interaction Protocol.................................................... 68
Figure 19. EGADSS with Two GLCompiler Components......................................... 72
Figure 20. Test Suite Generation Cycle............................................................... 76
Acknowledgments

I would like to express my gratitude for my loving and supportive wife, Victoria.

Thanks to my supervisors, mentors and friends Drs. Jens Weber and Morgan Price. This research would not have been possible without them or the rest of the EGADSS team (in order of appearance): Iryna Bilykh, Joel Legris, Emma MacEntee, Amanda Sinclair, Maike Dulk, Brad Barclay, Dmytro Draga and Lindsay Forland.

Many thanks to my supportive friends in the Pervasive Primary Care Informatics Lab: Christina Obry, Yury Bychkov, and Adeniyi Onabajo. May our research ties and friendship continue into the future.

Finally I would like to acknowledge my parents. Volumes cannot hold all that they have taught me.
Dedication

To my late Grandmother
Chapter 1 Introduction

Canada Health Infoway has published their Electronic Health Record Solution (EHRS) Blueprint [1]. An EHRS differs from other definitions of Electronic Health Records in that it includes people, organizations, process, systems, and standards that work together to enable clinical data exchange and improved care. This specification outlines a service-oriented architecture (SOA) for a national healthcare information network. In this architecture, software services that have been localized (integrated) in the past can now be service-based and accessible by a number of organizations. Clinical Decision Support (CDS) Systems fall into this category. These are referred to in the Blueprint as Point of Service Applications.

In the past, CDS Systems have either been completely detached from or completely integrated with single healthcare information systems. The introduction of the EHRS SOA provides opportunity for a new type of CDS system, the service-oriented decision-support system. This new architecture provides some advantages for CDS. For example, a governing authority can assume the role of developing and maintaining the guideline knowledge base that is easily accessed by multiple systems across the jurisdiction. This can encourage standardized care and promote best practices. Alternatively, each organization would have to maintain their own CDS system and knowledge base, an expensive and ongoing investment [2].

Knowledge in the form of clinical practice guidelines is not currently developed and published by any single organization. Different medical organizations publish paper-based guidelines around different areas of care. For example, the BC Centre for Disease
Control publishes Immunization guidelines while the BC Medical Association publishes cancer-screening guidelines. Currently, individual organizations must take these paper-based guidelines (from disparate sources) and encoded them into a machine-interpretable format for use in a local CDS System (Eg. Arden Syntax). A CDS designed for use in a service-oriented architecture could avoid the replication of this step at each site.

The thesis of this work is that extending an existing guideline language with standards for a service-oriented execution model can be used to enable the design and implementation of a service oriented CDS that is adoptable by different EMR client systems.

1.1 Service-Oriented Architecture and Service-based Decision Support

Canada Health Infoway’s EHRS Blueprint [1] defines a Service-oriented Architecture as:

An infrastructure where many N-tier applications are deployed, sharing common software services that are accessible from any user interface. In this environment, any application can access any service, provided the application has the proper security permissions. The greatest strength of a service-oriented architecture is the potential for repeatable rapid development of new applications. It depends on interoperable services for the provision of high-value business logic processing.

It is also helpful to reference Infoway’s definition of a service [1]:

Discrete units of application logic that expose loosely coupled message-based interfaces suitable for being accessed across a network.

We therefore define a service-based clinical decision support system as a discrete unit of clinical decision support application logic that employs a message-based interface and
is accessible over a network and that also acts as a shared common service accessible from any user interface provided the user has proper security permissions.

1.2 Traditional CDS Architecture

The first CDS systems were stand-alone programs that a clinician would have to manually enter information into in order to get recommendations. These systems worked well on targeted domains. An example of such a system was MYCIN [3] originally developed to control nosocomial infections. The downfall of these systems is that they do not fit well into a clinician’s workflow. They require active use; a clinician has to sit down and take time to enter a patient’s data. Often a decision has already been made and documented in a paper record by the time a physician has an opportunity to do this.

The second major type of decision support systems are integrated as part of a much larger clinical record system. Currently, this is the most implemented type of CDS systems in use. These systems have the advantage of being able to intervene at various points during a physician’s workflow. They are more integrated with a clinician’s workflow and can be more effective than stand-alone systems. However, the knowledge in the form of computer-interpretable clinical practice guidelines is almost completely bound to the host record system. As a result of the tight integration, guidelines are tied to the explicit patient data model used by the host system. In addition, these systems often encode guidelines in a proprietary language [4]. Thus sharing guidelines is impossible and maintaining a guideline knowledge base for an extended period of time is often cost-prohibitive for healthcare organizations [2].
1.2.1 Implementation of the CDS System

Evidence-based Guideline and Decision Support System (EGADSS) is the name of our prototype service-based clinical decision support system. It was originally designed as a network-accessed, CDS component providing point-of-care alerts for primary care physicians. It is designed to interface with existing Electronic Medical Records (EMRs, a clinical record system for primary care) via HL7 CDA Document exchange. HL7 CDA documents are XML documents that define structured medical data in a standard way using clinical vocabularies. These documents are exchanged (between EGADSS and an EMR) through a platform-independent web service interface.

As input, EGADSS receives a Patient Summary Document (HL7 CDA). Sections in the Patient Summary Document include demographics, social history, surgical history, active medical problems, and other relevant clinical data (See Section 3.1.1).

EGADSS has a knowledge base consisting of rules written in Arden Syntax. We recognize that this language was originally designed for use in integrated systems. However, it is the only open guideline language that has been recognized as an industry standard. Using a standards-based, freely available formalism was a key requirement for the EGADSS design team. Its integrated origins have provided some design challenges that are described in Chapter 3. Arden Syntax Medical Logic Modules (MLMs) can be written to perform logic on any aspect of data available in the patient summary document. As part of the EGADSS project we have written 5 test MLMs to date. These MLMs represent clinical practice guidelines from the areas of Immunizations and Chronic Disease Management.
After processing a Patient Summary Document, EGADSS returns a clinical recommendations document (HL7 CDA) to the EMR. This document contains patient-specific recommendations as well as requests for additional patient data. The stand-alone XML interface that EGADSS employs abstracts some difficult CDS implementation details from EGADSS, such as how to present the alert to a physician.

There were several high-level requirements placed on EGADSS from the beginning:

1) EGADSS was to be released under the GPL open source license. All components and frameworks that EGADSS used should also be available freely under similar conditions.

2) EGADSS was to be based on open standards wherever possible. It is important that EGADSS user not be encumbered by any third-party licensing fees.

3) EGADSS was to not store or retain any patient information (this requirement led to the requirement that EGADSS be stateless and memory-less).

Randolph categorizes CDS systems into several categories: Alerting, Reminding, Critiquing, Interpreting, Predicting, Diagnosing, Assisting, and Suggesting [5]. Alerting systems monitor a continuous stream of data and alert clinicians to abnormal values or patterns that require attention. Reminding systems notify clinicians of recurring or important tasks. Assisting systems help generate the clinical decision while critiquing systems let the clinician formulate the decision and provide an appropriateness rating based on their decision. Interpreting systems abstract form the clinical data and provide a report to clinicians. Diagnosing decision support systems evaluate systems and consider past history, laboratory values, etc in order to provide a list of possible diagnoses while Suggesting systems also provide information about the optimal decision.
EGADSS could be used as a basis for several of these types of systems, but most appropriately as part of Alerting, Reminding, Assisting, and Diagnosing systems.

1.3 Challenges of Service-based Decision Support

At the time of this project, there is only one openly-standardized guideline encoding language, Arden Syntax [6]. Arden Syntax was adopted as an ASTM standard in 1991. The current version, Arden Syntax 2.1 is now maintained by HL7 and ANSI [6]. Other guideline encoding languages exist and are described in Section 2.5, but Arden Syntax is the only open standard; by default it became the guideline encoding language for EGADSS. Our challenges reflect Arden Syntax's suitability for use in our system. The three major issues we faced were 1) Non-standardized Languages features that are part of Arden Syntax, 2) Lack of a formal Arden Syntax Semantic Description, and 3) The design and implementation of the system itself.

The hardest challenge we faced was the non-standardized language features. These were particularly difficult because there was nothing we could write or build to resolve the situation. Appropriate parties will have to agree upon and publish a standard that we can use in the future.

1.3.1 Non-Standardized Language Features

In order to understand why we face some challenges using Arden Syntax, it is beneficial to consider why Arden Syntax was created and what it is trying to achieve. We take this excerpt from the 2.1 specification [6].

Many obstacles to [clinical guideline] sharing have been identified: disparate vocabularies, maintenance issues, regional differences, liability, royalties, syntactic
differences, etc. This standard addresses one obstacle by defining a syntax for creating and sharing knowledge bases. In addition, the syntax facilitates addressing other obstacles by providing specific fields to enter maintenance information, assignment of clinical responsibility, links to the literature, and mappings between local vocabulary terms and terms in the knowledge base.

Arden Syntax has fields (or slots) that reference information specific to a particular clinical record system. In order to make our CDS System implementation service-oriented we had to standardize the logic in these slot or limit the scope of our system. The most well known example is the “curly braces problem,” which describes the system dependent manner that Arden Syntax MLMs retrieve data elements from a clinical record system. We will discuss our solution to this problem later (See Section 3.1). We present another example here.

1.3.1.1 Evoke Feature

In Arden Syntax there is a section of a MLM called the “Evoke” section that designates the event and/or time that trigger the guideline to execute. Evoke logic can be specified relative to a system event such as “storage_of_digoxin_order.” This statement of logic is totally and completely coupled to a single clinical record system. It contradicts the goal of having a service-based decision support system accessible by any user with proper security permission.

1.3.2 Lack of Formal Semantic Description

The Arden Syntax 2.1 Specification [6] defines language semantics in an informal manner. This informal nature led to the need for interpretation as we built the system. Some examples are described here.
1.3.2.1 Can there exist a negative duration?

In the Arden Syntax Specification there are operations for subtracting one time from another time, resulting in a duration. For example, 2006-02-21 – 2006-01-21 = 1 month. However, there is not restriction that the left-hand side be larger than the right-hand side to prevent a meaningless result such as negative one month. Or could negative one month mean one month ago? Is this a matter of context? The specification is unclear.

1.3.2.2 Duration Unary Operation conflicts with Binary Arithmetic

With respect to durations and arithmetic there are a series of unary functions associated with duration units (years, months, days, hours, minutes, seconds, [6], see Page 50). These functions convert year durations to month durations and convert days, hours, and minutes durations to seconds durations. This simplifies binary comparisons and operations. Conversions between seconds and months are made using a constant and are handled inside the binary operations themselves. Consider this example:

- 2 years > 24 months       (Apply unary “years” function)
- 24 months > 24 months     (Apply binary comparison function)
- False

In another location of the specification [6] (See Page 48), there is an example of subtracting a duration from another duration (3 days – 2 days = 1 day). This could not be input to the binary operation if the unary operation was applied first.
1.3.2.3 Underflows/Overflows

Occasionally, we saw places in the Arden specification where the authors had written, "In case of underflow/overflow return null." This was quite unclear because there was no mention of the magnitude of number that need be supported. Underflows and Overflows can differ from implementation to implementation and our goal is to provide consistent behaviour across different environments. These statements should be qualified and/or clarified.

1.3.3 Implementing the Service-based Decision Support System

The final major issue we faced was designing and building the EGADSS system. The key design elements of the EGADSS Arden Syntax Engine are the overall component-based architecture, the service interaction protocol, and the vendor-agnostic patient data model.

We present the issues of non-standardized language features, lack of formal language support, and implementing the service-based decision support system in more detail in Chapters 3, 4, and 5 respectively. Chapter 2 provides an overview of the clinical decision support domain as well as relevant background information. Chapter 6 presents our Evaluation and Chapter 7 is our Conclusions.
Chapter 2 Clinical Background

The frequency of medical errors is far too high in Canada. Steps must be taken to reduce the rate, particularly of easily preventable errors. Clinical Decision Support is one method that has proven useful in improving care and reducing medical errors. EGADSS is a Clinical Decision Support (CDS) system that could help improve care.

2.1 Adverse Events: Something needs to be done

Canadians are becoming more and more concerned with the quality of care in our healthcare system. A recent study by Drs. Baker and Norton showed that 7.5% of all hospital admissions in 2000 were a result of an adverse event. An adverse event is defined as an "unintended injuries or complication resulting in death, disability, or prolonged hospital stay that arise from health care management" [7]. Informally, one might call it a mistake by a healthcare provider. An example would be delivery of a medication, such as penicillin, to a patient with a known allergy to that medication [7].

7.5% in 2000 equates to 185,000 hospital admissions, out of a total of 2.5 million total admissions. Of these, roughly 70,000 (37% - 51%) are said to be preventable [7].

According to the study, 9250 to 23,750 deaths could have been prevented in Canada [7].

Other countries, performing similar studies, yield similar results. Adverse event rates vary from 2.9% to 16.6% [7]. Part of this variation is due to slight variations in the definition of an adverse event and slight differences in research methods.

The issue of patient safety cannot continue to be ignored. Statistics Canada reports that there were 542 homicides in the year 2000 [8]. It is difficult to imagine that at least 9250 preventable deaths (perhaps 20X the number of homicides) occurred in our healthcare
system during the same period and little has been done to prevent it, but that is what is suggested by the research and is echoed in research from other countries [9].

2.2 Clinical Decision Support: Improving Care

One effort to improve the quality of healthcare and reduce adverse events is the implementation of electronic medical record systems with CDS systems. Much like how a commercial pilot has many instruments that help him diagnose problems and manage flying a plane, a clinician too can benefit from the aid of tools. A CDS system is a system that provides physicians with patient-specific reminders, assessments, or recommendations at the point of care in order to assist in clinical decision making [10]. Several studies have demonstrated that using CDS systems can reduce medical errors.

In a study by Evans et al., a CDS system was used to provide immediate feedback to physicians on the use of antibiotics and other anti-infective agents [11]. The results showed a significant reduction in orders for drugs that patients had reported allergies to, in excess drug dosages, and in antibiotic susceptibility mismatches. In addition, there was a reduction in adverse events caused by anti-infective agents. Moreover, there was a major reduction in the total operating cost of the hospital, partially as a result in the reduction in the length of hospital stays.

Classen et al. used a CDS system to study the timing of antibiotic administration to prevent surgical wound infections [12]. They found it useful in decreasing the adverse event rates (frequency of infections). Moreover, they observed a decrease in costs as a result.

CDS systems are often linked with Computerized Physician Order Entry (CPOE) systems. In 1998, Bates et al. observed a 55 percent reduction in serious medication
errors when a CPOE (with some CDS capabilities) was implemented at a tertiary-care hospital [13].

CDS systems have also been used to improve preventive care. Balas et al. used a CDS for prompting physicians with preventive care reminders [14]. The result was a 13.1% increase in the quality of preventive care.

Overall, Johnston et al. show the impact of CDS systems on quality of care [15]. They showed that CDS systems could improve the adherence of physicians to care standards (best-practice guidelines). In addition, the use of CDSs can improve the rate of uptake by physicians when new evidence is published.

More recent research is addressing the problem of how to successfully deploy a CDS system [16, 17]. The increasing evidence of the effects of CDS systems demonstrates that they improve care.

2.3 Generic CDS System Architecture

In the simplest form of a CDS system is an inference engine where the knowledge base or rules are derived from clinical decision logic (guidelines). The knowledge is complemented by the fact base, which consists of patient specific data. New patient specific recommendation facts are inferred by applying the rules in the knowledge base to the patient data fact base (See Figure 1). We present and describe rule-based decision support systems but there may be other types such as those based on neural networks.
2.4 The HL7 Guideline Representation Standard: Arden Syntax

In the medical community clinical practice guidelines are developed as text-based documents. On one level, they can be thought of as clinical workflows that reference specific elements of patient data at decision points. We present the Diabetes A1C guideline as an example [18]. This version was developed by the BC Guidelines and Protocols Advisory Committee and edited by Dr. Morgan Price, Department of Family Practice, University of British Columbia. Clinician interpretation is essential because published guidelines often reference unclear terms such as "elderly." What does elderly mean? It can very between regions and races. Such terms must be defined by a clinician so that they reference the local patient population.

The following illustrative “Diabetes A1C guideline” is based around the recommendation that an A1C lab test, which gives an indication of a patient’s blood sugar levels over the past 2-3 months, is due in diabetics every three months. If the A1C
result is above a target range of 7%, then the clinician should be reminded and consider changing the current diabetes management (http://www.diabetes.org). Better control of A1C levels corresponds with less diabetic complications such as blindness, heart attacks, amputations, and death. The flowchart representation is shown in Figure 2. The data input consists of: 1) a list of all the types of diabetes a patient has been diagnosed with, and 2) the value, unit, and date of the last A1C lab test.
Figure 2. Flowchart Representation of Diabetes A1C Guideline.

In order for these text-based (illustrated graphically to simplify) guidelines to become useful in providing automated point of care suggestions they must be translated into a machine interpretable language. This is the nature of Arden Syntax; it is a machine-interpretable language to represent clinical decision-making logic.
We show the Diabetes A1C MLM logic section in Figure 3. The full MLM is available in Appendix B.

Diabetes A1C MLM. To simplify the representation below, we have abstracted the ICD-9 and LOINC codes, but they are included in the full MLM. One inference had to be made from the original text-based guideline, if one (or all) test value(s) is not available then the guideline recommends that the patient is due for an A1C test.

```
logic:
   if (not exists diabetes)
      then
         conclude false;
      endif;

   if (((not exists latest_a1c_date) or (not exists latest_a1c_value) or (not exists latest_a1c_unit))
      then
         patient_due := true;
         conclude true;
      endif;

   if ((now - latest_a1c_date) < 3 months)
      then
         if ((latest_a1c_value > 7) AND (latest_a1c_unit = "%"))
            then
               patient_above := true;
               conclude true;
            else
               conclude false;
            endif;
         else
            patient_due := true;
            conclude true;
         endif;
   ;;
```

Figure 3. MLM Representation of Diabetes A1C Guideline.

Arden Syntax evolved out of collaboration among three major systems: the HELP system (LDS Hospital in Salt Lake City), the CARE system (Regenstrief Institute in Indianapolis), and the Arden Syntax System (Columbia Presbyterian Medical Centre in New York). It was intended as a point of care reminder system where a single piece of medical logic, or a single decision, is encoded in a medical logic module (MLM). Many MLMs make up a knowledge base [6].
2.5 Non-Standard Guideline Languages – Related Work

Most of the current research in this area has focused around designing and building a guideline encoding language that will facilitate guideline sharing/reuse by different healthcare organizations. We can see that these projects follow Arden Syntax in the idea that portability is a matter of guideline language standardization, but none has been standardized or widely adopted. There are several groups developing guideline formalisms; five are described here. Table 1 provides a direct comparison.

2.5.1 PRODIGY

PRODIGY was developed by the University of Newcastle and was designed for chronic disease management in primary care. The PRODIGY project uses the Protégé ontology-modeling environment [19] to encode guidelines. As of 2003, two primary care EMR vendors had included PRODIGY components with their EMRs [20].

2.5.2 PROforma

PROforma is from the Advanced Computation Laboratory of Cancer Research in the UK. Unlike all other current projects it takes a logic programming approach, formally based on the R2L language [20]. It has a minimal set of modelling constructs and of all the languages, this one intrigued the EGADSS design team the most due to its adoption and use in real clinical settings. However, licensing conditions made it inappropriate for use in an open-source CDS system.
2.5.3 GUIDE

GUIDE originated at the University of Pavia. It makes use of decision tress and influence diagrams for decision analytics. It also uses Petri nets to implement guideline care flow [20]. Two GUIDE editing-execution environments have been realized and been used in some clinical testing [21].

2.5.4 EON/SAGE

EON consists of a suite of model to represent different parts of a clinical guideline (domain ontology, virtual medical record, roles). The Protégé environment is used for authoring guidelines. EON is currently used in the ATHENA system, which is deployed at Department of Veterans Affairs hospitals across the United States.

The SAGE Project uses the EON guideline language. They investigate decision support factors that interact with the guideline language such as the query interface, terminology mediation, and the act interface. The Domain Model is based on HL7’s Reference Information Model and SNOMED-CT is the chosen clinical vocabulary. Much like EGADSS, SAGE is not in control of the clinical workflow. It responds to opportunities. From our observations of the project, it seems to focus on the guideline authoring environment.

2.5.5 GLIF

GuideLine Interchange Format (GLIF) was a joint venture between Columbia, Stanford, and Harvard Universities (www.intermed.org). It features an HL7 Reference Information Model based patient data model, an object-oriented expression language GELLO, and useful constructs for modeling care flow [22]. Unfortunately, we were
unable to uncover anything more than a conceptual model for the GLIF language. There is no publicly-available concrete implementation Wang et al. designed and built a GLIF Execution Engine (GLEE) [23]. In terms of a language description we were unable to uncover anything more than a conceptual model. This made it impossible to consider for our use. However, the conceptual model is promising and could indeed be the future of guideline languages.
Table 1. Guideline Language Comparison Matrix (Blank cells mean information is not specified in the literature).

<table>
<thead>
<tr>
<th>Guideline Language</th>
<th>Patient Information Model</th>
<th>Medical Concept Model</th>
<th>Formal Syntax</th>
<th>Formal Semantics</th>
<th>Implementation</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prodigy</td>
<td>Self-defined VMR</td>
<td></td>
<td></td>
<td></td>
<td>Two vendor implementations. Used by large numbers of GPs in the UK.</td>
<td></td>
</tr>
<tr>
<td>PROforma</td>
<td>Guideline - EMR direct mapping</td>
<td></td>
<td>BNF</td>
<td>Operational (Finite state machines)</td>
<td>2 Systems in clinical use, both developed by InferMed.</td>
<td>Tallis, Arezzo</td>
</tr>
<tr>
<td>GUIDE</td>
<td>Self-defined VMR</td>
<td>Subset of ICD9-CM and LOINC</td>
<td></td>
<td></td>
<td>Prototype development with clinical evaluation in 4 hospitals, but no ongoing use.</td>
<td></td>
</tr>
<tr>
<td>EON/SAGE</td>
<td>HL7 RIM</td>
<td>SNOMED-CT</td>
<td></td>
<td></td>
<td>ATHENA, used in US Department of Veterans Affairs health centres across the United States</td>
<td></td>
</tr>
<tr>
<td>GLIF</td>
<td>HL7 RIM</td>
<td>UMLS</td>
<td>None</td>
<td>None</td>
<td>GLEE, used in two experimental settings (Israel Primary Care, Columbia University)</td>
<td>Protégé Plugin and sample guidelines</td>
</tr>
</tbody>
</table>
Chapter 3 Non-Standardized Language Features

Arden Syntax evolved out of CDS systems that are integrated with clinical record systems and even after standardization, the use of Arden Syntax still requires the Arden Syntax MLMs to be tightly coupled with a clinical record system. This specific coupling is a result of certain language features not being sufficiently standardized.

We discuss the major instances specific coupling we encountered, specifically problems related to the curly braces, evoke section, non-isolated query logic, implicit primary times, and c-style output formatting. We present our solutions in the cases that we were able to use standards to remove the coupling requirement.

3.1 Curly Braces

MLMs are intended to be portable across different implementations; however, Arden Syntax suffers from a vulnerability known as the “curly braces problem.” It stems from the fact that Arden Syntax must access the local data repository at each implementation. Inside the curly braces, Arden Syntax references local data variables, making these statements not portable without significant modification and testing.

The “curly braces” problem allows Arden Syntax MLMs to contain system specific query logic right. This requires the MLM to have intimate knowledge of the database (data model) structure.

3.1.1 EGADSS Patient Summary Document

We use the HL7 Clinical Document Architecture standard to remove this tight coupling with the host database structure. Clients (clinical record systems) export patient data into
a standardized clinical model, the EGADSS Patient Summary Document, describing the
minimum amount of data needed to determine which guidelines apply to the patient. This
CDA document is divided into structured sections as described in Table 2.

<table>
<thead>
<tr>
<th>CDA Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demographics</td>
<td>Patient gender, birth date, marital status</td>
</tr>
<tr>
<td>Active Medical Problems</td>
<td>ICD-9-CM codes and diagnosis date for all active</td>
</tr>
<tr>
<td></td>
<td>problems</td>
</tr>
<tr>
<td>Past Medical History</td>
<td>ICD-9-CM codes and diagnosis for all past problems</td>
</tr>
<tr>
<td>Surgical History</td>
<td>ICD-9-CM procedure codes</td>
</tr>
<tr>
<td>Current Medications</td>
<td>Health Canada Drug Product Database codes (DINs)</td>
</tr>
<tr>
<td>Allergies and Adverse</td>
<td>ICD-9-CM codes for allergies and adverse reactions.</td>
</tr>
<tr>
<td>Reactions</td>
<td></td>
</tr>
<tr>
<td>Family History</td>
<td>ICD-9-CM codes and diagnosis for family problems</td>
</tr>
<tr>
<td>Immunizations</td>
<td>British Columbia Center for Disease Control (BC CDC) codes</td>
</tr>
<tr>
<td>Social History</td>
<td>ICD-9-CM codes for such problems as smoking, etc</td>
</tr>
<tr>
<td>Recent Laboratory Data</td>
<td>LOINC codes for laboratory test</td>
</tr>
</tbody>
</table>

3.1.2 Standards-based Queries

A query method is still necessary in order to reference our client-independent patient
data model. Extensions to the Arden Syntax (GEL [18], GELLO [19]) provide query
expressions, however, neither of these is standardized. In addition, neither integrates well
with the underlying technology that the CDA is built on, XML. In this context, two query
expression languages make good candidates. XPath [24] and XQuery [25] are built for
querying XML data sources and are both W3C standards. XQuery provides rich
expression structures for performing many more operations than just data retrieval. It is
preferable to limit guideline logic to the “Logic” section in the MLM and eliminate logic
from the query expressions. The EGADSS team chose XPath because it is simpler and
more suitable for simple data retrievals.
The EGADSS Arden Syntax run-time engine uses the XPath expressions inside the curly braces to retrieve patient-specific information as CDA documents are processed by EGADSS (see Figure 4).  

```xml
knowledge:
  type: data-driven;
  data:
    let diabetes be read exist
      [num/ClinicalDocument/component/structuredBody/activeProblemsComponent/section
       [code/@code="11450-4"]/entry/observation/code
       [starts-with(@code, "250.")]
       or @code="648.0"
       or @code="648.8"
       or @code="790.29"]/@code];
```

Figure 4. XPath Query Embedded in Arden Syntax MLM.

### 3.1.3 EGADSS Recommendations Document

After comparison of the patient summary with the knowledge base EGADSS generates and returns a list of patient specific recommendations in the recommendations document. This document is a Level 2 CDA Template that serves as a container for clinical results returned by EGADSS. It may contain any number of clinical recommendations created by guidelines in the EGADSS knowledge base.

The CDA model does not have constructs for defining structured recommendations. We, therefore, based the structure of this document on the XML recommendation model included in Appendix X1 of the Arden Syntax 2.1 Specification [6]. The main elements are described in Table 3.

Requests for additional data are made in a query-by-example fashion [26]. Each section where EGADSS would like additional data is inserted into the recommendations document as an empty skeleton (including the appropriate section code).

### Table 3. EGADSS Recommendation Document and Element Descriptions.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>guideline title</td>
<td>Title of the Arden Syntax guideline represented as an MLM (Medical Logic Module)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>guideline encoding version</td>
<td>Encoding version number for the guideline</td>
</tr>
<tr>
<td>guideline citations</td>
<td>Reference for the guideline</td>
</tr>
<tr>
<td>guideline domain expert</td>
<td>Name of the clinical and informatics expert/specialist</td>
</tr>
<tr>
<td>guideline encoder</td>
<td>Name of the guideline encoder (Note: encoding of the guideline content is translation from human readable to machine format)</td>
</tr>
<tr>
<td>guideline encoding institution</td>
<td>Name of the institution that facilitated guideline encoding</td>
</tr>
<tr>
<td>guideline last encoded modification</td>
<td>The last time the guideline was encoded by the encoding institution</td>
</tr>
<tr>
<td>guideline recommendation subject</td>
<td>Human readable title of the specific recommendation generated in the guideline</td>
</tr>
<tr>
<td>guideline recommendation instruction</td>
<td>Recommended response to the detected condition</td>
</tr>
<tr>
<td>guideline recommendation conclusion</td>
<td>The main conclusion statement of the guideline after the condition</td>
</tr>
<tr>
<td>guideline recommendation urgency</td>
<td>An integer from 1-99 ranking the urgency of the recommendation. 99- most urgent / severe and 99 the least</td>
</tr>
</tbody>
</table>

### 3.2 Evoke Section

In Arden Syntax there is a section of a MLM called the “Evoke” that designates the event and/or time that triggers the guideline to execute. Evoke logic is specified in two ways: related to a system event such as “storage_of_digoxin_order”, or as a frequency with relation to time such as “every 6 months”.

System events are dependent on the client system. Standards have not been developed to cover clinical system events. The EGDASS design team chose to limit evoke logic to one system event, “incoming CDA document.” This allowed us to pursue a service-based approach without introducing a client dependency in the MLMs. Guidelines with an
Evoke section containing events that are specified according to a client system tightly couple the guideline to that specific client. Our goal was to eliminate this coupling.

For EGADSS to implement evoke logic as frequencies in relation to time, it would violate our high-level requirement that EGADSS be memory-less and stateless for privacy reasons. Such an approach would require registering some form of patient identifier in the EGADSS system, at minimum, so that when the event occurred EGADSS could request the pertinent patient information from the client to execute the guideline.

Since all guidelines in the system are triggered as a result of the same system event, we have eliminated this section from our guidelines, or rather; it is still there but must be left empty.

3.3 Non-isolated Query Logic

The “Data” section of an MLM is where data elements and retrieval queries are located. Queries are explicitly defined inside braces and couple a guideline to a record system implementation, known as the “curly braces” problem of Arden Syntax. Because of this coupling, the Arden Syntax specification does not specify any restrictions or structuring inside these braces. There are however, additional restrictions that can be placed on query results that are defined outside of the curly braces (See “read_where” statement in Arden Syntax 2.1 Specification [6]).

We have designed EGADSS to use the XPath standard as our query language. It is a natural fit for querying our standard-based CDA/XML patient documents. In addition, the XPath 2.0 specification is quite feature rich and provides for all conceivable query logic needed at this time. In order to modularize our queries we have restricted query logic to
XPath 2.0 statements inside the curly braces, and do not allow structures for query logic outside of the braces.

3.4 Implicit Primary Times

In Arden Syntax, a Primary Time is a time stamp associated with a data value. For example, when a lab test value is entered in the clinical record system there is a time stamp implicitly associated with that value. Given that before EGADSS all Arden Syntax implementations were integrated with the record system, it is possible to understand that Arden would provide operations that use these values. An example of such an operation would be taking two lab test values and comparing to see whether one occurred “before” the other.

Our CDA document interface provides the same functionality but it is explicit rather than implicit like Primary Times. In the lab test section there is an explicit location for a lab test time value and a lab test data value. Temporal logic operators can therefore operate on the time value and not on the data value. For example, in our formal semantic description operators “earliest” and “latest” only operate on lists of time values, not on lists of data values. EGADSS does not support operations on implicit Primary Times.

3.5 C-style Output Formatting

The original Arden Syntax specification includes complex logic for string formatting to facilitate displaying messages on command-line user interfaces (common user interface on the original integrated record systems that employed Arden Syntax). Recent versions of the Arden Syntax specification included an appendix with an XML DTD for structured message output [6].
In the service-based design of EGADSS, representing EGADSS recommendations/messages to the user is completely handled by the client. We have, therefore, restricted all of our message output to structured XML messages and do not allow C-style formatted string output.
Chapter 4 Formal Semantic Description

The Arden Syntax Specification contains both a syntactical specification and a semantic specification. The syntax specification is a BNF grammar. The semantic description attaches a paragraph of narrative text to each operation in the BNF grammar. We found this narrative text, at times, unclear and incomplete. We were sometimes left to guess at the intended behavior of the language.

4.1 Specifying Computer Languages

A computer (or programming) language can be described as “a formal notation for describing algorithms for execution by a computer”[27]. Using this definition, executable Guideline Languages are programming languages. We, therefore, consider the Arden Syntax specification in the same context that we consider any other programming language specification.

A programming language specification has two important parts (for the scope of this thesis): syntax and semantics. Informally, syntax can be thought of as how the program is denoted (in terms of structure and keywords) and semantics can be thought of as what the program means (in terms of behavior).

4.1.1 Syntax

Syntax, formally defined, is “a set of formal rules that describe the composition of a program from letters, digits, and other characters”[27]. Syntax can further be broken down into syntactical rules and lexical rules [27].
Lexical rules define what combinations of letters and digits can be used to make up "words," or lexical constructs in the program. For example, '<>' is a valid operator in SQL, but not in Java. The most common and widespread way to specify lexical rules is using regular expressions.

Syntactical rules tell how valid lexical constructs can be combined to make well-formed programs. For example, in Java a variable cannot be declared without expressing its type directly in front of it. The most common and widely accepted way to specify syntactical rules is using a context-free grammar-definition language known as Backus-Naur Form (BNF).

Combined, lexical rules and syntactical rules provide a formal and unambiguous way to specify the syntax of a programming language. The Arden Syntax specification employs this method to describe its program structure.

4.1.2 Semantics

Semantics "give the meaning of any syntactically valid program" [27]. They specify the behavior of syntactical constructs and, therefore, programs. An example of the need for semantic description can be found by looking at the numerous programming languages in existence. Many of them use the same set of characters, in similar or the same combination, to perform something different. In Java, the variable names "begin" and "end" are quite valid. However, in PASCAL, these words have special behavior (semantics) and must be used carefully.

Static semantics are structural constraints on a program that cannot be described by a context-free grammar, but nevertheless, can be validated before a program executes [28, 29]. For example, in Java a variable must be declared (associated with a type) before it
can be used; all variable uses can then be statically checked before execution to ensure that operations on variables are type consistent. In such compile-time checked languages, we can only execute programs that are valid with respect to syntax and static semantics [27].

Dynamic Semantics describe effects of executing a program [28]. These are often referred to as the semantics of a programming language.

4.2 Requirements on a Guideline Language Semantic Description Formalism (GLDSF)

Formal Guideline Languages are used to express clinical logic in a machine executable form. When combined with a patient data repository (or clinical record system), executable guidelines produce point of care reminders/alerts for clinical staff. Typical clinical domains that benefit from such reminders can include preventive care (e.g. cancer screening), chronic disease management (e.g. diabetes), immunizations, prescriptions (e.g. drug-drug interactions) and many others. These computer-interpretable guidelines (CIGs) can play a critical role in patient care. The language that is used to encode such guidelines must be considered carefully. More specifically, we will carefully consider the semantic description formalism of such a guideline language (GLDSF).

Implementers of language systems such as compilers, interpreters, and runtime environments must build such systems based on the language specification or language manual. If the language specification is not clear (requires some interpretation) they could inadvertently introduce defects into the system that could be potentially fatal.

R1. GLDSF should facilitate precise and unambiguous semantic descriptions.
CIGs encoded in a formal guideline language should behave as intended. Given the critical nature of CIGs, it is desirable that they can have their behavior verified. This refers to verification stronger than simply testing. This refers to a formal mathematical proof that the CIG will behave according to the semantic description.

**R2. GLDSF should support formal CIG behavior verification.**

A GLDSF should not be connected to any specific technology or any specific platform. CDS systems can be implemented on any platform in any environment.

**R3. GLDSF should support portable implementation.**

Portable implementations are typically made possible through standardization. If a CIG language is to be a standard then it must be easily maintainable. Maintainability can be broken down into different aspects.

First there is the ability to understand the semantic description, its comprehensibility. We must recognize that a CIG language standard will most likely be maintained by a standards organization, such as HL7, as is Arden Syntax [6]. HL7 deals exclusively with healthcare-related standards and is the dominant standards body, particularly in North America. Committees manage standards. A committee could consist of clinicians, informaticians, and others who have no formal IT education plus a few computer scientists. We therefore define comprehensibility in this context. The less time it takes a healthcare language developer to grasp a language semantic description, the more comprehensible it is.

**R4. GLDSF should be comprehensible.**
There is also the ability to easily modify, or evolve the semantic description. This includes factors of modularity. Tool support is another aspect of maintainability. Good tools can greatly enhance the maintainability of a language semantic description. However, this is not a primary concern because good tools can be developed as the language progresses.

**R5. GLDSF should be evolvable.**

### 4.3 Advantages of a Formal Semantic Description

Formal semantics provide a precise description of language behavior defined according to a mathematical model. As is described below, formal semantics address three of the requirements outlined for a CIG language:

**R1. GLDSF should facilitate precise and unambiguous semantic descriptions**

**R2. GLDSF should support formal CIG behavior verification**

**R3. GLDSF should support portable implementation.**

#### 4.3.1 GLDSF should facilitate precise and unambiguous semantic descriptions

"The primary goal of formal semantics is to provide more effective communication between the language designer and the various audiences with an interest in the language"[30].

Formal semantic descriptions are precise and unambiguous. Semantic Descriptions that overcome the incompleteness and ambiguity of natural language are a great help to those developing systems using these languages. Indeed, implementing a language system for Arden Syntax would have been easier with a formal semantic description.
4.3.2 Support for Program Verification

Particularly in the area of treating patients, the notion of program correctness is very important. It is comforting to know that “programs [CIGs] can be proven correct in a mathematical way and” language systems “can be validated to produce exactly the behavior described in the language definition”[31].

4.3.3 Portability and Language Standardization

Few programming languages have been standardized. One reason for this is due to the many subtle and small details that must be addressed during such an effort. There are so many small details that it is impossible to address them all in a semantic description written in prose. Mathematical precision is necessary to eliminate ambiguity.

According to [27], “The ability to provide formal semantics makes language definitions independent from implementation.” Fulfilling R1 yields a semantic description that is appropriate for standardization and facilitates portable implementation.

4.4 Selecting a Formal Semantic Description Method

There is no widely-accepted approach for expressing semantics [27]. Different methods exist but none has achieved acceptance above all others. We present the most common methods here and evaluate them with respect to the requirements outlined in Section 3.2.1.

The approaches are each illustrated with an ongoing example (from [29]) based on the following syntax production from a grammar representing expressions (See Section 4.4.1 for an informal description).
\[ \text{expr} ::= \text{let } \text{var} = \text{expr1} \text{ in } \text{expr2} \]

We evaluate the approaches in terms of comprehensibility, evolvability, and completeness. We consider a semantic-description formalism comprehensible if we understood it easily and found references to other authors that the approach is understandable. If we found specific references stating a semantic-description formalism is too long or too complex to be useful then we concluded that approach was not comprehensible. We evaluate evolvability in terms of the concepts provided for modularization in the semantic-description formalism. Finally, completeness refers to the capability of an approach to represent both static and dynamic semantics. We would like to provide as much semantic information in the description as possible.

4.4.1 Natural Language Description

This semantic description method is not formal, but useful to compare with the other methods because it is the current semantic description method used by Arden Syntax. Semantics are described using an informal, narrative format. Typically a paragraph of prose is associated with each production in the BNF grammar. This approach is the most common used in language reference manuals and is used by both the language designer and the programmer. This method gives an intuitive view of the language, but suffers from the ambiguity and wordiness of natural language.

4.4.1.1 Example

Occurrences of \text{var} in \text{expr2} denote the value \text{expr1}. The value of \text{expr2} is the value of the whole expression \text{expr}. [29]
4.4.1.2 Evaluation

This approach is quite comprehensible. Developers are able to learn quickly from the semantic description without any knowledge of a special semantic notation. However, such descriptions can be ambiguous. Although this method can be applied to the semantic description in a modular fashion, it is not easily evolvable; changes made in one section can easily contradict a description in another section. There is no way to predictably detect this event and the developer may be left confused.

4.4.2 Attribute Grammars

The idea of Attribute Grammars was first introduced by Knuth in 1968 [32] and is well-described in [28, 29, 33]. Attribute grammars are a way of extending syntax specifications to allow constructs to be defined according to their context [28, 29]. These semantic extensions may be defined based on a concrete syntax specification but it is often more understandable to base them on the abstract syntax of a language specification [28]. The abstract syntax tree is decorated by adding semantic definitions to each syntax production [28]. A semantic definition has three components: attributes, semantic functions, and predicate functions [28, 33].

Each construct is decorated with attributes [28]. Attributes are associated with each non-terminal symbol, and act like variables (they have values assigned to them) [33]. Attributes are either synthesized or inherited. Synthesized attributes are associated with non-terminal symbols on the left side of the production rule and allow information to be passed up the parse tree. Inherited attributes are associated with non-terminals on the right side of the production and allow information to be pass down the parse tree [29, 33].
Intrinsic attributes are associated with leaf nodes (terminals) whose value is determined outside the parse tree (E.g. from a symbol table) [33]. Once given intrinsic attribute values, semantic functions can be used to calculate all the remaining attributes [33].

Each production is decorated with semantic functions and predicate functions [28, 33]. Also known as, attribute computation functions [33], semantic functions specify how attribute values are calculated. The value of synthetic attributes depend on the value of child nodes in the parse tree while the value of inherited attributes depend on the parent nodes, and those of sibling nodes [33].

Predicate functions are Boolean expression over the attribute set that must be true in order for a derivation to be allowed. A derivation is only permitted if every predicate associated with every non-terminal is true [33]. Predicate functions are often specified as part of the semantic functions.

An abstract syntax tree is said to be fully attributed when all the attribute values in a parse tree have been computed [33].

4.4.2.1 Example

Attribute val of <expr> denotes a value. Attribute env for environment binds variables to values. The operation bind(x, v, env) creates a new environment with x bound to v; the bindings for all other variables are as in env.

\[ (<expr>).val := (<expr2>).val \]

The value of <expr> is the value of <expr2>.
\( (\text{expr1}).\text{env} = (\text{expr}).\text{env} \)

Variable bindings in \(<\text{expr1}>\) are the same as in \(<\text{expr}>\).

\( (\text{expr2}).\text{env} := \text{bind}(x,(\text{expr1}).\text{val},\text{env}) \)

In \(<\text{expr2}>\), \(x\) is bound to the value of \(<\text{expr1}>\).

[29]

4.4.2.2 Evaluation

Attribute grammars have been well-adapted to compiler design [28]. They are not always seen as complete enough. Also, with attribute grammars a large number of semantic rules is necessary to describe a real programming language; the semantic description often becomes long and difficult to understand [33].

4.4.3 Operational Semantics

Operational Semantics describe the meaning of a program by executing its statements on a machine (real or virtual). Indeed the machine can be an actual computer, and the operational semantics can be specified by a translator for the language written in the machine code of the chosen machine. Such was the case with C and FORTRAN at one point [31]. In fact, at one time definitional compilers/interpreters were the standard. However, problems with availability of the defining machine/translator/compiler/interpreter made the use of this method difficult. In addition, the underlying machine may not have been completely specified making any definitional compiler/interpreter built on
top of it unreliable [31]. Most modern approaches to operational semantics use an 
abstract machine of some form.

This approach is based on machines, not on logic or mathematics. It describes changes 
in a machine's state when it executes a particular statement. In fact, these semantics can 
be implemented using a programming language [31] yielding an "executable semantic 
description." This allows us to test the correctness of the semantic description by testing 
the resulting interpreter (executable specification).

There are many different abstract machines that can be used to define semantics in this 
manner. Louden uses a reduction machine [31]. In this manner the semantic rules are 
specified in a way that highly resemble logic inference rules allowing the semantic 
description to be easily implemented with a logic programming language (E.g. Prolog).

4.4.3.1 Example

The interpreter eval takes two parameters: an 
expression to be evaluated and an environment with the 
variable bindings.

\[ eval((expr), env) = eval((expr2), bind(x, eval((expr1), env), env)) \] [29]

4.4.3.2 Evaluation

A large difficulty in using operational semantics is selecting the appropriate machine 
language (real or abstract). For example, PL/1 originally had semantics specified using 
VDL. However, the semantic description was so long and complex that it served virtually 
no practical purpose [31]. Evolvability is also a factor of machine language selection. In
this instance of PL/1, this method is not comprehensible or evolvable. In addition operational semantics, as described here, do not readily specify static semantics.

4.4.4 Denotational Semantics

The most rigorous and widely known method for describing semantics is Denotational Semantics. It was first described by Scott and Strachey [34], however, a good introduction can be found in [29] [31, 33]. This method is based on recursive function theory [33]. A complete introduction is long, complex, and beyond the scope of this thesis. We will provide a brief introduction to the basics.

The central idea is that each language construct is represented by mathematical object. An instance of a language construct is associated with an instance of its mathematical object by way of a function. This function, therefore, maps from the syntactic domain to the semantic domain.

A Denotational definition consists of 3 parts [31]:

**Syntactic Domain** consists of the set of all syntactically correct programs.

**Semantic Domain** consists of the set of all syntactically correct programs plus mathematical structures.

**Valuations Function** is the definition of the semantic function itself.

With Denotational semantics, mathematical objects are rigorously defined. They represent the precise meaning of the corresponding language construct.
4.4.4.1 Example

The meaning of expression $E$, written as $\llbracket E \rrbracket$ is a function from environments to values. Thus, $\llbracket E \rrbracket_{env}$, the application of $\llbracket E \rrbracket$ to its environment $env$ is a value.

$\llbracket \text{let } x = E_1 \text{ in } E_2 \rrbracket_{env} = \llbracket E_2 \rrbracket_{bind(x,\llbracket E_1 \rrbracket_{env},env)} \quad [29]$

4.4.4.2 Evaluation

As the descriptions are based on functions (higher-order functions), they can be described using a functional programming language. This may not help with comprehensibility but it greatly aids in proving program correctness and determining the meaning of programs [31]. Also, Denotational semantics do not easily represent static semantics of programs.

4.4.5 Axiomatic Semantics

Axiomatic semantics are based on assertions about the data being manipulated by the program. They describe the effects that a program has on data during execution.

There are two types:

**Preconditions** are true just before a statement executes.

**Post conditions** are true just after a statement executes.

Assertions are written using elements of mathematical logic. As assertions are essentially Boolean expression they can be evaluated as expressions in the language itself. They are true or false at any given time during the execution of a program.
Consider the statement \( \langle \text{var} \rangle := \langle \text{var} \rangle \times 5 \). A precondition could be \( \{ \langle \text{var} \rangle = Y \} \), the resulting post condition would then be \( \{ \langle \text{var} \rangle = Y \times 5 \} \).

Formally an axiomatic semantic description of the syntax construct C is defined \( \{ P \} C \{ Q \} \) where P and Q are assertions; P is the precondition and Q is the post condition [31].

Writing preconditions and post conditions is not always exhaustive. Meaning there is no set of preconditions or post conditions that can necessarily specify the semantics of a construct in completeness.

The example above was quite specific. There should be a way to specify a general relation between P and Q with regards to C. To do this we engage in a goal-oriented activity [31], meaning we usually have a better idea of what we want the outcome of the statement to be. We, therefore, define the post condition of the statement first. We then find the most general precondition that allows the post condition to be fulfilled; this is the weakest precondition [31].

4.4.5.1 Example

Read the logical formula \( env \mapsto E : v \) as, "In environment \( env \), expression \( E \) has the value \( v \)." The rule for let-expression is

\[
\begin{align*}
env \mapsto E_1 : v_1 & \quad \text{bind} (x, v_1, env) \mapsto E_2 : v_2 \\
env \mapsto \text{let } x = E_1 \text{ in } E_2 : v_2
\end{align*}
\]
4.4.5.2 Evaluation

Although I have shown simple examples to introduce Axiomatic Semantics, there are many language constructs that are difficult to describe using this method. Semantic descriptions are modular and this method is useful for research involving proving program correctness but it is far too complex to be considered comprehensible or well-suited to support system developers. In addition, it provides no means to specify static semantics.

4.4.6 Montages

Montages provide a semi-visual formalism for describing languages that provide a unified view of syntax, static aspects and semantics, and dynamic semantics[35]. Montages, is the French word for “assembling” or “putting together.” Indeed, the authors of this approach have recognized that no single approach to semantic representation is acceptable and this unified approach involves “putting together” some of the previously introduced methods.

A language is specified as a collection of Montages; each is associated with a production rule [35]. Each Montage has four parts: production rule, static aspects, static semantics, and dynamic semantics. The static and dynamic semantics are specified using Gurevich’s Abstract State Machines (ASMs) [35, 36]. ASMs from each production combine to form a single ASM (composed of two ASMs: one for static analysis and semantics, one for dynamic semantics) for the entire Montage collection. The language used for describing these ASMs is XASM, formally described by Kutter and Annlauf [37].
The production rule is from the syntax tree. The productions are used to build the parse tree, which is the initial input to the program [35].

The static aspects are used to build up an execution model. Static analysis decorates leaves of the tree with control and data flow building a sequence of tokens that is needed by the ASM [35]. Control flow specifies the order in which statements are executed [35]. Data flow describes how values flow through operations [35]. Control and data flow is provided as attributes to the tokens [35] (not unlike attribute grammars). The static aspects are modeled graphically control and data flow is labeled by arrows between tokens [35], a diagram akin to a data flow graph.

Static Semantics are specified in a way similar to attribute grammars, only they are denoted using syntax from ASMs. These static conditions are checked during static analysis [35].

Dynamic semantics are written using a form of operational semantics where the virtual machine is an ASM. They assume that a program’s control flow and data flow are given (in the form of functions between parts of the program text) [35]. They use ASM transition rules to update program counter and state using the control and data flow functions. Unlike other semantic methods (based on abstract syntax trees), Montages base the dynamic semantics on token sequence [35]. In this model the leaves are considered instructions, called tasks [35].

Program execution is modeled by the evolution of two functions: CT and S [35]. CT, an abstract program counter, points to the line in the program being executed [35]. S is the current value of the store [35]. The initial state for execution is the result of static
analysis. Next states of the ASM are reached by iteratively triggering transition rules [35].

4.4.6.1 Example

First the expression expr1 is visited and evaluated, then the expression expr2 is visited and evaluated. Finally the assign transitional rule is triggered and the dynamic semantics of this production are executed; the value of expr1 is assigned to the variable that expr2 represents.

4.4.6.2 Evaluation

Kutter [35] argues that montages are easily understandable. We partially agree. Montages are easily understandable to a person with the background in language development that has had time to learn the visual notation and formal semantic
description. The target audience of the Arden Syntax semantic description might not have such a background and would find this semantic description not comprehensible.

Two features of Montages make them more evolvable than the other approaches. First, the compositional, modular construction of the set of Montages, and second the length of the specification is comparable to a common reference manual [35] while the length of a semantic description using another approach can be much longer.

Although there are structures for representing static and dynamic semantics, static semantics has the same weakness as attribute grammars; namely, there is no way to guarantee completeness of a static semantic specification.

4.4.7 Conclusion

We recognize that no described approach completely fulfills the requirements outlined above (See Table 4). We propose to develop a semantic description for Arden Syntax using Montages for the reasons described above.
| Table 4: Comparison of Formal Semantic Description Approaches |
|-----------------|----------------|----------------|----------------|
|                 | Natural Language | Attribute Grammars | Operational | Denotational | Axiomatic | Montages |
| Models Dynamic Semantics                  | ✓              |                  |              |              |          |          |
| Models Static Semantics                     |                |                  |              |              |          |          |
| Evolvable                                      |                |                  |              |              |          |          |
| Comprehensible                                    |                |                  |              |              |          |          |
| Supports Verification                               |                |                  |              |              |          |          |
| Facilitates Portability                           |                |                  |              |              |          |          |
| Unambiguous Language                              | ✓              |                  |              |              |          |          |
4.5 Using Montages to Implement a Formal Semantic Description

In order to build our Formal Semantic Description for Arden Syntax we first had to convert the BNF grammar included in the 2.1 specification to one suitable for use in Montages. Then there is the process of writing the Montages. Finally we look at limitations of using Montages for describing semantics and restrictions that EGADSS placed on our formal semantic description.

4.5.1 Converting the Arden Syntax Specification BNF to the Arden Syntax Montages eBNF

The BNF grammar included in the most recent Arden Syntax Specification is specified in an LALR format (Look Ahead, Left-to-right parse, Rightmost derivation, see Appel [38]). The eBNF that a Montages semantic description is based on has certain conditions; productions can take one of two forms:

1. $A := B \, C \, D \, D$ – A is made up of B, C, D, and D in that specific order.

2. $A = E | F | G$ – A can be replaced by one of the alternatives E, F, or G.

The rules help guarantee that only each non-terminal appears on the left side of only one production. In addition Montages eBNF productions can use list {} and optional [] structures.

We constructed a set of transformation rules to make the Arden Syntax LALR BNF grammar suitable for use in a Montages semantic description. These rules are specified in Table 5 and should be applied in the given order.

Table 5. BNF Grammar Transformation Rules

| Transformation Rule | Symbolic Expression | Specific Example |
| Rule 1 – Remove empty alternative productions | $\%_1 = * \%_2 *_2$  
$\%_2 = \{ *_3 \} E \{ *_4 \}$  
$\Rightarrow$  
$\%_1 = *_1 \{ *_3 \} *_4 \}_2$ | $\text{expr} = \text{expr}_\text{and}$  
$\Rightarrow$  
$\text{expr} = [\text{expr}_\text{and}]$ |
| Rule 2 – Collapse single-alternative productions (applied optionally to increase readability) | $\%_1 = *_1 \%_2 *_2$  
$\%_2 = (\ast-1)$  
$\Rightarrow$  
$\%_1 = *_1 (\ast-1) *_2$ | $\text{expr} = \text{expr}_\text{expr}_\text{and} * \text{expr}_\text{or}$  
$\Rightarrow$  
$\text{expr} = \text{expr}_\text{expr}_\text{or}$ |
| Rule 3 – Remove redundant alternatives | $\% = *_1 *_2 *_3$  
$\Rightarrow$  
$\% = [*_1] *_2 [*_3]$ | $\text{expr} = \text{expr} \text{"or"}_\text{expr}_\text{and}$  
$\Rightarrow$  
$\text{expr} = [\text{expr} \text{"or"}_\text{expr}_\text{and}]$  
$\text{expr}_\text{and}$ |
| Rule 4 – Remove Multi-token alternatives | $\% = (\ast-\%}_1$  
$1(\ast-\%}_2$  
$\ldots$  
$1(\ast-\%}_n$  
$\Rightarrow$  
$\% = (\ast-\%}_1$  
$\% = (\ast-\%}_2$  
$\ldots$  
$\% = (\ast-\%}_n$ | $\text{expr} = \{\text{\"expr}_\text{and} \text{\"}\}$  
$\Rightarrow$  
$\text{expr} = \{\text{\"expr}_\text{and} \text{\"}\}$  
$\text{expr} = \{\text{\"expr}_\text{and} \text{\"}\}$ |
| Rule 5 – Convert List Format | $\%_1 = *_1 \%_2 *_2$  
$\Rightarrow$  
$\%_1 = *_1 \{ \%_2 \}_2$ | $\text{expr} = \text{expr}_\text{and}_* \Rightarrow$  
$\text{expr} = \{\text{expr}_\text{and}\}$ |

**Notation:**

* = Any sequence (including empty) of terminals or non-terminals  
% = Any single non-terminal  
E = empty sequence  
{} = Optional  
(* - *) = Sequence of terminals and/or non-terminals with no alternative productions  
(* - %) = Multi-token sequence of terminals and/or non-terminals  
[E] = E  
Subscripts indicate unique language structures
4.5.2 Writing Montages

We have chosen the Boolean "AND" expression from Arden Syntax to demonstrate how to build a Montages.

4.5.2.1 Insert the production from the eBNF Grammar

The first step to building Montages is to insert the production rule into the top section (See Figure 5).

![Expr_and ::= [expr_and "and"] expr_not]

Figure 5. Montages for "AND" expression with the production rule included.

4.5.2.2 Create Synthetic Attributes

There are three synthetic attributes that we have included in this Montages. The first one, "andExpr" uses the "NoNode" XASM operation to determine if the optional portion of the production is present. "NoNode" returns true if the optional portion is not there, therefore andExpr is true if the production is indeed an "AND" expression. Attribute
"staticType" is set to the "staticType" attribute of "expr_not." Attribute "andStaticType" is set to the staticType attribute of "expr_and" only if the expression is an "AND" expression, otherwise it is null (would mean that expr_and on the right side of the production does not exist). The synthetic attributes are shown in Figure 6.

![Figure 6. "AND" expression Montages with Synthetic Attributes.](image)

4.5.2.3 Define Static Semantics

The "AND" operation only accepts Boolean expression types. Static semantics must enforce this. These static semantics are illustrated in Figure 7 by adding XASM statements to the "Condition" section. Note that the static semantics are only enforced if the production is an "AND" expression.
Figure 7. Example "AND" Expression Montages with Static Semantics.

4.5.2.4 Define Control Flow

The control flow is illustrated by an Abstract State Machine where non-terminals are represented by rectangles and transition rules are illustrated by ovals (See Figure 8). Transition rules typically model the dynamic semantics and describe the change of function state. Dotted lines with arrows denote the flow of control. Each production rule ASM has an Initial (I) and Terminal (T) state; these are the entry and exit points for program execution. Optional non-terminal nodes are included in the control flow ASM and skipped over during execution if not present.
4.5.2.5 Define Dynamic Semantics

Finally the dynamic semantics are specified as transition rules in XASM. In this example (See Figure 9), when the control flow reaches the "and" node in the abstract state machine, the transition rule "and" is executed.
4.5.3 Montages Limitations

We encounter some limitations using Montages to describe Arden Syntax. These are described here.

4.5.3.1 XASM Types

Montages base their expression of dynamic semantics on the XASM programming language. XASM is an abstract language with limited type support. There was no logic/structures available for handling the primitive time and duration types of Arden Syntax. However, XASM does provide means to make calls to external functions from within the program. We used this feature extensively to handle Arden’s temporal primitives.
As a result of this limitation we have had to include many complex external functions as part of our semantic description. These do not contribute to the ease of semantic description understanding.

4.5.3.2 Underflow/Overflows

A nice feature of the Arden Syntax 2.0 Specification is the informal specification of magnitudes of certain values. For example, times must be supported back to 01 Jan 1800. Meeting this requirement meant some further refinement in building our system (See Section 5.3.2).

Kutter [39] does not mention underflows or overflows in his dissertation on Montages. We had no way to describe these values in our Montages semantic description. This is an area that needs attention in Montages and in our semantic description.

4.5.4 Restrictions on the Arden Syntax Formal Semantic Description

The Arden Syntax Formal Semantic Description is included as Appendix A Arden Syntax Formal Semantic Description.

There are parts of the semantic description that were limited in our implementation because of requirements of the EGADSS System. We describe these directly below.

4.5.4.1 No Nested Calls to other MLMs

In the current EGADSS we do not allow “call” expressions that execute other MLMs from within an MLM. We acknowledge that this feature is of value and we would like to include it in a future version of EGADSS to facilitate modular CIG construction.
4.5.4.2 Times are Limited to Date and Do Not Include Time of Day

Early in EGADSS design we decided that in order to simplify EGADSS we would initially support times only in date format not including time of day (E.g. 2006-02-21). This decision was tightly linked to the target medical domain of EGADSS, family practice. In family practice temporal logic need only be at a “day” level of granularity, not at a level of hours or minutes. This decision could have been different if EGADSS targeted a domain such as emergency care where decisions are made on a minute-by-minute basis.

We do have plans to include times and operations that support hours, minutes, and seconds so that EGADSS can be used in other domains of care.
Chapter 5  CDS System Design

The key design elements of the EGADSS Arden Syntax Engine are the overall component-based architecture, the service interaction protocol, and the vendor-agnostic patient data model (described in Section 0).

There were some high-level requirements placed on the development of EGADSS that affected our design decisions. For privacy and security reasons EGADSS had to be stateless and memory-less. Storing patient specific data introduces a whole new dimension of complexity that we could not approach in this version of EGADSS.

EGADSS is designed as a supplemental decision support tool to help physicians, but not a tool to make decisions for physicians. At this point, most physicians do not use or rely on a tool such as EGADSS. We therefore place one essential property on the EGADSS system; it should not hinder clinical practice. Aside from that, it is acceptable if EGADSS does not fulfill other properties such as terminating. In the end EGADSS is just a helper that makes suggestions to a physician; it is up to the physician to make the decision and take action.

5.1  EGADSS Overall Architecture

EGADSS employs an evolvable, component-based architecture (See Figure 10). Components interact through exposed interfaces and EGADSS exposes four external interfaces for clients to access (Described in Section 5.1.1). This flexible design allows for many future extensions that require little or no modification to existing components (Described in Section 5.4).
Figure 10. EGADSS Component Architecture

5.1.1 External Interfaces

EGADSS exposes four external interfaces for use by the client system: IKnowledgeMgt, IAudit, ISysConfig, and ITrans. We have included simplified UML descriptions of these external interfaces.

5.1.1.1 IKnowledgeMgt

This interface provides methods for managing the knowledge base of CIGs.
5.1.1.2 IAudit

This interface provides methods for accessing the system logs. The operation `getAuditReport` returns a simple formatted report of the system’s audit trail. The `mode` is a field that holds the current auditing level (e.g. full – audit logs full input/output documents, minimal – audit logs only time stamp of transactions).

5.1.1.3 ISysConfig

This interface provides features for initializing, shutting down, and viewing the configuration of EGADSS.
5.1.1.4 ITrans

This interface provides functionality for transmitting the CDA Patient Summary document from the client and returning CDA recommendations document from EGADSS.

\begin{verbatim}
<<interface>>
ITrans
+getRecommendations(patientDoc) : resultsDoc
\end{verbatim}

Figure 14. UML description of ITrans Interface.

5.1.2 Component Descriptions

5.1.2.1 ConfigManager

The Configuration Manager is the central control of the EGADSS component. It controls the start up and shutdown of all the other components. It is also responsible for Auditing; hence, it exposes an internal interface IAuditLog that all other components depend on to log system events. It depends on the Data Storage component to persist the Audit Logs.

5.1.2.2 KnowledgeManager

The Knowledge Manager handles the Arden Syntax MLMs. It receives them from the client and stores them in the Data Storage component. It depends on the CIG Compiler to convert them from Arden Syntax to the native inference engine format. Finally, it passes the native version of the CIGs to the Inference Engine to load into its rule base.
5.1.2.3 TransController

The Transaction Controller manages the document exchanges between EGADSS and the client system. In so doing it coordinates patient information travel among the EGADSS components. It receives the document from the client and sends it to the DocImpEx for validation. It then sends the document to the Inference Engine and waits for the results to come back. It sends the results file (in interference engine native format) back to the DocImpEx component and receives a well-formed CDA recommendations document that it returns to the client (See Figure 15).

![Sequence Diagram](image)

**Figure 15. GetRecommendations Sequence Diagram**

5.1.2.4 DocImpEx

The Document Import Export component validates incoming documents against the standards-based schema to ensure conformance. These documents are received through the IDocumentImport interface (See Figure 10 and Figure 15). After inference, it receives the patient specific recommendations from the TransController and converts them from a native representation to the CDA Recommendations XML document using the Velocity template engine [40] this operation is exposed through the IDocument Export Interface.
Before returning the new recommendations document to the TransController it validates the new document against the standards-based schema describing the Recommendations CDA Document to ensure that the new document is well formed. In our design these two interfaces (IDocumentImport and IDocumentExport) are realized by the same component. In future versions this may not be the case.

5.1.2.5 Inference Engine

The Inference Engine Component wraps the CLIPS Expert System Shell [41]. By leveraging an existing, mature expert system, we gained rich set of features with less effort than constructing such a system ourselves.

Selecting CLIPS as the core of our inference engine was a major decision. We needed an engine that we could freely distribute (preferably open source), yet we also needed a system that was sufficiently mature and proven. There was also the issue of speed; if a physician has to wait for more than a few seconds for recommendations then the system could disrupt their workflow. The engine should also run on the most common platforms. After considering several open-source java rule-based engines we chose C-based CLIPS because of the maturity, portability, and speed.

CLIPS was developed by NASA at the Johnson Space Centre in the 1980s. It is a forward-chaining, multi-paradigm, expert system shell that can easily be embedded in other programs. It is public domain software and binary downloads are available for Windows, Mac, and Unix operating systems.

The DocImpEx and the CIG Compiler both depend on the Inference Engine. Both handle CLIPS specific files; the CIG Compiler produces CLIPS rule files and the DocImpEx handles CLIPS fact files.
Running the inference engine is fairly simple:

- Load CIGs in CLIPS as CLIPS rules.
- Assert patient information in CLIPS as CLIPS facts.
- Run engine (Rules operate on patient facts to produce new recommendation facts).
- Export recommendation facts
- Reset CLIPS to prepare for next patient document (Removes all facts but leaves rules loaded)

5.1.2.6 CIG Compiler

In operation the CIG compiler is simple and easy to understand. An MLM representation of a CIG goes in and a CLIPS representation of a CIG comes out. However, the actual translation is much more complex. We present a high-level overview of the translation (compilation) process.

Each Arden Syntax MLM maps to two CLIPS structures, a CLIPS Rule and a CLIPS Function (See Figure 16). CLIPS Rules have a left-hand side and a ride-hand side that executes only if the left-hand side evaluates to true. CLIPS functions execute a set of statements.

The data section of the Arden Syntax MLM maps to the left-hand side of the CLIPS rule (See Line 1, Figure 16). The right-hand side of the rule contains the logic portion of the Arden Syntax MLM (See Line 2, Figure 16). Therefore, if all data elements are present in the patient summary then the right-hand side of the rule (logic portion of the MLM) executes. In this fashion, MLMs are data-driven.

The action slot of the Arden Syntax MLM translates to a function in CLIPS (See Line 3, Figure 16). This function is called if the logic portion of the MLM concludes true.

One a finer level, each Arden Syntax “Write” statement translates to a CLIPS MAKE-INSTANCE statement of a CLINICAL_GUIDELINE_RECOMMENDATION (See Line
5, Figure 16). Some of the information in the recommendation instance comes from the maintenance section of the Arden Syntax MLM (See Line 4, Figure 16).
Figure 16. Sample Arden Syntax to CLIPS Translation
Representing patient information in CLIPS was a major obstacle. First we tried representing each patient summary document as a complex lisp-like list structure. We mapped the XPath query statements to data element references. But the list structures were too complex. CLIPS would not reference the data elements.

Next we tried converting each Patient Summary document into a COOL (CLIPS Object Oriented Language) object model. The intention was to then map the XPath queries to object references in the CLIPS language. However, user-defined COOL objects could not contain fields that were also user-defined COOL objects. This constraint made representing our complex patient summary document in COOL impossible.

Finally we found a method that works. XPath statements are isolated from MLMs and returned separately from the CLIPS rules. These XPath statements are then held by the inference engine and executed directly on the patient document at runtime to populate the CLIPS fact base with only the data elements that are needed by the CIGs. This method also speeds up runtime and keeps the fact based to a minimum because only the needed patient data elements are loaded into the inference engine.

5.1.2.7 Data Storage

The simplest of all the components, Data Storage provides a means to persist audit trails and MLM CIGs loaded in the knowledge base. In this first iteration of EGADSS, audit trails and MLMs are persisted directly on the file system. Data Storage is accessed through the IDataAccess interface.
5.2 Service Interaction Protocol

There is a dilemma EGADSS faces when receiving information from a client. EGADSS can provide more patient-specific recommendations when it receives more patient information. It is therefore beneficial for EGADSS to receive as much information as possible about a patient. But, on the other hand, the more information EGADSS receives the more cumbersome it becomes to process that patient document; information processing incurs a performance lag. Put succinctly, it is difficult to be thorough and fast. In addition, there are many different medical conditions; one patient may have a long surgical history while another patient may have a long history of allergies. The usefulness of all this information depends on the CIGs loaded in EGADSS at the time of processing.

5.2.1 2-step Protocol

We initially developed a 2-step protocol to mitigate this problem. The initial patient summary document would intentionally be lean, just enough information to give an overview of the patient’s condition. EGADSS would then request additional data from the client specific to the patient’s condition and the capacities of the current knowledge base (See Figure 17).
Figure 17. 2-Step Service Interaction Protocol.

However, this strategy left a lot of un-answered questions. What if a client could not support a request for additional data? This model could be too complex for some EMR vendors. Or perhaps one request is not enough; there is always the possibility of getting more information that could provide a clearer picture of the patient’s condition. Based on this principle, when would EGADSS ever know it had enough information? It could not.

5.2.2 N-step Protocol

We developed the n-step Protocol to address these issues. Every recommendation document returned from EGADSS has a request for additional data embedded in it. The client has the choice to add more data and make another request to EGADSS for further recommendations or it can simply display the recommendations to the clinician (See Figure 18). There is no limit on the number of requests a client can make to EGADSS, each time providing more information about the patient.
In order for EGADSS to remain stateless in a service-oriented architecture, the client must re-send the entire information plus the additional information in each iteration.

Although the recommendations document has query information in it, we refer to it throughout this thesis as simply the recommendations document.

5.3 Implementation Issues

The major implementation issues we encounter involved what kind of type inference EGADSS should use, how to avoid overflow errors when durations are represented in seconds, and how to handle hierarchies of clinical diagnosis codes. We describe our approach to these issues here.
5.3.1 Type Inference – static or dynamic?

There was an issue regarding the type of the data elements returned by XPath queries. This can be summarized as the issue of static versus dynamic type inference. With static type inference, a variable is associated with a type at compile-time and the use of it must be consistent throughout the program. With dynamic type inference, a variable is not associated with a type and its use is only governed by its context within the program. Indeed with dynamic type inference a variable can be used as a string in one statement and as an integer in the next. The major implication is that type errors can be found at compile-time using static type inference and will not be found until run-time using dynamic type inference. This is an implementation issue and was not mentioned in the Arden Syntax 2.1 Specification [9].

EGADSS employs static type inference in order to catch type errors as early as possible. Employing this strategy, the EGADSS Arden compiler was unable ensure type consistency between the data section and the logic section of an MLM without additional type information about each data element returned by an XPath query. Since the Arden Syntax Specification [9] places no restrictions on what is contained inside the curly braces, EGADSS uses the first three characters to indicate the type of data element represented by the XPath expression; this notation adheres to the standard. In Figure 4, the first three characters inside the curly braces indicate the type “number.”
5.3.2 Seconds Overflow Problem

Internally, CLIPS could not handle the calculations involving temporal durations of seconds. For example, if it were to calculate or compare a date back into the 1940s, it would overflow if it had to represent the time since then in seconds.

The Arden Syntax 2.1 Specification says that the implementation should be able to handle dates back to 1800-01-01 [6]. Therefore, we needed to adopt a new strategy for representing durations internally.

All durations are represented internally as a two-item list. The first item specifies the number of months and the second item specifies the remainder of seconds. Employing this technique, we are able to handle dates back beyond 1800-01-01.

5.3.3 Code Hierarchies – XPath 1.0/2.0

We initially used the XPath 1.0 specification as the basis for our queries. However, as our medical logic became more complex there became a need to handle “hierarchies” of clinical codes. For example, ICD-9 code “651” refers to “Multiple Gestation” (pregnant with more than one child). This code can be constrained further: “651.0” represents twins and “651.1” represents triplets. There is a fifth digit that further characterizes complications with these pregnancies. Perhaps a maternity care CIG simply needs to know if the mother has had a prior episode of multiple gestation. The XPath query would need to search for all events in the Medical History section with codes “651”, “651.x”, or “651.xx” where x could be any alphanumeric value. XPath 1.0 could not reference all of these values in straightforward manner.
We then looked at the XPath 2.0 specification. It included many new features including better wildcard support and a handy "starts-with" operation that facilitates handling code hierarchies. We adopted XPath 2.0 as our new query standard.

5.4 Future Innovations

There are different directions that we could go in the future of EGADSS. Adding additional CIG language support and introducing a new engine are two potential technical innovations. Targeting a specific clinical domain and expanding the knowledge base would allow us to evaluate the clinical impacts of EGADSS.

5.4.1 Additional CIG Language Support

There is no widely used CIG encoding language. Of all the languages mentioned in Section 2.5, they are all equally adopted and (barely) used. Arden Syntax is the only one that is recognized as a standard, but that does not make it well used.

If EGADSS could support several, or more than one of these languages then we could share knowledge with more institutions. EGADSS could be adopted in different environments and the adopter could choose the CIG language most suitable for their clinical domain.

Thanks to the flexible design of EGADSS we could easily add a second CIG compiler component for a new CIG language (See Figure 19). Both CIG compilers could compile to the same inference engine language (CLIPS). This is similar to how languages implemented in the Microsoft .NET framework compile to the Common Language Runtime (CLR) [42].
So, for example, a healthcare organization may receive cancer screening CIGs from the BCMA written in PROforma and immunization CIGs from the BCCDC written in Arden Syntax. If EGADSS supported both PROforma and Arden Syntax then both sets of CIGs could be executed in a single runtime environment.

Figure 19. EGADSS with Two GLCompiler Components.

5.4.2 Targeted Domain/Additional Input Document Support

The next logical clinical step with EGADSS is to validate some of our clinical assumptions. We know that CDS systems have been useful to improve patient care in the past. However, we need to ensure that our service-based approach works.

We would need to develop our knowledge base in a targeted domain, such as maternity care. It would be impractical and too much work to develop a knowledge base covering a broad spectrum of care. Also, with a broad array of CIGs clinical outcomes would be harder to recognize.
This research could involve developing new specialized input documents. Then
documents could then be triaged by the Transaction Controller as EGADSS receives
them and routed to appropriate components. For example, EGADSS could have two
Inference Engine components. One Engine would have a Cancer Screening knowledge
base and the other engine could have a Maternity Care knowledge base. Based on the
input document type, EGADSS could route the document to the appropriate inference engine.

5.4.3 Advanced Inference Techniques

There is a fuzzyCLIPS engine available [43]. It would be interesting to explore the
possibility of adding fuzzy logic/variables to a CIG language and implement a fuzzy CDS
system. This would be particularly useful for describing the vague terms used in
published guidelines such as "elderly" (see Section 2.4).

This is only one example of experimental inference techniques that we could apply to
this domain where simple logic has been used in the past.

5.5 Conclusion

EGADSS has a flexible, component-based architecture that supports the current
functionality well and can easily be adapted to support future innovations.
Chapter 6  Evaluation

Since we have implemented two major contributions we divide the evaluation accordingly. We first consider the Arden Syntax Formal Semantic Description then we take a closer look at the EGADSS CDS System Implementation. We also discuss our overall goal of designing and building a CDS system that is available as a service to different EMR implementations.

6.1  Arden Syntax Formal Semantic Description

We can evaluate the semantic description for its correctness compared to the 2.1 specification [6] and we can evaluate the usefulness of the new semantic description. The major difficulty with making any comparison to a document written in prose is that it must be read, understood, and applied by a person. Such an evaluation is inherently subjective. Having written the new semantic description myself, any evaluation I provide is biased; I therefore present my remarks as observations.

6.1.1.1  XASM Documentation

The language XASM is used to describe the static and dynamic syntax of Montages. This extension to Gurevich’s Abstract State Machines [36] was developed for use in Montages. The use of XASM is not well documented. In fact all known documentation related to XASM is contained in three academic publications [37, 39, 44]. The use of the language XASM has been simplified in these works and what we could glean came from the simple examples. An XASM user guide would be a major contribution to the Montages community.
6.1.1.2 Support for Temporal Primitives

We could not find sufficient documentation about the types and use of primitives in XASM. Arden Syntax has some temporal primitives that require additional logic during operations. We had to code this logic in the semantic description using external functions (written in CLIPS). Since these temporal primitives are fairly pervasive in Arden Syntax there is a large number external functions included in the Montages semantic description. Putting so much of the semantic description in code compromises the visual nature of Montages.

6.2 CDS System Implementation

The CDS System implementation can be evaluated with respect to the Arden Syntax Formal Semantic Description and also consider with respect to the original Arden Syntax 2.1 Specification.

6.2.1 Proving Correctness of CDS implementations

Our initial goal was to use the approach developed by Kalinov et al. to test the mpC compiler [45] to verify the behaviour of the EGADSS Compiler/Engine.

We were unable to follow this approach due to lack of tool support at this time. We give an overview of the approach here and note it as future work because new tools are under development.
6.2.1.1 Test Case Generation

The test case generator (See Figure 20) takes two initial inputs, a file containing a set of primitives and a file containing a list of all the operations with placeholders as parameters.

![Diagram](https://via.placeholder.com/150)

**Figure 20. Test Suite Generation Cycle.**

The Iterator combines the two input files by substituting the primitives into the placeholder of each operation. If we consider the example in Table 6 (from the Arden
Syntax Montages semantic description), the Iterator would yield four syntactically correct expressions (See Table 7).

### Table 6. Example Test Generator Input.

<table>
<thead>
<tr>
<th>Primitive Input Expressions</th>
<th>Operation Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time t</td>
<td>BinaryExpr (&quot;*&quot;, $1, $2)</td>
</tr>
<tr>
<td>Number n</td>
<td></td>
</tr>
</tbody>
</table>

### Table 7. Syntactically Valid Expressions Generated from Example Input.

<table>
<thead>
<tr>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryExpr (&quot;*&quot;, t, t)</td>
</tr>
<tr>
<td>BinaryExpr (&quot;*&quot;, t, n)</td>
</tr>
<tr>
<td>BinaryExpr (&quot;*&quot;, n, n)</td>
</tr>
<tr>
<td>BinaryExpr (&quot;*&quot;, n, t)</td>
</tr>
</tbody>
</table>

The Filter validates each of the syntactically correct expressions against the Montages semantic description to determine which expressions are semantically valid. In the case of our example there is only one semantically correct expression, BinaryExpr("*", n, n). This valid expression becomes the basis of a test case (combined with appropriate variable instantiations and program structures) and is appended to the primitive input to yield a new input file (See Table 8). The test generator commences another test generation cycle using the original Operation Template and the new input file.

### Table 8. New Input Expressions for next Test Generation Cycle.

<table>
<thead>
<tr>
<th>Input Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time t</td>
</tr>
<tr>
<td>Number n</td>
</tr>
<tr>
<td>BinaryExpr(&quot;*&quot;, n, n)</td>
</tr>
</tbody>
</table>
6.2.1.2 Test Oracle Generation

Using the Montages semantic description as a base, an executable oracle can be generated automatically. The Test Suite can be executed on this oracle yielding a set of expected outputs.

The test suite can then be executed in our implementation and the actual output compared with the expected outputs for correctness.

6.2.1.3 Coverage

How many iterations of the test generating cycle does it take to provide sufficient coverage of the semantic description? Kalinov et al. used a heuristic called update rule coverage to see whether every update rule in the static and dynamic semantics of a Montages semantic description is executed [46]. They found that after two iterations of the test generating cycle this condition was satisfied.

6.2.1.4 Emerging Tool Support

The inventor of the Montages semantic description formalism, Phillip Kutter, developed a prototype environment for writing and executing Montages semantic descriptions. He has since finished his PhD and moved on to found a company based on his research (See http://www.montages.com). He recently communicated that they are no longer supporting the prototype and the new commercial execution environment is currently under development. Unfortunately, the prototype was not fully functional and would not support the development of the Arden Syntax Montages semantic description. We look forward to the new execution environment and note as future research the opportunity to test the executable semantic description.
6.2.2 Correct with respect to Original Specification

One major reason we make the Arden Syntax engine available as an open source reference implementation was to provide a way to help future Arden Syntax developers evaluate the correctness of their own implementations. When we first looked at testing our implementation, we searched for an Arden Syntax engine that we could use as an oracle. We wanted something that we could run in parallel to our engine, compare output and validate its correctness. Since no reference implementation existed, we labelled ours as such. In combination with the Montages semantic description it will provide a sure foundation of Arden Syntax behaviour.

However, we still need a way to evaluate the correctness of our implementation. We therefore use a set of test CIGs, patient profiles, and the Arden Syntax 2.1 Specification to generate a set of expected outputs. We then compare the expected output to the real output to demonstrate correct (or incorrect) behaviour of our implementation. As mentioned earlier, because the 2.1 Specification is written in prose it must be read and applied by a person; this is a slow and tedious process. We therefore only have 4 CIGs and 3 patient profiles in our test suite. We present the Diabetes A1C guideline as an example here.

6.2.2.1 Example Guideline: Diabetes A1C

We refer to the sample guideline from Section 2.4 as an example of a test guideline.

6.2.2.2 Patient Profiles and Expected Output

We present an example patient profile, expected outputs, and actual outputs in Table 9 below.

<table>
<thead>
<tr>
<th>Patient Profile</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age 28, Gender: Male, Diabetes: No, Last A1C: Never</td>
<td>No Recommendation</td>
<td>No Recommendation</td>
</tr>
<tr>
<td>Age 28, Gender: Male, Diabetes: Yes, Last A1C: 21 Feb 2006, 6%</td>
<td>Patient Due for A1C</td>
<td>Patient Due for A1C</td>
</tr>
<tr>
<td>Age: 28, Gender: Male, Diabetes: Yes, Last A1C: 20 May 2005, 8%</td>
<td>Patient above 7%</td>
<td>Patient above 7%</td>
</tr>
<tr>
<td>Age: 28, Gender: Male, Diabetes: Yes, Last A1C: 20 May 2005, 6%</td>
<td>No Recommendation</td>
<td>No Recommendation</td>
</tr>
</tbody>
</table>

At the time of writing, all of the actual outputs are correct. In previous months, this method of testing did help us discover some major defects. We describe these defects now.

6.2.2.3 Checking for the Absence of a Value

CLIPS is a data-driven, rule-based system. Data values for each rule are specified as parameters to the rule. If all parameters are present as facts then the rule executes. In one instance the CIG checks for the absence of a value, “if a patient has not received a certain immunization then recommend the immunization”. The problem was that in specific patient profile the immunization did not exist in the patient summary document, so it was never asserted into the fact base. The rule was never executed and we never received the
recommendation for that patient. We redesigned the engine to populate empty facts for these conditions so that the rule would fire and execute properly.

6.2.2.4 The Td Bug

When resolved, the problem from Section 6.2.2.3 caused another issue fondly named "the Td bug". Inserting empty facts made it possible to try and perform operations on data values that do not exist. We now check for the existence of data values in our MLMs before we perform operations on them.

6.3 Adoption by different EMRs

To date EGADSS has been integrated with 2 clients. One is a test client that we constructed to provide a web-based interface with the system (Available at http://nsmobile.org:10800/EgadssWebModule/PatientSelection.jsp).

EGADSS has also been integrated with the open source Technology Assisted Practice Applications Suite (OpenTAPAS, available at http://opentapas.org). OpenTAPAS specifically targets physicians who desire a hybrid practice (partially electronic and partially paper based).

By not placing any requirement that EGADSS depend on any proprietary data model or workflow EGADSS is more adoptable by new EMRs than most existing CDS Systems, which make assumptions about such structures. Indeed they only need to comply with existing standards to interact with EGADSS. Some vendors have also expressed interest in integrating their products with EGADSS’ service-based architecture including one commercial vendor and another open source EMR. EMRs that already export CDA documents can adopt EGADSS with very little extension in functionality.
6.4 Conclusion

Using the formal Arden Syntax Montages semantic description to validate the behaviour of an Arden Syntax Implementation is shown to be a rigorous and effective testing methodology. However, tool support for writing and executing Montages is minimal right now but will be improving in the future.

The most effective method of evaluating the correctness of our Arden Syntax implementation was using a form of “patient profile unit testing” where expected outputs were developed by humans using the Arden Syntax 2.1 Specification [6] and compared with actual output from our engine.

EGADSS has been integrated with two client systems and is adoptable by different EMRs.
Chapter 7 Conclusion

7.1 Contributions

EGADSS was built by a team. For the purpose of this thesis, I (the author) must distinguish my personal contributions from those of the rest of the team.

1) I did an in depth investigation of the Arden Syntax 2.1 Specification. I evaluated its appropriateness for use in a service-oriented CDS environment and defined an appropriate subset of the language for use in the EGADSS environment.

2) I investigated the requirements of a guideline language semantic description formalism. I evaluated formal specification methods and selected Montages as the preferred description method for guideline language semantics.

3) I wrote a Montages Formal Semantic Description of the Arden Syntax 2.1 Specification.

4) I designed the flexible compilation/translator that facilitates the use of more than one CIG language.

5) I built the CIG Compiler Component, which parses Arden Syntax MLMs and translates them into CLIPS Rules and Functions.

6) I successfully developed the Inference Engine Component, with the CLIPS Expert System Shell as a core.

7.2 Findings

We designed a CDS system for use in a service-oriented architecture. EGADSS has been tested with 5 test CIGs and several patient profiles to verify that it behaves
correctly. It has also been integrated with two clients systems, and can easily be adopted by other EMRs.

We found that Arden Syntax, in its current version, is not suitable for use in a service-oriented CDS system. This is probably a result of its design for use in CDS systems that are integrated with record system. Specifically, the Evoke section is tied directly to a record system implementation. Standards must be developed to support this functionality. Primary Times are not a data value that is captured in every record system, but they are integral to several Arden Syntax constructs. Finally, output string formatting is not necessary because our MLMs (and we suggest that other service-oriented CDS Systems) will output XML structured recommendations.

Montages are a promising method to formally specify languages. However, the language used to define static and dynamic semantics, XASM, is poorly documented (only in a few academic journal publications). A user manual is a necessary document that should be developed. We encountered difficulty implementing the temporal primitive types that are part of Arden Syntax. We are unsure if this functionality cannot be implemented with XASM in a straightforward manner or whether our difficulty was a result of insufficient documentation.

7.3 Future Research

In the immediate future we hope to team up with clinical researchers and conduct a clinical trial focused around a specific domain of care (possibly maternity care). We will build a knowledge base of maternity care CIGs and track clinical outcomes related to each of the guidelines. We hope to find a link between the use of EGADSS and the quality of care patients receive. A positive outcome would mean we have achieved our
goal, to make a difference in people lives by using technology to support primary care physicians.
Bibliography


Appendix A
Arden Syntax Formal Semantic Description

program ::= \{mlm\}

mlm ::= maintenance_category
      library_category
      knowledge_category
      "end"

---

Diagram:

- Program defined as a list of mlm.
- mlm defined as maintenance_category, library_category, knowledge_category, and "end".
- Diagram showing the structure of mlm with relationships between categories.

---
maintenance_category ::= "Maintenance:" title_slot mlmname_slot [arden_version_slot] version_slot institution_slot author_slot specialist_slot date_slot validation_slot

library_category ::= "Library:" purpose_slot explanation_slot keywords_slot [citations_slot] [links_slot]
citations_slot = citations_listslot | citations_textslot
links_slot = links_listslot | links_textslot
knowledge_category ::= "Knowledge:" type_slot data_slot [priority_slot] 
evoke_slot logic_slot action_slot [urgency_slot]

---

title_slot ::= "Title:" text ";;"

attr value == text.Name

---

mimname_slot ::= "Mimname:" mimname_text ";;"

attr value == mimname_text.Name

---
arden_version_slot ::= "Arden:" [arden_version] ";;"
arden_version = version2 | version21
version2 ::= "Version" "2"
version21 ::= "Version" "2.1"

attr value == arden_version.Name

version_slot ::= "Version:" text ";;"

attr value == text.Name

institution_slot ::= "Institution:" [text] ";;"

attr value == text.Name

I ----------- > S-arden_version -------- > T

I ----------- > skip ---------------- > T

I ----------- > skip ---------------- > T
author_slot ::= "Author:" [text] ";;"
attr value == text.Name

specialist_slot ::= "Specialist:" [text] ";;"
attr value == text.Name

date_slot ::= "Date:" iso_date ";;"
attr value == iso_date.Name
validation_slot ::= "Validation:" validation_code "::"
validation_code = "Production" | "Research" | "Testing" | "Expired"

attr value == validation_code.Name

purpose_slot ::= "Purpose:" [text] "::"

attr value == text.Name

explanation_slot ::= "Explanation:" [text] "::"

attr value == text.Name
keywords_slot ::= "Keywords:" [text] ";;"
attr value == text.Name

---
citations_listslot ::= "Citations:" [citations_list] ";;"

---
citations_textslot = "Citations:" link_ttext ";;"
attr value == link_ttext.Name

citations_list ::= single_citation ["]" [citations_list]

single_citation ::= [digits "." [citation_type]] string_literal

citation_type = "support" I "refute"

attr value == string_literal.Name
attr type == citation_type.Name

links_listslot ::= "Links:" [links_list] ";;"
links_textslot ::= "Links:" link_ttext ";;"

attr value == link_ttext.Name

| l | ----> skip | ----> T |

links_list ::= [[links_list] ";;;"] single_link

| l | ----> S-links_list | ----> S-single_link | ----> T |

single_link ::= [link_type] [string_literal] term

link_type = "url_link" | "mesh_link" | "other_link" | "exe_link"

attr value == string_literal.Name
attr type == link_type.Name

| l | ----> skip | ----> T |
type_slot ::= "Type:" type_code ";;;"
          type_code = "Data_Driven" | "Data-Driven"
          attr value == type_code.Name

          | --- -> | o  skip  | --- -> | T |

data_slot ::= "Data:" data_block ";;;"

          | --- -> | S-data_block | --- -> | T |

priority_slot ::= "Priority:" number ";;;"

          | --- -> | S-number | --- -> | T |
evoke_slot ::= "Evoke:" ";;"  

    I  ---->     skip  ----> T

logic_slot ::= "Logic:" logic_block ";;"  

    I  ---->          S-logic_block  ----> T

action_slot ::= "Action:" action_block ";;"  

    I  ---->          S-action_block  ----> T
urgency_slot ::= "Urgency:" urgency_val ";"
urgency_val = number | identifier

logic_block ::= [logic_block ",;" ] [logic_statement]
logic_statement = logic_assignment | logic_if | logic_for | logic_while
| logic_conclude
logic_assignment = logic_assexpr | logic_asscall

logic_if ::= "if" expr "then" logic_block ";" | logic_elseif
logic_elseif = "endif" | logic_else | logic_elsif

attr staticType == S-Expr.staticType
condition staticType = BooleanType

I ----> S-expr
   
S-Expr.value ----> S-logic_block
|                  | S-logic_elseif
S-Expr.value = false ----> T
logic_for ::= "for" Identifier "In" expr "do" logic_block ";" "enddo"

attr staticType == S-Expr.staticType

condition staticType = NumberType

@increment:
  identifier.value = identifier.value + 1

logic_while ::= "while" expr "do" logic_block ";" "enddo"

attr staticType == S-Expr.staticType

condition staticType = BooleanType
logic_conclude ::= "conclude" expr

attr staticType == S-Expr.staticType

condition staticType = BooleanType

I -----> S-expr

↓

"conclude"

---

@"conclude":
value := S-expr.value

logic_e = "else" logic_block ";" "endif"

I -----> S-logic_block

---

T
logic_elseif ::= "elseif" expr "then" logic_block ;" logic_elseif

attr staticType == S-expr.staticType

condition staticType = BooleanType

```
T
```

logic_assexpr ::= identifier_becomes expr

```
T
```

@assign
$S-identifier_becomes.signature$ := S-expr.value
\begin{verbatim}
Identifier\_becomes = ident\_eq \lor ident\_be
\end{verbatim}

\begin{verbatim}
ident\_eq ::= \text{identifier} "\:\:\text{":=\:}" \\
attr\ signature = S\text{-}\text{identifier}.Name \\
I \longrightarrow \text{skip} \longrightarrow T
\end{verbatim}

\begin{verbatim}
ident\_be ::= \text{"let" \text{identifier} \text{"be"}} \\
attr\ signature = S\text{-}\text{identifier}.Name \\
I \longrightarrow \text{skip} \longrightarrow T
\end{verbatim}
expr ::= [expr "or"] expr_and

attr orExpr == (not S-"or".NoNode)
attr staticType == S-expr_and.staticType
attr orStaticType == (if orExpr S-expr.staticType else null)

condition (if orExpr then staticType = boolean and orStaticType = boolean)

expr_and ::= [expr_and "and"] expr_not

attr andExpr == (not S-"and".NoNode)
attr staticType == S-expr_not.staticType
attr andStaticType == (if andExpr S-expr_and.staticType else null)

condition (if andExpr then staticType = boolean and andStaticType = boolean)
expr_not ::= ["not"] expr_comparison
expr_comparison = expr_comp1 | expr_comp2 | expr_comp3 | expr_comp4

attr notExpr == (not S-"not".Node)
attr staticType == S-expr_comparison.staticType

condition (if notExpr then staticType = boolean else true)

expr_comp1 ::= [expr_string simple_comp_op] expr_string
simple_comp_op = ":=" | "eq" | ":<" | ""lt" | ":>" | ""gt" | "=" | ":e" | ":>=" | ""ge"

attr compExpr == (not S-simple_comp_op.NoNode)
attr staticType == S2-expr_string.staticType
attr leftStaticType == (if compExpr S1-expr_string.staticType else null)
attr op = (if compExpr S-simple_comp_op.Name else null)

*see next page for transition rule
@comp
  if !compExpr then
    value := expr_string.value
  elseif op = "eq" or op = "=" then
    if S1.expr_string.value = null or S2.expr_string.value = null then
      value := null
      staticType := "null"
    elseif leftStaticType != staticType then
      value := false
      staticType := "boolean"
    else
      value := BinaryCompare (op, leftStaticType, staticType,
                               S1.expr_string.value, S2.expr_string.value)
      staticType := CalculateType (op, leftStaticType, staticType,
                                   S1.expr_string.value, S2.expr_string.value)
  elseif op = "ne" or op = "<>" then
    if S1.expr_string.value = null or S2.expr_string.value = null then
      value := null
      staticType := "null"
    elseif leftStaticType != staticType then
      value := true
      staticType := "boolean"
    else
      value := BinaryCompare (op, leftStaticType, staticType,
                               S1.expr_string.value, S2.expr_string.value)
      staticType := CalculateType (op, leftStaticType, staticType,
                                   S1.expr_string.value, S2.expr_string.value)
  else
    if leftStaticType != staticType then
      value := null
      staticType := "null"
    else
      value := BinaryCompare (op, leftStaticType, staticType,
                               S1.expr_string.value, S2.expr_string.value)
      staticType := CalculateType (op, leftStaticType, staticType,
                                   S1.expr_string.value, S2.expr_string.value)
expr_comp2 := expr_string is ["not"] main_comp_op
main_comp_op = [temporal_comp_op] I unary_comp_op I main_comp_opbin
is = "is" I "are" I "was" I "were"

attr notExpr == (not S-"not".NoNode)
attr notNull == (not S-main_comp_op.NoNode)

1 -
  S-expr_string
  compare
  ⇒ T
  S-main_comp_op

@compare
  if (S-main_comp_op.compType = "unary") then
    value := UnaryCompare (S-main_comp_op.op,
      S-expr_string.staticType,
      S-expr_string.value)
    staticType := UnaryCompareType (S-main_comp_op.op,
      S-expr_string.staticType,
      S-expr_string.value)
  else
    value := BinaryCompare (S-main_comp_op.op,
      S-expr_string.staticType,
      S-main_comp_op.staticType,
      S-expr_string.value,
      S-main_comp_op.value)
    staticType := BinaryCompareType(S-main_comp_op.op,
      S-expr_string.staticType,
      S-main_comp_op.staticType,
      S-expr_string.value,
      S-main_comp_op.value)
  endif

  if (notExpr) then value := not value endif
main_comp_opbin ::= binary_comp_op expr_string
binary_comp_op = bin_great | bin_less | "in"

attr op = (if not S:"in".NoNode then "in" else S-binary_comp_op.op)
attr compType = "binary"
attr staticType = S-expr_string.staticType

@value
value ::= S-expr_string.value

unary_comp_op = "present" | "null" | "boolean" | "number" | "time"
| "duration" | "string" | "list"

attr op = Name
attr compType = "unary"
attr staticType = "null"

I - skip - T
bin\_great ::= "greater" "than" ["or" "equal"]
attr orequal == (not S-"equal".NoNode)
attr op == (if orequal then ">=" else ">")

\[ I \Rightarrow \text{skip} \Rightarrow T \]

bin\_less ::= "less" "than" ["or" "equal"]
attr orequal == (not S-"equal".NoNode)
attr op == (if orequal then "<=" else "<")

\[ I \Rightarrow \text{skip} \Rightarrow T \]

temporal\_comp\_op = temp\_before I temp\_after

\[ I \Rightarrow \text{skip} \Rightarrow T \]
temp_before ::= "before" expr_string

attr op == S."before".Name
attr compType = "binary"
attr staticType = expr_string.staticType

condition staticType = "time"

@value
value := S-expr_string.value

temp_after ::= "after" expr_string

attr op == S."after".Name
attr compType = "binary"
attr staticType = expr_string.staticType

condition staticType = "time"

@value
value := S-expr_string.value
expr_comp3 ::= expr_string [not] in_comp_op
in_comp_op = "in"

attr staticType = S-expr_string.staticType
attr value == S-expr_string.value

* This montages follows the exact Arden Syntax Specification, but there is doubt as to whether this is the original author's intention.

expr_comp4 ::= expr_string occur ['"not"'] [temporal_comp_op]
occur = "occur" | "occurs" | "occurred"

attr notExpr == (not S-"not".NoNode)
attr notNull == (not S-main_comp_op.NoNode)

@compare
value := BinaryCompare (S-temporal_comp_op.op,
S-expr_string.staticType,
S-temporal_comp_op.staticType,
S-expr_string.value,
S-temporal_comp_op.value)
staticType := BinaryCompareType(S-temporal_comp_op.op,
S-expr_string.staticType,
S-temporal_comp_op.staticType,
S-expr_string.value,
S-temporal_comp_op.value)

if (notExpr) then value := not value endif
expr_string ::= [expr_string "il"] expr_plus
expr_plus = expr_plust I expr_minust

attr concat == (not S-"il".NoNode)

@concat
if (concat) then
  value := Concat (S-expr_string.value, S-expr_plus.value)
  staticType := "string"
else
  value := S-expr_plus.value
  staticType := S-expr_plus.staticType

expr_plust ::= [[expr_plus] "+"] expr_times

attr binaryPlus == (not S-expr_plus.NoNode)
attr unaryPlus == (S-expr_plus.NoNode and not S-"+".NoNode)

@plus
if binaryPlus then
  value := BinaryOp ("+", S-expr_plus.staticType, S-expr_times.staticType
                   , S-expr_plus.value, S-expr_times.value)
  staticType := BinaryOpType ("+", S-expr_plus.staticType, S-expr_times.staticType
                               , S-expr_plus.value, S-expr_times.value)
elseif unaryPlus
  value := UnaryOp ("+", S-expr_times.staticType, S-expr_times.value)
  staticType := UnaryOpType ()
else
  value := S-expr_times.value
  staticType := S-expr_times.staticType
expr_minus ::= [expr_plus] "-" expr_times

expr_times = expr_timesb | expr_div

attr binaryMinus == (not S-expr_plus.NoNode)

1 ->
S-expr_plus

minus

S-expr_times

@minus
if binaryMinus then
value := BinaryOp ("-", S-expr_plus.staticType, S-expr_times.staticType,
S-expr_plus.value, S-expr_times.value)
staticType := BinaryOpType ("-", S-expr_plus.staticType, S-expr_times.staticType,
S-expr_plus.value, S-expr_times.value)
else
value := UnaryOp ("-", S-expr_times.staticType, S-expr_times.value)
staticType := UnaryOpType ("-", S-expr_times.staticType, S-expr_times.value)

expr_timesb ::= [expr_times "*"] expr_before

attr timeExpr == (not S-expr_times.NoNode)

1 ->
S-expr_times

times

S-expr_before

@times
if timesExpr then
value := BinaryOp (S."*".Name, S-expr_times.staticType, S-expr_before.staticType,
S-expr_times.value, S-expr_before.value)
staticType := BinaryOpType (S."*".Name, S-expr_times.staticType,
S-expr_before.staticType, S-expr_times.value, S-expr_before.value)
else
value := S-expr_before.value
staticType := S-expr_before.staticType
expr_divb ::= expr_times "/" expr_before

@div
value := BinaryOp (S-"/".Name, S-expr_times.staticType,
S-expr_before.staticType, S-expr_times.value,
S-expr_before.value)
staticType := BinaryOpType (S-"/".Name, S-expr_times.staticType,
S-expr_before.staticType , S-expr_times.value,
S-expr_before.value)

expr_before = expr_beforeago | expr_afterago | expr_fromago
expr_beforeago ::= [expr_duration "before"] expr_ago

attr beforeExpr == (not S="before".NoNode)
attr staticType == S-expr_ago.staticType

condition (if beforeExpr then expr.duration.staticType == "duration"
and S-expr_ago.staticType == "time" else true)

@before
if beforeExpr then
  value := BinaryOp (S="before".Name, S-expr_duration.staticType,
      S-expr_ago.staticType, S-expr_duration.value, S-expr_ago.value)
  staticType := "time"
else
  value := S-expr_ago.value

expr_ago = expr_function | expr_agodur

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I ==&gt; skip ==&gt; T</td>
</tr>
</tbody>
</table>

expr_function = expr_factor | expr_funcof | expr_funcfrom1

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I ==&gt; skip ==&gt; T</td>
</tr>
</tbody>
</table>
expr_afterago ::= expr_duration "after" expr_ago

condition
S-expr_duration.staticType = "duration" and
S-expr_ago.staticType = "time"

1 → S-expr_duration after S-expr_ago → T

@after
value := BinaryOp (S-"after".Name, S-expr_duration.staticType,
S-expr_ago.staticType, S-expr_duration.value, S-expr_ago.value)
staticType := "time"

expr_fromago ::= expr_duration "from" expr_ago

condition
S-expr_duration.staticType = "duration" and
S-expr_ago.staticType = "time"

1 → S-expr_duration from S-expr_ago → T

@from
value := BinaryOp (S-"from".Name, S-duration.staticType,
S-expr_ago.staticType, S-expr_duration.value, S-expr_ago.value)
staticType := "time"
expr_agodur ::= expr_duration ["ago"]
attr agoExpr == (not S-"ago".NoNode)
attr staticType == S-expr_duration.staticType

condition (if agoExpr then S-expr_duration.staticType = "duration"
else true)

expr_duration ::= expr_function duration_op
duration_op = "year" | "years" | "month" | "months" | "week" | "weeks" |
"day" | "days"

condition S-expr_function.staticType = "number"

expr_agodur

I - ➞ S-expr_duration ➞ ago ➞ T

@ago
if agoExpr then
  value := UnaryOp (S-"ago".Name, staticType, S-expr_duration.value)
  staticType := "time"
else
  value := S-expr_agodur.value
@value
  value := UnaryOp (S-duration_op.Name, S-expr_function.staticType, S-expr_function.value)
  staticType := "duration"
expr_funcof := of_func_op ["of"] expr_function
of_func_op = of_read_func_op | of_noread_func_op
of_read_func_op = "count" | "exist" | "exists"
of_noread_func_op = "any" | "all" | "abs" | exyear | exmonth | exday

@value:
value := UnaryOp (S-of_func-op.Name, S-expr_function.staticType, S-expr_function.value)
staticType := UnaryOpType (S-of_func-op.Name, S-expr_function.staticType, S-expr_function.value)

expr_funcfrom1 := from_of_func_op ["of"] expr_function

@value:
value := UnaryOp (S-from_of_func-op.Name, S-expr_function.staticType, S-expr_function.value)
staticType := UnaryOpType (S-from_of_func-op.Name, S-expr_function.staticType, S-expr_function.value)
expr_factor := expr_factor_atom ["[" expr "]"]
expr_factor_atom = identifier | number | string_literal | time_value
  | boolean_value | null | it | "it" | "they" | time_value = "now" | iso_date_time | iso_date
  | "(" expr ")"

I -----> S-expr      value ------> T
        \        |
          \      S-expr_factor_atom

exyear := "extract" "year"

I -----> skip ------> T

exmonth := "extract" "month"

I -----> skip ------> T

exday := "extract" "day"

I -----> skip ------> T
data_block ::= [data_block "," ] [data_assignment]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-data_block</td>
<td>S-data_statement</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

data_assignment ::= identifier_becomes "read" "{" data_mapping "}"

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-read_phrase</td>
<td>assign</td>
</tr>
<tr>
<td></td>
<td>S-identifier_becomes</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

@assign
$S$-identifier_becomes.signature$ := S-read_phrase.value

action_block ::= [action_block "," ] [action_statement]
action_statement = action_if | action_write | action_return

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S-action_block</td>
<td>S-action_statement</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
**action_if** ::= "if" expr "then" action_block ";" action_elseif
action_elseif = "endif" I action_e I action_eif

```
attr staticType == S-Expr.staticType
```

**condition**
staticType = BooleanType

```
I  S-expr
   \---------->
   S-expr.value
   \---------->
   S-action_block
------>
   T
   S-action_elseif
```

**action_write** ::= "write" expr

```
attr staticType == S-Expr.staticType
```

**condition**
staticType = "string"

```
I  S-expr
   \---------->
       assign
   \---------->
   T
```

@assign
value := S-expr.value
**action_e ::= "else" action_block ";" "endif"**

```
I ----> S-action_block ----> T
```

**number ::= digits**

attr constantVal == Name.strToInt
attr staticType == "int"

```
I ----> setVal ----> T
```

@setVal:
value := constantVal

**boolean_value = "true" I "false"**

attr constantVal == (if Name = "true" then true else false)
attr staticType == "boolean"

```
I ----> setVal ----> T
```

@setVal:
value := constantVal
THIS PAGE MISSING FROM ORIGINAL DOCUMENT SUBMITTED
External Functions

(deffunction binaryOp (?op ?leftType ?rightType ?leftValue ?RightValue)
  (if (or (eq ?op "=") (eq ?op "eq"))
      (if (or (eq ?leftType "boolean") (or (eq ?leftType "time") (eq ?leftType "string")))
          (return (eq ?leftValue ?rightValue))
        else
          (if (eq ?leftType "number")
              (return (= ?leftValue ?rightValue))
            else
              (if (eq ?leftType "duration")
                  (return (durationEqDuration ?leftValue ?rightValue))
                )
          )
      )
  )
  )
else
  (if (or (eq ?op "<>") (eq ?op "ne"))
      (if (or (eq ?leftType "boolean") (or (eq ?leftType "time") (eq ?leftType "string")))
          (return (neq ?leftValue ?rightValue))
        else
          (if (eq ?leftType "number")
              (return (<> ?leftValue ?rightValue))
            else
              (if (eq ?leftType "duration")
                  (return (durationNegDuration ?leftValue ?rightValue))
                )
          )
      )
  )
else
  (if (or (eq ?op ">")) (eq ?op "gt"))
      (if (eq ?leftType "number")
          (return (> ?leftValue ?rightValue))
        else
          (if (eq ?leftType "time")
              (return (timeGtTime ?leftValue ?rightValue))
            else
              (if (eq ?leftType "duration")
                  (return (durationGtDuration ?leftValue ?rightValue))
                else
                  (if (eq ?leftType "string")
                      (return (stringGtString ?leftValue ?rightValue))
                    )
              )
          )
      )
  )
else
  (if (or (eq ?op "<") (eq ?op "lt"))
      (if (eq ?leftType "number")
          (return (< ?leftValue ?rightValue))
        )
  )
)
else
  (if (eq ?leftType "time") then
   (return (timeLtTime ?leftValue ?rightValue))
   else
     (if (eq ?leftType "duration") then
      (return (durationLtDuration ?leftValue ?rightValue))
     else
       (if (eq ?leftType "string")
        (return (stringLtString ?leftValue ?rightValue))
       )
     )
  )
else
  (if (or (eq ?op ">=") (eq ?op "ge")) then
    (if (eq ?leftType "number") then
     (return (>= ?leftValue ?rightValue))
    else
     (if (eq ?leftType "time") then
      (return (timeGeTime ?leftValue ?rightValue))
     else
      (if (eq ?leftType "duration") then
       (return (durationGeDuration ?leftValue ?rightValue))
      else
       (if (eq ?leftType "string")
        (return (stringGeString ?leftValue ?rightValue))
       )
      )
     )
  )
else
  (if (or (eq ?op "<") (eq ?op "le")) then
    (if (eq ?leftType "number") then
     (return (<= ?leftValue ?rightValue))
    else
     (if (eq ?leftType "time") then
      (return (timeLeTime ?leftValue ?rightValue))
     else
      (if (eq ?leftType "duration") then
       (return (durationLeDuration ?leftValue ?rightValue))
      else
       (if (eq ?leftType "string")
        (return (stringLeString ?leftValue ?rightValue))
       )
      )
     )
  )
else
  (if (eq ?op "before") then
   (if (and (eq ?leftType "time") (eq ?rightType "time"))
    then
     (return (timeLtTime ?leftValue ?rightValue))
    else
     (if (and (eq ?leftType "duration") (eq ?rightType "time"))
      then
       (return (timeMinusDuration ?rightValue ?leftValue))
      else
       (return (timeLtTime ?leftValue ?rightValue))
     )
   )
  )
// other type? or another before operation?
}

else
  (if (eq ?op "after") then
    (if (and (eq ?leftType "time") (eq ?rightType "time")) then
      (return (timeGtTime ?leftValue ?rightValue))
    else
      (return (timePlusDuration ?rightValue ?leftValue))
    )
  )

else
  (if (eq ?op "+")
    ; number + number -> number
    (if (and (eq ?leftType "number") (eq ?rightType "number")) then
      (return (+ ?leftValue ?rightValue))
    else
      ; time + duration -> time
      (if (and (eq ?leftType "time") (eq ?rightType "duration")) then
        (return (timePlusDuration ?leftValue ?rightValue))
      else
        ; duration + time -> time
        (if (and (eq ?leftType "duration") (eq ?rightType "time")) then
          (return (timePlusDuration ?rightValue ?leftValue))
        else
          ; duration + duration -> duration
          (if (and (eq ?leftType "duration") (eq ?rightType "duration")) then
            (return (durationPlusDuration ?leftValue ?rightValue))
          )
        )
      )
    )
  )
else
  (if (eq ?op "-")
    ; number - number -> number
    (if (and (eq ?leftType "number") (eq ?rightType "number")) then
      (return (- ?leftValue ?rightValue))
    else
      ; duration - duration -> duration
      (if (and (eq ?leftType "duration") (eq ?rightType "duration")) then
        (return (durationMinusDuration ?leftValue ?rightValue))
      else
        ...
; time - duration -> time
(if (and (eq ?leftType "time") (eq ?rightType
?rightValue))
  (return (timeMinusDuration ?leftValue
?rightValue))
else
  ; time - time -> duration
  (if (and (eq ?leftType "time") (eq
?rightType "time")) then
    (return (timeMinusTime ?leftValue
?rightValue))
  )
else
  (if (eq ?op "**") then
    ; number * number -> number
    (if (and (eq ?leftType "number") (eq ?rightType
"number")) then
      (return (* ?leftValue ?rightValue))
    else
      ; number * duration -> duration
      (if (and (eq ?leftType "number") (eq
?rightType "duration")) then
        (return (numberTimesDuration ?leftValue
?rightValue))
      )
    )
else
  (if (eq ?op "/") then
    ; number / number -> number
    (if (and (eq ?leftType "number") (eq
?rightType "number")) then
      (return (/ ?leftValue ?rightValue))
    else
      ; duration / number -> duration
      (if (and (eq ?leftType "duration") (eq
?rightType "number")) then
        (return (durationDividedByNumber
?leftValue ?rightValue))
      )
    )
else
  ; duration from time - 2 days from Mar 1 ->
  Mar 3
  (if (eq ?op "from") then
    (return (timePlusDuration ?rightValue
?leftValue))
  else
    ; any-type in anytype(list) -> boolean
(if (eq ?op "in") then
   (return (contains ?leftValue ?rightValue)))
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
}
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
else
    ; invert sign on duration
    (if (eq type "duration") then
      (return (numberTimesDuration -1 ?value))
    )
  )
else
  (if (eq ?op "ago") then
    (return (timeMinusDuration ?now ?value))
  else
    (if (or (eq ?op "year") (eq ?op "years")) then
      (return (str-cat (* ?value 12) " months"))
    else
      (if (or (eq ?op "month") (eq ?op "months")) then
        (return (str-cat ?value " months"))
      else
        (if (or (eq ?op "week") (eq ?op "weeks")) then
          (return (str-cat (* ?value 604800) " seconds"))
        else
          (if (or (eq ?op "day") (eq ?op "days")) then
            (return (str-cat (* ?value 86400) " seconds"))
          else
            (if (eq ?op "count") then
              (return (length? ?value))
            else
              (if (or (eq ?op "exist") (eq ?op "exists")) then
                (return (exists ?value))
              else
                (if (eq ?op "any") then
                  (if (eq ?type "boolean") then
                    (return (any ?value))
                  else
                    ; returns null if the type of elements in the list are not bool
                    (return null)
                  )
                else
                  (if (eq ?op "all") then
                    ; list of zero members returns true
                    (if (= (count ?value) 0) then
                      (return (= 1 l))
                    else
                      ; only works on lists of boolean values
                      (if (eq ?type "boolean")
                        (return null)
                      else
                        (return (all ?value))
                      )
                    )
                  )
                )
              )
            )
          )
        )
      )
    )
  )
)
else
  (if (eq ?op "abs") then
    ; only works on numbers
    (if (eq ?type "number") then
      (return (abs ?value))
    )
  else
    (if (eq ?op "exyear") then
      ; only works on time
      (if (eq ?type "time") then
        (return (sub-string 1 4
              ?value))
      else
        (return null)
      )
    else
      (if (eq ?op "exmonth") then
        ; only works on time -
        (if (eq ?type "time")
          (return (sub-string 6 7
              ?value))
        else
          (return null)
        )
      else
        (if (eq ?op "exday") then
          ; only works on time
          (if (eq ?type "time")
            (return (sub-string 9
                10 ?value))
          else
            (return null)
          )
        else
          (return null)
        )
      )
    )
  )
)
(deffunction binaryOpType (?op ?leftType ?rightType ?leftValue ?RightValue)
  (if (or (eq ?op "+") (eq ?op "-")) then
    (if (or (eq ?leftType "boolean") (eq ?leftType "time") (eq ?leftType "string")) then
      (return "boolean")
    else
      (if (eq ?leftType "number") then
        (return "boolean")
      else
        (if (eq ?leftType "duration") then
          (return "boolean")
      )
    )
  else
    (if (or (eq ?op "/") (eq ?op "*")) then
      (if (eq ?leftType "number") then
        (return "boolean")
      else
        (if (eq ?leftType "duration") then
          (return "boolean")
      )
    else
      (if (eq ?op "<") (eq ?op "<=") then
        (if (eq ?leftType "number") then
          (return "boolean")
        else
          (if (eq ?leftType "time") then
            (return "boolean")
          else
            (if (eq ?leftType "duration") then
              (return "boolean")
            else
              (if (eq ?leftType "string")
                (return "boolean")
            )
          )
        )
      else
        (if (eq ?leftType "number") then
          (return "boolean")
        else
          (if (eq ?leftType "time") then
            (return "boolean")
          else
            (if (eq ?leftType "duration") then
              (return "boolean")
            else
              (if (eq ?leftType "string")
                (return "boolean")
            )
          )
        )
      )
    else
      (if (eq ?leftType "number") then
        (return "boolean")
      else
        (if (eq ?leftType "time") then
          (return "boolean")
        else
          (if (eq ?leftType "duration") then
            (return "boolean")
          else
            (if (eq ?leftType "string")
              (return "boolean")
          )
        )
      )
  )
)
(if (eq ?leftType "duration") then
   (return "boolean")
else
   (if (eq ?leftType "string")
      (return "boolean")
   )
)

else
   (if (or (eq ?op ">=") (eq ?op "ge")) then
      (if (eq ?leftType "number") then
         (return "boolean")
      else
         (if (eq ?leftType "time") then
            (return "boolean")
         else
            (if (eq ?leftType "duration") then
               (return "boolean")
            else
               (if (eq ?leftType "string")
                  (return "boolean")
               )
         )
      )
   )
else
   (if (or (eq ?op "<=") (eq ?op "le")) then
      (if (eq ?leftType "number") then
         (return "boolean")
      else
         (if (eq ?leftType "time") then
            (return "boolean")
         else
            (if (eq ?leftType "duration") then
               (return "boolean")
            else
               (if (eq ?leftType "string")
                  (return "boolean")
               )
         )
      )
   )
else
   (if (eq ?op "before") then
      (if (and (eq ?leftType "time") (eq ?rightType "time"))
         then
         (return "boolean")
      else
         (if (and (eq ?leftType "duration") (eq ?rightType "time"))
            then
            (return "time")
         else
            // other type? or another before operation?
            )
      )
   )
else
   (if (eq ?op "after") then

(if (and (eq ?leftType "time") (eq ?rightType "time"))
   (return "boolean")
else
   (if (and (eq ?leftType "duration") (eq ?rightType "time"))
      (return "time")
else
   // other type? or another before operation?
)
)
else
   (if (eq ?op "+")
      ; number + number -> number
      (if (and (eq ?leftType "number") (eq ?rightType "number"))
         (return "number")
      else
      ; time + duration -> time
      (if (and (eq ?leftType "time") (eq ?rightType "duration"))
         (return "time")
      else
      ; duration + time -> time
      (if (and (eq ?leftType "duration") (eq ?rightType "time"))
         (return "duration")
      else
      ; duration + duration -> duration
      (if (and (eq ?leftType "duration") (eq ?rightType "duration"))
         (return "duration")
      else
      ; number - number -> number
      (if (and (eq ?leftType "number") (eq ?rightType "number"))
         (return "number")
      else
      ; duration - duration -> duration
      (if (and (eq ?leftType "duration") (eq ?rightType "duration"))
         (return "duration")
      else
      ; time - duration -> time
      (if (and (eq ?leftType "time") (eq ?rightType "duration"))
         (return "time")
      else
      ; time - time -> duration
      (if (and (eq ?leftType "time") (eq ?rightType "time"))
         (return "duration")
      )
   )
)
else
  (if (eq ?op "*" ) then
    ; number * number -> number
    (if (and (eq ?leftType "number") (eq ?rightType "number")) then
     (return "number")
    else
    ; number * duration -> duration
    (if (and (eq ?leftType "number") (eq
     ?rightType "duration")) then
     (return "duration")
    else
    ; duration * number -> duration
    (if (and (eq ?leftType "duration") (eq
     ?rightType "number")) then
     (return "duration")
    (if (eq ?op "/" ) then
     ; number / number -> number
     (if (and (eq ?leftType "number") (eq
     ?rightType "number")) then
     (return "number")
     else
     ; duration / number -> duration
     (if (and (eq ?leftType "duration") (eq
     ?rightType "number")) then
     (return "duration")
     (else
     ; duration from time - 2 days from Mar 1 ->
     Mar 3
     (if (eq ?op "from") then
     (return "time")
     else
     any-type in anytype(list) -> boolean
     (if (eq ?op "in") then
     (return "boolean")
     )
     )
  )
)
(deffunction UnaryOpType (?op ?type ?value)
  (if (eq ?op "present")
       (return "boolean")
   else
    (if (eq ?op "null")
        (return "boolean")
     else
      (if (eq ?op "boolean")
          (return "boolean")
       else
        (if (eq ?op "number")
            (return "boolean")
         else
          (if (eq ?op "time")
              (return "boolean")
           else
            (if (eq ?op "duration")
                (return "boolean")
            else
             (if (eq ?op "string")
                 (return "boolean")
              else
              (if (eq ?op "list")
                  (return "boolean")
              else
                (if (eq ?op "+")
                    ; any type but number returns null
                    (if (neq ?type "number")
                        (return "null")
                    else
                      ; has no effect
                      (return "number")
                    )
                else
                  (if (eq ?op "-")
                      ; invert sign on number
                      (if (eq ?type "number")
                          (return "number")
                       else
                        ; invert sign on duration
                        (if (eq type "duration")
                            (return "duration")
                        )
                  )
              )
          )
       )
   )
else
  (if (eq ?op "ago")
      (return "time")
   else
    (if (or (eq ?op "year") (eq ?op "years"))
        (return "duration")
    else
      (if (or (eq ?op "month") (eq ?op "months"))
          (return "duration")
        else
          )
  )
(if (or (eq ?op "week") (eq ?op "weeks"))
  (return "duration")
else
  (if (or (eq ?op "day") (eq ?op "days"))
    (return "duration")
else
  (if (eq ?op "count") then
    (return "number")
else
  (if (or (eq ?op "exist") (eq ?op "exists")) then
    (return "boolean")
else
  (if (eq ?op "any") then
    (if (eq ?type "boolean") then
      (return "boolean")
else
    ; returns null if the type of elements in the list are not bool
    (return "null")
  )
else
  (if (eq ?op "all") then
    ; list of zero members returns true
    (if (= (count ?value) 0) then
      (return "boolean")
else
    ; only works on lists of boolean values
    (if (neq ?type "boolean")
      (return "null")
else
      (return "boolean")
    )
  )
else
  (if (eq ?op "abs") then
    ; only works on numbers
    (if (eq ?type "number") then
      (return "number")
    )
else
  (if (eq ?op "exyear") then
    ; only works on time
    (if (eq ?type "time") then
      (return "number")
else
    (return "null")
  )
else
  (if (eq ?op "exmonth") then
    ; only works on time -
then

(if (eq ?type "time")
    (return "number")
  else
    (return "null")
)
else
    (if (eq ?op "exday") then
        ; only works on time
        (if (eq ?type "time")
            (return "number")
        else
            (return "null")
    )

)

(deffunction Concat (?leftValue ?rightValue)
    (return (str-cat ?leftValue ?rightValue))
)
(deffunction durGetNum (?dur)
    (nth$ 1 (explode$ ?dur))
)
(deffunction durGetUnit (?dur)
    (nth$ 2 (explode$ ?dur))
)
(deffunction monthsToSeconds (?right)
    (bind ?num (durGetNum ?right))
    (return (str-cat (numToString (* (stringToNum ?num) 2629746)) " seconds")))
)
(deffunction durationEqDuration (?left ?right)
  (if (eq (durGetUnit ?left) (durGetUnit ?right))
    (if (eq (durGetUnit ?left) "months")
      (bind ?left (monthsToSeconds ?left))
      else
      (bind ?right (monthsToSeconds ?right)))
  )
  (return (eq (durGetNum ?left) (durGetNum ?right)))
)

(deffunction durationNeqDuration (?left ?right)
  (if (not (eq (durGetUnit ?left) (durGetUnit ?right)))
    (if (eq (durGetUnit ?left) "months")
      (bind ?left (monthsToSeconds ?left))
      else
      (bind ?right (monthsToSeconds ?right)))
  )
  (return (not (eq (durGetNum ?left) (durGetNum ?right)))
)

(deffunction timeGtTime (?left ?right)
  (if (or (< (str-length ?left) 10) (< (str-length ?right) 10)) then
    (return "timegttime: Wrong Input Length")
  )
  (bind ?dleft (str-cat (sub-string 1 4 ?left) (sub-string 6 7 ?left)
    (sub-string 9 10 ?left)))
  (bind ?dright (str-cat (sub-string 1 4 ?right) (sub-string 6 7
    ?right) (sub-string 9 10 ?right)))
  (return (> (stringToNum ?dleft) (stringToNum ?dright)))
)

(deffunction durationGtDuration (?left ?right)
  (if (eq (durGetUnit ?left) (durGetUnit ?right))
    (if (eq (durGetUnit ?left) "months")
      (bind ?left (monthsToSeconds ?left))
      else
      (bind ?right (monthsToSeconds ?right)))
  )
  (return (> (stringToNum (durGetNum ?left)) (stringToNum (durGetNum
    ?right))))
)

(deffunction stringGtString (?left ?right)
  (return (> (str-compare ?left ?right) 0))
)

(deffunction timeLtTime (?left ?right)
  (if (or (< (str-length ?left) 10) (< (str-length ?right) 10)) then
    (return "timegttime: Wrong Input Length")
  )
  (bind ?dleft (str-cat (sub-string 1 4 ?left) (sub-string 6 7 ?left)
    (sub-string 9 10 ?left)))
  (bind ?dright (str-cat (sub-string 1 4 ?right) (sub-string 6 7
    ?right) (sub-string 9 10 ?right)))
  (return (< (stringToNum ?dleft) (stringToNum ?dright)))
)

(bind ?dleft (str-cat (sub-string 1 4 ?left) (sub-string 6 7 ?left) (sub-string 9 10 ?left)))

(bind ?dright (str-cat (sub-string 1 4 ?right) (sub-string 6 7 ?right) (sub-string 9 10 ?right)))

(return (< (stringToNum ?dleft) (stringToNum ?dright)))

(deffunction durationLtDuration (?left ?right)
  (if (neq (durGetUnit ?left) (durGetUnit ?right))
    (if (eq (durGetUnit ?left) "months")
      (bind ?left (monthsToSeconds ?left))
    else
      (bind ?right (monthsToSeconds ?right)))
  )

(return (< (stringToNum (durGetNum ?left)) (stringToNum (durGetNum ?right)))))

(deffunction stringLtString (?left ?right)
  (return (< (str-compare ?left ?right) 0))
)

(deffunction timeGetTime (?left ?right)
  (if (or (<> (str-length ?left) 10) (<> (str-length ?right) 10)) then
    (return "timeGetTime: Wrong Input Length"
  )

(bind ?dleft (str-cat (sub-string 1 4 ?left) (sub-string 6 7 ?left) (sub-string 9 10 ?left)))

(bind ?dright (str-cat (sub-string 1 4 ?right) (sub-string 6 7 ?right) (sub-string 9 10 ?right)))

(return (>= (stringToNum ?dleft) (stringToNum ?dright)))

(deffunction durationGeDuration (?left ?right)
  (if (neq (durGetUnit ?left) (durGetUnit ?right))
    (if (eq (durGetUnit ?left) "months")
      (bind ?left (monthsToSeconds ?left))
    else
      (bind ?right (monthsToSeconds ?right))
  )

(return (>= (stringToNum (durGetNum ?left)) (stringToNum (durGetNum ?right)))))

(deffunction stringGeString (?left ?right)
  (return (>= (str-compare ?left ?right) 0)))
(deffunction timeLeTime (?left ?right)
  (if (or (<= (str-length ?left) 10) (<= (str-length ?right) 10))
      (return "timegettime: Wrong Input Length")
    )
  (bind ?dleft (str-cat (sub-string 1 4 ?left) (sub-string 6 7 ?left) (sub-string 9 10 ?left)))
  (bind ?dright (str-cat (sub-string 1 4 ?right) (sub-string 6 7 ?right) (sub-string 9 10 ?right)))
  (return (<= (stringToNum ?dleft) (stringToNum ?dright)))
)

(deffunction durationLeDuration (?left ?right)
  (if (neg (durGetUnit ?left) (durGetUnit ?right))
    (return (<= (stringToNum (durGetNum ?left)) (stringToNum (durGetNum ?right))))
  )
)

(deffunction stringLeString (?left ?right)
  (return (<= (str-compare ?left ?right) 0))
)

(deffunction isTime (?time)
  (if (<= (str-length ?time) 10)
    (return (= 0 1))
  )
  (bind ?year (sub-string 1 4 ?time))
  (bind ?dash1 (sub-string 5 5 ?time))
  (bind ?month (sub-string 6 7 ?time))
  (bind ?dash2 (sub-string 8 8 ?time))
  (bind ?day (sub-string 9 10 ?time))
  (return (and (integerp ?year) (and (eq ?dash1 "-") (and (integerp ?month) (and (eq ?dash2 "-") (integerp ?day))))))
)

(deffunction isDuration (?dur)
  (bind ?isnum (integerp (durGetNum ?dur)))
  (bind ?unit (durGetUnit ?dur))
  (return (and ?isnum (or (eq ?unit "months") (eq ?unit "seconds"))))
)

(deffunction exists ($?items)
  (> (length$ ?items) 0))
The number of seconds since 1800-01-01
(def function secondsSinceBase (?time)
  (bind ?year (sub-string 1 4 ?time))
  (bind ?month (sub-string 6 7 ?time))
  (bind ?day (sub-string 9 10 ?time))

  ; number of years since 1800 * 31556952 seconds/year
  (bind ?secondsy (* (- ?year 1800) 31556952))

  ; number of months since 01 * 2629746 seconds /month
  (bind ?secondsm (* (- ?month 01) 2629746))

  ; number of days since 01 * 86400 seconds/day
  (bind ?secondsd (* (- ?day 01) 86400))

  ; add all the seconds together
  (return (+ ?secondsy (+ ?secondsm ?secondsd)))
)

(def function secondsSinceBaseToTime(?seconds)
  ; seconds/ 31556952 -> number of years
  (bind ?years (/ ?seconds 31556952))

  ; get seconds leftover
  (bind ?seconds (mod ?seconds 31556952))

  ; seconds / 2629746 -> number of months
  (bind ?months (/ ?seconds 2629746))
  (bind ?seconds (mod ?seconds 2629746))

  ; seconds / 84600 -> number of days
  (bind ?days (/ ?seconds 2629746))

  ; Apr, Jun, Sep, Nov can't have more than 30 days
  (if (and (> ?days 30) (or (= ?month 4) (or (= ?month 6) (or (= ?month 9) (= ?month 11))))) then
    (bind ?days 30)
  )

  ; Feb can't have more than 28 days
  (if (and (> ?days 28) (= ?month 2)) then
    (bind ?days 28)
  )

  (return (str-cat ?years "-" ?months "-" ?days))
)

(def function timeMinusDuration (?time ?duration)
  (bind ?year (sub-string 1 4 ?time))
  (bind ?month (sub-string 6 7 ?time))
  (bind ?day (sub-string 9 10 ?time))

  ; duration is in months
  (if (eq (durGetUnit ?duration) "months") then
    ; months / 12 -> years to subtract

(bind ?years (~ ?years (/ (durGetNum ?duration) 12)))
; leftover months
(bind ?months (~ ?months (mod (durGetNum ?duration) 12)))

; have to roll back a year? (negative month result)
(if (< ?months 1)
  (bind ?years (~ ?years 1))
  (bind ?months (+ ?months 12))
)
(return (str-cat ?years "-" ?months "-" ?day))

; duration is in seconds
else
  ; get the number of seconds since 1800-01-01
  (bind ?oldSecondsSinceBase (secondsSinceBase ?time))

  ; subtract seconds duration
  (bind ?newSecondsSinceBase (- ?seconds (durGetNum ?duration)))

  ; convert seconds since 1800-01-01 back to a time
  (return (secondsSinceBaseToTime ?newSecondsSinceBase))
)
)

(deffunction timePlusDuration (?time ?duration)
  (bind ?year (sub-string 1 4 ?time))
  (bind ?month (sub-string 6 7 ?time))
  (bind ?day (sub-string 9 10 ?time))

  ; duration is in months
  (if (eq (durGetUnit ?duration) "months") then
    ; months / 12 -> years to add
    (bind ?years (+ ?years (/ (durGetNum ?duration) 12)))
    ; leftover months
    (bind ?months (+ ?months (mod (durGetNum ?duration) 12)))

    ; have to roll forward a year? (months > 12)
    (if (> ?months 12)
      (bind ?years (+ ?years 1))
      (bind ?months (~ ?months 12))
    )
    (return (str-cat ?years "-" ?months "-" ?day))
  )

  ; duration is in seconds
  else
    ; get the number of seconds since 1800-01-01
    (bind ?oldSecondsSinceBase (secondsSinceBase ?time))

    ; add seconds duration
    (bind ?newSecondsSinceBase (+ ?seconds (durGetNum ?duration)))

    ; convert seconds since 1800-01-01 back to a time
    (return (secondsSinceBaseToTime ?newSecondsSinceBase))
  )
)
)

(deffunction durationPlusDuration (?leftDur ?rightDur)
; separate durations into units and numbers
(bind ?lunit (durGetUnit ?leftDur))
(bind ?runit (durGetUnit ?rightDur))
(bind ?lnum (durGetNum ?leftDur))
(bind ?rnum (durGetNum ?rightDur))

; both types months
(if (and (eq ?lunit "months") (eq ?runit "months")) then
 (return (str-cat (+ ?lnum ?rnum) " months"))
else
 ; both types seconds
 (if (and (eq ?lunit "seconds") (eq ?runit "seconds")) then
  (return (str-cat (+ ?lnum ?rnum) " seconds"))
else
 ; left months, right seconds
  (if (and (eq ?lunit "months") (eq ?runit "seconds")) then
   (bind ?newVal (monthsToSeconds ?leftDur))
   (return (str-cat (+ (durGetNum ?newVal) ?rnum) " seconds"))
  else
 ; right months, leftSeconds
   (if (and (eq ?lunit "seconds") (eq ?runit "months")) then
    (bind ?newVal (monthsToSeconds ?rightDur))
    (return (str-cat (+ ?lnum (durGetNum ?newVal)) " seconds"))
   )
)
)
)
}

(deffunction durationMinusDuration (?leftdur ?rightDur)
 ; separate durations into units and numbers
 (bind ?lunit (durGetUnit ?leftDur))
 (bind ?runit (durGetUnit ?rightDur))
 (bind ?lnum (durGetNum ?leftDur))
 (bind ?rnum (durGetNum ?rightDur))

 ; both types months
 (if (and (eq ?lunit "months") (eq ?runit "months")) then
  (return (str-cat (- ?lnum ?rnum) " months"))
else
 ; both types seconds
 (if (and (eq ?lunit "seconds") (eq ?runit "seconds")) then
  (return (str-cat (- ?lnum ?rnum) " seconds"))
else
 ; left months, right seconds
  (if (and (eq ?lunit "months") (eq ?runit "seconds")) then
   (bind ?newVal (monthsToSeconds ?leftDur))
   (return (str-cat (- (durGetNum ?newVal) ?rnum) " seconds"))
  else
 ; right months, leftSeconds
   (if (and (eq ?lunit "seconds") (eq ?runit "months")) then
    (bind ?newVal (monthsToSeconds ?rightDur))
    (return (str-cat (- ?lnum (durGetNum ?newVal)) " seconds"))
   )
)
)
)
(defun timeMinusTime (leftTime rightTime)
  (bind !secs (secondsSinceBase leftTime))
  (bind !secs (secondsSinceBase rightTime))
  (return (str-cat (- !secs !secs) " seconds"))
)

(defun numberTimesDuration (num dur)
  ; case dur units month
  (if (eq (durGetUnit dur) "months")
      (return (str-cat (* num (durGetNum dur)) "months"))
    else
      ; case dur units seconds
      (if (eq (durGetUnit dur) "seconds")
          (return (str-cat (* num (durGetNum dur)) "seconds"))
        )
  )
)

(defun durationDividedByNumber (dur num)
  ; case dur units month
  (if (eq (durGetUnit dur) "months")
      (return (str-cat (/ (durGetNum dur) num) "months"))
    else
      ; case dur units seconds
      (if (eq (durGetUnit dur) "seconds")
          (return (str-cat (/ (durGetNum dur) num) "seconds"))
        )
  )
)

(defun contains (value list)
  (if (eq (member $ value list) (= 0 1))
      (return (= 0 1))
    else
      (return (= 0 0))
  )
)

(defun any (list)
  ; iterate through the list
  (loop-for-count (i 1 (length $ list)) do
    ; if any element is equal to true then return true
    (if (eq (nth $ i list) (= 0 0))
        (return (= 0 0))
      )
  )
)

(defun all (list)
  ; iterate through the list
  (loop-for-count (i 1 (length $ list)) do
    ; if any element is not equal to true then return false
    (if (neg (nth $ i list) (= 0 0))
        (return (= 0 1))
      )
  )
)
Appendix B
Diabetes A1C MLM

maintenance:
  title: Routine A1C in Diabetics;
  mlmname: diabetes_alc;
  version: 0.1;
  institution: University of British Columbia, Department of Family Practice;
  author: Glen McCallum - glen.mccallum@egadss.org;
  specialist: Morgan Price, MD - priceless@famlymed.ca;
  date: 2006-01-19;
  validation: testing;
library:
  purpose: To screen every 3 months for A1C test in all diabetic patients.
  explanation: This patient is a diabetic and, therefore, should be screened every 3 months with a A1C. A further recommendation occurs when alc not controlled but performed.;
  keywords: Diabetes, blood sugar, A1C;
knowledge:
  type: data-driven;
data:
  let diabetes be read exist
  {num/ClinicalDocument/component/structuredBody/activeProblemsComponent/section[code/@code="11450-4"]/entry/observation/code[@code="250"] or code[@code="648.0" or code[@code="648.6" or code[@code="790.29"]]/@code};
  let latest alc value be read exist
  {num/ClinicalDocument/component/structuredBody/labsComponent/section[code/@code="11502-2"]/entry[observation/code[@code="4548-4"]]/observation[@code="max"/ClinicalDocument/component/structuredBody/labsComponent/section/code[@code="11502-2"]/entry/observation/effectiveTime/@value]}/value[@value];
  let latest alc unit be read exist
  {str/ClinicalDocument/component/structuredBody/labsComponent/section/code[@code="11502-2"]/entry[observation/code[@code="4548-4"]]/observation[@code="max"/ClinicalDocument/component/structuredBody/labsComponent/section/code[@code="11502-2"]/entry/observation/effectiveTime/@value]}/value[@unit];
  let latest alc date be read exist
  {tim/ClinicalDocument/component/structuredBody/labsComponent/section[code[@code="11502-2"]/entry[observation/code[@code="4548-4"]]/observation[@code="max"/ClinicalDocument/component/structuredBody/labsComponent/section/code[@code="11502-2"]/entry/observation/effectiveTime/@value]}/effectiveTime[@value];

; evoke:

; logic:

if (not exists diabetes)
  then
    conclude false;
endif;

if ((not exists latest alc date) or (not exists latest alc value) or (not exists latest alc unit))
  then
conclude false;
endif;

if ( (now - latest_alc_date) < 3 months) then
  if ( (latest_alc_value > 7) AND (latest_alc_unit = "%")) then
    patient_above := true;
    conclude true;
  else
    conclude false;
  endif;
else
  patient_due := true;
  conclude true;
endif;

action:
  if patient_above then
    write "<?xml version="1.0" encoding="UTF-8"?>
<structured.message><body><subject>Diabetes: Last A1C > 7%</subject><conclusion>The target goal for A1C is 6-7%. This patient's A1C is above target. Consider discussing lifestyle, diet and therapy with the patient to better control blood sugars.</conclusion><recommendation><instruction>The target goal for A1C is 6-7%. This patient's A1C is above target. Consider discussing lifestyle, diet and therapy with the patient to better control blood sugars.</instruction><recommendation></body></structured.message>";
  elseif patient_due then
    write "<?xml version="1.0" encoding="UTF-8"?>
<structured.message><body><subject>Patient due for A1C</subject><conclusion>The last documented A1C was at least three months ago. Current recommendations are to have an A1C every three months on all diabetic patients.</conclusion><recommendation><instruction>The last documented A1C was at least three months ago. Current recommendations are to have an A1C every three months on all diabetic patients.</instruction><recommendation></body></structured.message>";
  endif;

end;

urgency: 60;