

# Architectural Analysis of Microsoft Dynamics NAV

Tom Hvitved

hvitved@diku.dk

Department of Computer Science

University of Copenhagen

May 26, 2009

## Abstract

This report describes our hands-on experience with the Enterprise Resource Planning (ERP) system Microsoft Dynamics NAV. Much literature exists on Microsoft Dynamics NAV, but none seem to have computer scientists as the main audience. We fill this gap by presenting the architecture of Microsoft Dynamics NAV using well-known concepts and terminology from computer science.

Our architecture analysis is *object based*, meaning that we present the components of NAV as classes in object oriented programming (OOP). During this analysis we address *upgradability* and *performance* problems. Our main observations are presented as *hypotheses*, the most important of which are (1) Lack of database *joins* – and in general database *views* – exacerbates an unnormalized database design which affects both performance and upgradability; (2) Lack of well-defined modules with well-defined interfaces affect upgradability negatively – introduction of joins is needed in order to redesign the software architecture in a modular fashion; and (3) Two key features of Microsoft Dynamics NAV, Sum Index Field Technology and reporting, have performance problems in the current implementation on Microsoft SQL Server.

We propose solutions for (1) - (3) which all take into account that Microsoft Dynamics NAV must be *backwards compatible*: The supply-chain of Microsoft Dynamics NAV is from Microsoft to customers via *partners*, who customize/modify the base product to meet the requirements of the customer. If customers are to upgrade to a new version of the ERP system, it is therefore crucial that the customizations can be ported as well – without having to implement the customizations from scratch. The proposed solutions for (3) are *asymptotically* faster than the current implementations.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Technical Architecture</b>	<b>2</b>
2.1	Prerequisites and Conventions . . . . .	3
2.2	The building blocks of NAV . . . . .	3
2.3	Tables . . . . .	4
2.3.1	Interface . . . . .	4
2.3.2	Triggers . . . . .	6
2.3.3	Keys and Sum Index Field Technology . . . . .	7
2.3.4	Table relations . . . . .	12
2.3.5	Table relations formalized . . . . .	17
2.3.6	FlowFields and FlowFilters . . . . .	18
2.3.7	Iteration . . . . .	22
2.4	Codeunits . . . . .	24
2.4.1	Interface . . . . .	25
2.4.2	C/AL . . . . .	25
2.5	Forms . . . . .	30
2.5.1	Interface . . . . .	30
2.6	Reports . . . . .	31
2.6.1	Interface . . . . .	31
2.7	Runtime system (C/SIDE) . . . . .	34
2.7.1	Concurrency . . . . .	34
2.7.2	NAV 2009 . . . . .	35
<b>3</b>	<b>NAV Design Patterns</b>	<b>36</b>
3.1	Tables . . . . .	36
3.1.1	Master tables . . . . .	36
3.1.2	Templates . . . . .	36
3.1.3	Journals . . . . .	37
3.1.4	Ledgers . . . . .	37
3.1.5	References . . . . .	38
3.1.6	Registers . . . . .	38
3.1.7	Posted documents . . . . .	39
3.1.8	Setup tables . . . . .	40
3.1.9	Virtual tables . . . . .	40
3.2	Codeunits . . . . .	40
3.2.1	Table independent libraries . . . . .	40
3.2.2	Table dependent libraries . . . . .	41
3.2.3	Posting routines . . . . .	41
3.3	Forms . . . . .	41
3.3.1	Card forms . . . . .	41
3.3.2	Tabular/list forms . . . . .	42

3.3.3	Header/detail forms . . . . .	42
3.4	Reports . . . . .	43
<b>4</b>	<b>Architectural Redesign</b>	<b>44</b>
<b>5</b>	<b>Conclusion &amp; Future Work</b>	<b>47</b>
5.1	Future Work . . . . .	48
	<b>Bibliography</b>	<b>49</b>
<b>A</b>	<b>Glossary</b>	<b>52</b>
<b>B</b>	<b>C/AL Parser</b>	<b>53</b>
B.1	Abstract Syntax Tree . . . . .	53
B.2	System Calls . . . . .	59
B.3	Lexer Definition . . . . .	60
B.4	Parser Definition . . . . .	62

# 1 Introduction

This document summarizes the findings of an analysis of the Enterprise Resource Planning (ERP) system Microsoft Dynamics NAV (formerly *Navision*). NAV is an ERP system from Microsoft, targeting small- and medium sized enterprises (SMEs). Microsoft Dynamics NAV has a substantial market share, and is distributed via Microsoft partners, who modify and customize the system to meet the requirements of the individual customers. The following numbers from the Microsoft Dynamics NAV web page [12] illustrate the success of NAV (and hence the relevance of this report):

- More than 57,000 customers (SMEs) worldwide.
- More than 2,700 certified partners worldwide.
- More than 1,500 registered add-ons (*verticals*).

One of the reasons for the success of Microsoft Dynamics NAV is *high flexibility*. Almost all functionality of NAV is written in a domain specific language (C/AL) which is customizable and extensible by NAV partners. C/AL is a relatively small language, making it easy for non-computer scientists to learn (typically NAV developers do not have a background in computer science).

The goal of this report is to document the present NAV architecture from a birds point of view, and come up with a redesign of the architecture which permits modularization, similar to what is known from Service Oriented Architecture (SOA) [3]. We aim at presenting the NAV architecture as concisely as possible; there exists much literature on NAV (see the list references for some), but none of them seem to have computer scientists as the main audience. We hope to fill this gap, by using well-known concepts (and terminology) from computer science such as semantics, invariants, relational databases, object oriented programming etc. Note however that due to the lack of formal documentation of NAV, the formalizations in this document describe our intuition, which may not be coherent with the reality (ultimately, the implementation/code of NAV is the formalization, which is however not very abstract).

Unfortunately we did not succeed with the second goal of reengineering (parts of) the NAV architecture, and we claim that this goal requires introduction of new functionality in Microsoft Dynamics NAV.

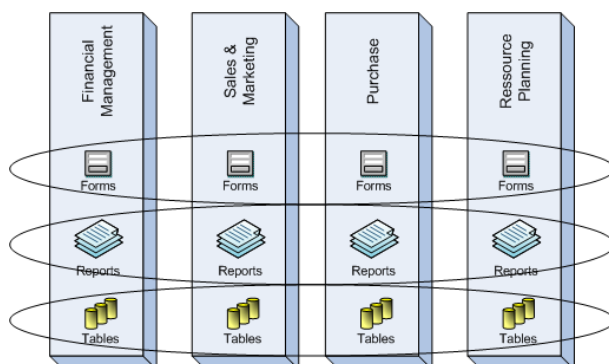
The rest of this document is structured as follows: First we give an overall introduction to NAV, which includes a thorough technical overview and a description of typical NAV design patterns. Then we proceed to a description of our attempt at modularizing NAV functionality, and why our approach did not work. We propose another method for modularizing NAV, and conclude with a list of topics for future work (some of which have already

been initiated). Throughout the document we present *hypotheses*, which are our key observations, and for some of the main problems encountered, we propose solutions.

## 2 Technical Architecture

In this section we present the technical architecture of NAV using concepts and terminology from computer science. In order to avoid a conflict of terminology, we will try to consistently differentiate NAV terminology (by underlining> any first occurrences) and “normal computer science” terminology. Appendix A summarizes the key terms used in NAV and their correspondent in computer science terminology. The presentation in this section will focus on *what* the different key parts of NAV are, rather than *how* they are used. Figure 1 shows this distinction: We focus on the “horizontal” parts rather than the “vertical” parts (the vertical parts are described in course notes [14]).

**Figure 1** Vertical vs. horizontal presentation of Microsoft Dynamics NAV.



Throughout our analysis we have been working with Microsoft Dynamics NAV W1 5.0 SP1, using Microsoft SQL Server 2005 as the underlying database system<sup>1</sup>. NAV uses a classical two-tier architecture with a database system and a thick client running all business logic. We have deliberately chosen only to work with the SQL database system, as opposed to using the native NAV database system, since future releases of NAV will only support Microsoft SQL Server. For most parts of the analysis this choice makes no difference, we will however include the native database briefly when discussing Sum Index Field Technology in Section 2.3.3.

<sup>1</sup>Section 2.7.2 describes why the results of our analysis carry over to the recent version of NAV, Microsoft Dynamics NAV 2009.

## 2.1 Prerequisites and Conventions

In order to get full outcome from this section, the reader should be familiar with the following areas in computer science:

- Relational databases and SQL [2].
- Object oriented programming (OOP) [1].
- Denotational semantics [21].
- Algorithms and data structures [4].

Throughout the report we will use the following typographical conventions:

- NAV objects (tables, codeunits, forms and reports) use the following font: NAV object.
- Column names (fields) use the following font: Field.
- Code (C/AL) use the following font: Code.
- Keyboard strokes use the following font: Key.
- The first occurrence of an important NAV term is underlined.

## 2.2 The building blocks of NAV

As mentioned in the introduction, all business logic (C/AL code – more on C/AL in Section 2.4) runs entirely on the client. This however does not mean that the client is an “out of the box” atomic entity. Instead the client is a runtime system for executing various “building blocks” (called NAV objects – not to be mistaken with objects in OOP) containing business logic.

This means that the client itself does not implement the various components of the ERP system; rather the components are made up from NAV objects, which are then executed by the client. Provided that a proper license is used, it is possible to customize the components by altering the NAV objects they are made up of. This is a key feature of NAV: Almost all functionality is located in NAV objects that are modifiable, making it a highly customizable ERP system. The downside to this flexibility is that developers may alter C/AL code supplied by Microsoft, resulting in the need for a code merge when Microsoft releases a new version of the base code (this is known as the “update problem” in ERP systems [5]).

NAV objects can be divided into the following types:

- Tables

- Codeunits
- Forms
- Reports
- (Dataports), (XMLports) and (MenuSuites)

The following sections describe the first four object types, as these are the core object types of NAV. Our description will consist of an “outside in” and an “inside out” approach: The former is an *object based* description, where we conceptualize the interface each NAV object provides. In this setting each NAV object corresponds to a *class* in OOP, which can be instantiated in other classes (i.e. NAV objects in NAV terminology). The latter will describe how each entry in the class interface is presented in the internal representation/implementation (via both screen dumps and code fragments).

It should be mentioned that our analysis does not take into account the historical evolution of NAV: Many design choices are without a doubt due to the fact that NAV needs to be backwards compatible, as the customizations done by NAV partners need to be portable to new versions. We hence present the architecture *as is*.

Data- and XMLports deal with exporting data from NAV to clear text and XML respectively, and MenuSuites deal with the GUI only (no business logic), so these object types will not be discussed in this report. For a more thorough description of Dataports, XMLports and MenuSuites see [18].

## 2.3 Tables

### 2.3.1 Interface

Microsoft Dynamics NAV uses table objects to store data persistently. An NAV table corresponds to a class in OOP, with the methods and properties presented in Figure 2.

The methods and properties below the keyword **Constant** mean that the entries are shared by all instances of the table object and cannot be changed at runtime. In contrary, entries below **Per instance** can have different values per instance (e.g. a method may refer to instance variables).

Each NAV table has a name (1) and a signature (2). The signature describes the fields (columns in SQL terminology) and the type of each field (*SimpleType* is defined later, so for now just think “simple types”, such as strings, integers, etc.). In the implementation/internal representation each NAV table consists of exactly one table in the SQL database (with the exception of tables containing FlowFields and FlowFilters, more on these in Section 2.3.6).

**Figure 2** The semantic model of a NAV table.

<b>Constant</b>	
1. $Name \in String$	(table name)
2. $\Sigma : String \rightarrow_{\text{fin}} SimpleType \times \mathcal{P}(Property)$	(signature/table schema)
3. $Fields \stackrel{\text{def}}{=} \text{dom}\Sigma$	
4. $PrimaryKey \in Fields^+$	(non-empty primary key definition)
5. $Indexes : Fields^+ \rightarrow \mathcal{P}(Fields)$	(table indexes and Sum Index Field definitions)
6. $TableRelation : Fields \rightarrow TableRelationExp$	(table relations)
7. $\Sigma_{FlowField} : String \rightarrow_{\text{fin}} FlowFieldExp$	(FlowField definitions)
8. $\Sigma_{FlowFilter} : String \rightarrow_{\text{fin}} SimpleType$	(FlowFilter definitions)
9. $\text{dom}\Sigma \cap \text{dom}\Sigma_{FlowField} \cap \text{dom}\Sigma_{FlowFilter} = \emptyset$	(non-overlapping definitions)
<b>Per instance</b>	
10. Built-in methods	(OnInsert, OnDelete, etc.)
11. $Vars : String \rightarrow_{\text{fin}} Type$	(user-defined instance variables)
12. $Methods : String \rightarrow_{\text{fin}} Procedure$	(user-defined methods)
13. Mutators	(built-in methods for updating state, e.g. set a FlowFilter)
14. Iterator	(an iterator for traversing data in the table. Key methods: FIND, INSERT, MODIFY, DELETE)

Rows in a table are called records in NAV terminology. The term record is however overloaded, and sometimes (in particular in C/AL code) it refers to an entire table object, rather than a tuple of values in the table. We assume that this is the case since a table object is essentially an iterator (more on this in Section 2.3.7), and hence it always points to a particular column in the underlying SQL table.

Each field in a table may have associated *properties*, which differ depending on the type of the field; An example is the property `NotBlank`, which – when set – will not allow a record with a null-value in the field (i.e. the field “cannot be blank”). The set *Property* consists of all possible properties (cf. the co-domain of  $\Sigma$ ).

**Figure 3** The NAV table Customer with properties for the field No..

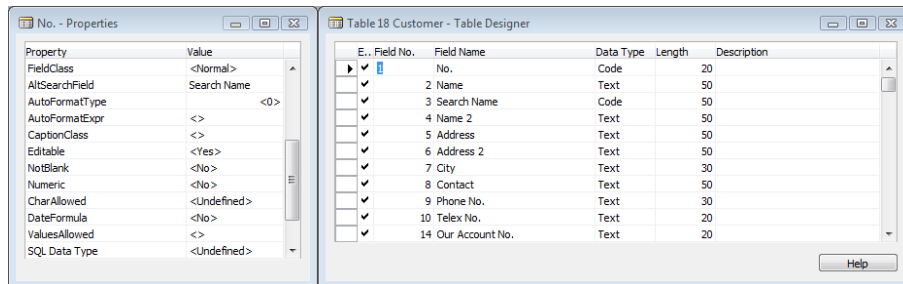


Figure 3 shows the table `Customer`, which holds the information of customers in NAV (only the first 11 fields are included). The figure also shows some of the properties of the first field `No.`. Letting  $\Sigma$  denote the signa-

ture of the `Customer` table, we then have that e.g.  $\pi_1(\Sigma(\text{No.})) = \text{Code}[20]$ ,  $\text{Editable} \in \pi_2(\Sigma(\text{No.}))$  and  $\text{NotBlank} \notin \pi_2(\Sigma(\text{No.}))$ .

What we have seen so far is very similar to SQL. NAV tables have, however, some features which cannot be modeled directly in SQL. We give here a description of some of those features, namely *triggers*, *keys*, *table relations*, *FlowFields/FlowFilters* and the use of an *iterator*.

### 2.3.2 Triggers

Each NAV table has a set of predefined triggers (Figure 2, (10)), which may contain C/AL code. Triggers in NAV correspond to normal methods in OOP, and not triggers as known from active databases [2]. The table below summarizes all table triggers:

Name	Scope	Event
<code>OnInsert</code>	Table	Insertion of a new record
<code>OnModify</code>	Table	Modification of an existing record
<code>OnDelete</code>	Table	Deletion of a record
<code>OnRename</code>	Table	Modification of a field in an existing record, which is part of the tables primary key (Section 2.3.3)
<code>OnValidate</code>	Field	Data is entered into the field, or validation is requested
<code>OnLookup</code>	Field	The user does a lookup (F6) on the field

The scope “Table” means that there is one trigger per table, and the scope “Field” means one trigger per field.

To the authors’ knowledge, table triggers are only executed automatically when the table is accessed directly from the user interface (i.e. from forms, cf. Section 2.5). Thus if a record is inserted programatically (via C/AL code), then the `OnInsert` trigger will *not* be automatically fired! (it is possible to manually trigger the `OnInsert` trigger, however). The same goes for field triggers, where it is *possible* to execute the `OnValidate` code manually from the code, but it is *not* done automatically on insertion/modification.

Thus NAV table triggers are in fact GUI triggers, which makes it dangerous to use them for invariants/validation, as they are not always executed. We believe a more consistent use of triggers should be applied.

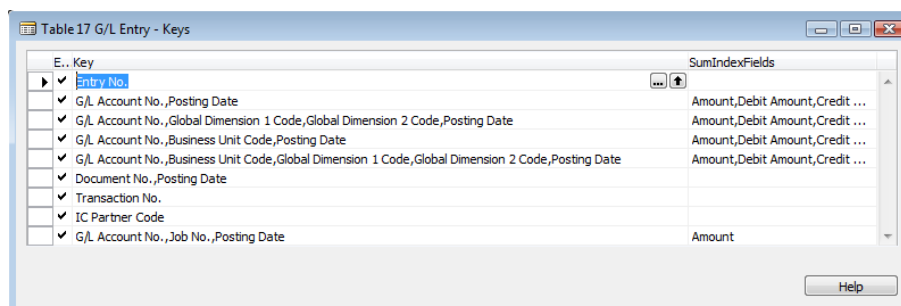
**Hypothesis 1:** *The use of NAV table triggers for business logic (validation) can lead to inconsistent data, as NAV table triggers are not always executed. Solution: Encapsulate tables so they are only accessible via one interface – build database-style triggers (which cannot be escaped) into this encapsulation.*

Besides from the built-in triggers, NAV tables can contain user-definable methods (12) and variables (11). User-definable variables can be used to instantiate other NAV objects (and the table object itself), but we will return to this in Section 2.4, as this is exactly the functionality of codeunits.

### 2.3.3 Keys and Sum Index Field Technology

NAV keys correspond to keys/indexes in database terminology. NAV keys play almost the same role as in SQL: performance enhancements and uniqueness constraints. Each table in NAV *must* have a primary key (Figure 2, (4)), which is translated directly into a primary key in the underlying SQL table. This automatically implies that no two records can be equal, as they must differ on at least one of the fields in the primary key. Figure 4 shows the NAV keys defined on the table G/L Entry.

**Figure 4** NAV Keys on the NAV table G/L Entry.



The first key (Entry no.) is the primary key. In Figure 2 this corresponds to  $PrimaryKey = (Entry\ no.)$ . The other keys are secondary keys, which correspond to indexes in database terminology. The primary key is implicitly appended to all indexes. This means that for instance the second NAV key in Figure 4 is really an index on

$$(G/L\ Account\ No.,\ Posting\ Date,\ Entry\ No.)$$

The column `SumIndexFields` in Figure 4 is where NAV keys differ from SQL keys/indexes: In order to calculate `FlowFields` fast (Section 2.3.6), NAV uses Sum Index Field Technology (SIFT). In Figure 4 `Amount` is defined as a Sum Index Field for the index  $(G/L\ Account\ No.,\ Posting\ Date)$ , which in our semantic model (Figure 2, (5)) means that

$$Amount \in Indexes(G/L\ Account\ No.,\ Posting\ Date)$$

(Note that the order of the fields in the argument to *Indexes* matters). So what does it mean when in general  $F \in Indexes(F_1, \dots, F_n)$  on a table

T? First of all it means that the DBMS maintains an ordinary index on  $(F_1, \dots, F_n, F_{n+1}, \dots, F_{n+m})$ , where *PrimaryKey* =  $(F_{n+1}, \dots, F_{n+m})$  (remember that the primary key is always appended to indexes). As known from database theory, such an index enables support for (*optimal*) *range queries* of the form

$$\sigma_{F_1=v_1 \wedge \dots \wedge F_{i-1}=v_{i-1} \wedge F_i \in [v_i; v'_i]}(\mathbb{T})$$

where  $i \leq n + m$ , each  $v_i/v'_i$  is a value of type  $\pi_1(\Sigma(F_i))$  and we have used notation from relational algebra [2]. The condition  $F_i \in [v_i; v'_i]$  is equivalent to  $v_i \leq F_i \leq v'_i$ , which implies that each type  $\pi_1(\Sigma(F_i))$  must be ordered; and indeed this is the case for all fields allowed in a NAV key – thus one cannot define an index using a field  $F$  if e.g.  $\pi_1(\Sigma(F)) = \text{BLOB}$ .

But having such an index also enables support for computation of aggregated sums<sup>2</sup> (*optimal*) *range sum queries* of the form

$$\sum_{r \in \sigma_{F_1=v_1 \wedge \dots \wedge F_{i-1}=v_{i-1} \wedge F_i \in [v_i; v'_i]}(\mathbb{T})} \pi_F(r)$$

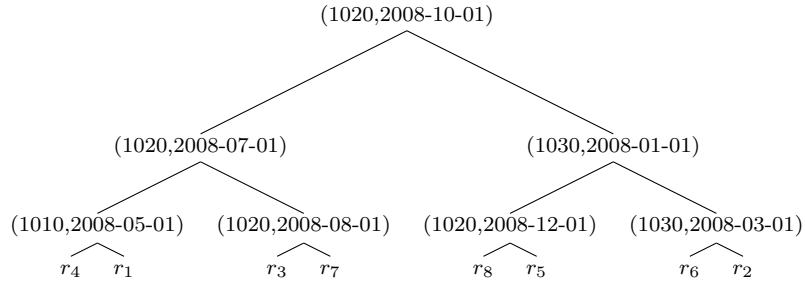
where again  $i \leq n + m$  and each  $v_i/v'_i$  is a value of type  $\pi_1(\Sigma(F_i))$ .

We demonstrate, by example, how NAV indexes can be implemented using balanced search trees (e.g. B+ trees) – and in fact we believe this is in principle how they are implemented in the native NAV database. Consider the table *G/L Entry* from Figure 4 and assume that it contains the following records (throughout this example we omit the implicit appending of the primary key, as it makes no difference):

	G/L Account No.	Posting Date	Amount	...
$r_1 =$	1010	2008-05-01	100	...
$r_2 =$	1030	2008-03-01	300	...
$r_3 =$	1020	2008-07-01	600	...
$r_4 =$	1010	2008-01-01	200	...
$r_5 =$	1020	2008-12-01	100	...
$r_6 =$	1030	2008-01-01	200	...
$r_7 =$	1020	2008-08-01	500	...
$r_8 =$	1020	2008-10-01	200	...

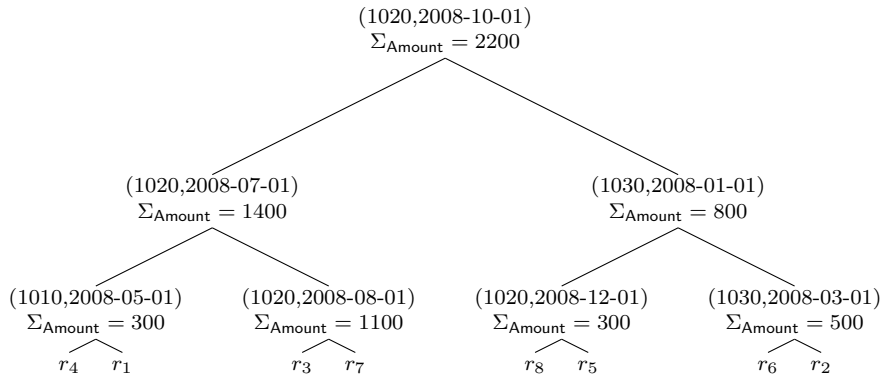
Then as in an ordinary database index, the DBMS maintains a balanced search tree (where the sorting is lexicographically on *G/L Account No.* and *Posting Date*), which may look as follows (here we use the invariant that each node in the left sub tree has a key which is *strictly less*, and each node in the right sub tree has a key which is *greater than or equal*):

<sup>2</sup>Actually SIFT indexes can also be used to compute other aggregated functions, but for now we only consider sums. Section 2.3.6 includes a full list of SIFT functions.



It is then a well-known result from algorithmics that we can search, insert and delete (and hence update, all preserving balancedness) in  $\mathcal{O}(\log n)$ , where  $n$  is the number of records in the table (cf. Chapter 13 in [4] on *red-black trees*, which are similar to B+ trees). This means that the range queries mentioned earlier can be performed in logarithmic time (i.e. logarithmic time – in the size of the table – for indexing plus linear time – in the size of output – for actually retrieving the records).

Now when **Amount** is defined as a Sum Index Field for the index above, we instrument each *internal* node in the search tree with the accumulated sum of all records beneath the node. The index tree above becomes:



It is shown in [4], Theorem 14.1, that maintaining augmented data in a balanced search tree which only depends on the value of the augmented data in the child nodes, introduces no asymptotic overhead on insertion, deletion and update. Since this is the case for aggregated sum<sup>3</sup> (the sum in a node is exactly the sum of the aggregated sums in the child nodes), we get that the aggregated sums can be maintained in  $\mathcal{O}(\log n)$  as well.

Now if we want for instance to compute the sum of all postings on account 1020 from 2008-06-01 to 2008-11-01, then we need to perform two searches in the index tree (one for the lower bound and one for the upper). The informal algorithm is as follows:

<sup>3</sup>And in fact this is the case for all aggregated SIFT functions in NAV, which is why we believe the original implementation used augmented B+ trees.

1. Do a binary search on the tree until we find the first node with key  $k$ , such that  $k \in [(1020,2008-06-01);(1020,2008-11-01)]$  (this is the root node in the example).
2. Do a binary search on the *left* sub tree wrt. the lower bound, i.e. (1020,2008-06-01).
  - Each time we go left, record the sum in the right sub tree (this happens once, where the value 1100 is recorded).
  - Each time we go right, record nothing (this happens once).
  - When a leaf is reached, record the value, if  $k \geq (1020,2008-06-01)$  (this does not happen since the leaf is  $r_1$ ).
3. Do a binary search on the *right* sub tree wrt. the upper bound, i.e. (1020,2008-11-01).
  - Each time we go left, record nothing (this happens twice).
  - Each time we go right, record the sum in the left sub tree (this does not happen).
  - When a leaf is reached, record the value, if  $k \leq (1020,2008-11-01)$  (this happens since the leaf is  $r_8$ ).
4. Return the sum of all recorded values ( $1100 + 200 = 1300$ ).

Since the index tree is always balanced, we are guaranteed that the range sum queries will be in  $\mathcal{O}(\log n)$ , where  $n$  is the number of records in the table.

**Hypothesis 2:** *The native NAV database uses B+ trees [2] for indexes, where each node is augmented with accumulated data. This implies insertion/update and range (sum) queries complexity that is logarithmic in the number of records in the table.*

The benefit of being able to compute total sums in logarithmic time comes at the cost of having to maintain intermediate results when updating tables with Sum Index Fields. A thing to note here is that it is up to the programmer to decide which Sum Index Fields to maintain (i.e. the programmer must specify the map *Indexes*), which may result in an unnecessary amount of indexes being maintained; if for instance a report computes very slow, it is easy to add indexes to make the computation go faster, however it is not *a priori* evident when an index is maintained, but never utilized. This is analogous to deciding when to add an index to a SQL database.

**Hypothesis 3:** *SIFT indexes in NAV are potential performance bottlenecks: When a new record is inserted, intermediate results need to be updated. Possible solution: Automatic generation of SIFT indexes – either from traces in the database or from static code analysis – ensuring that only those indexes which are needed are actually maintained.*

As mentioned in Hypothesis 2, we believe the native database uses B+ trees with logarithmic complexity to maintain SIFT indexes. The implementation in Microsoft SQL Server uses *materialized views* [2] (also called *indexed views*): If  $F \in \text{Indexes}(F_1, \dots, F_n)$  is defined on a table  $T$ , a materialized view is created with the following (simplified) query:

```
SELECT COUNT(*), SUM(F)
FROM T
GROUP BY F_1, \dots, F_n
```

and an ordinary index is created on  $(F_1, \dots, F_n)$ . Having the index means that the view can be updated in worst-case  $\mathcal{O}(\log n)$  on insertion, deletion and modification in  $T$ , where  $n$  is the number of rows in  $T$ . The reason why this is possible is that **SUM** and **COUNT** are *reversible*, i.e. when a row is deleted we can subtract its value in  $F$  from the sum, and decrement the counter (as we shall see soon, this is not the case for all SIFT functions).

Thus maintaining the augmented sums can – as before – be done in  $\mathcal{O}(\log n)$ , assuming that Microsoft SQL Server utilizes the reversibility of **SUM** and **COUNT** to do so. However, range sum queries are now no longer in  $\mathcal{O}(\log n)$ , but worst-case  $\mathcal{O}(n)$ : If  $T$  contains  $n$  rows that are all distinct on  $(F_1, \dots, F_n)$ , then the materialized view will contain  $n$  rows as well. So if we want the sum of *all* rows, then we need to add up  $n$  values!

**Hypothesis 4:** *The implementation of Sum Index Field indexes in Microsoft SQL Server is non-optimal, as the accumulated sums are not directly attached to the B+ index trees. Maintenance of intermediate sums and counts can be done optimally (provided that Microsoft SQL Server utilizes the reversibility of SUM and COUNT), but calculation of range sum queries has worst-case linear complexity.*

So a better solution for Sum Index Field Technology is to augment the B+ index trees. But even though Microsoft SQL Server is being used, it may not be possible to get access to the internal structure of the database indexes, and to augment these. An alternative solution is to *emulate* the augmented B+ tree in a table, where each row represents a node, with pointers to its child nodes. Adding an ordinary index on the pointer identifiers will then make it possible to look up nodes in  $\mathcal{O}(\log n)$ , providing an emulated index tree with complexity  $\mathcal{O}(\log^2 n)$ . So for updates this solution is sub optimal with respect to materialized views, but still asymptotically faster for calculating range sum queries. The question is, however, if the constant overhead will be too high for this solution to be useful.

In Section 2.3.6 we will see that it is also possible to compute *minimum* (and *maximum*) over a field in a table. As is the case for sum and count, minimum can be maintained in an augmented B+ tree in  $\mathcal{O}(\log n)$ . It is however not possible to maintain minimum in a materialized view in  $\mathcal{O}(\log n)$ , as minimum is not reversible! If it is known that  $\min(\pi_F(\sigma_{F_1=v_1 \wedge \dots \wedge F_n=v_n}(T))) = m$

and a row with  $F_i = v_i, i = 1, \dots, n$  is deleted, then the new minimum cannot be computed without inspecting all rows with  $F_i = v_i, i = 1, \dots, n$ .

We suspect this to be the reason why the materialized view is not equipped with minimum (and maximum) as well as sum and count. So if we want to compute minimum over a field, then SIFT will no longer help, and one would expect instead the obvious linear time algorithm:

```
SELECT MIN(F)
FROM T
WHERE F1 = v1, ..., Fi = vi
```

for computing the “range minimum query”  $\min_{r \in \sigma_{F_1=v_1 \wedge \dots \wedge F_i=v_i}(T)} \pi_F(r)$ . But in fact the implementation is even worse: To find the minimum over  $F$ , the following algorithm is applied (as witnessed by inspecting a database trace):

1. Let  $S = \sigma_{F_1=v_1 \wedge \dots \wedge F_i=v_i}(T)$ .
2. Let  $m = \pi_F(r)$ , where  $r$  is the first row in  $S$ .
3. Let  $S = \sigma_{F < m \wedge F_1=v_1 \wedge \dots \wedge F_i=v_i}(T)$ . If  $S = \emptyset$  return  $m$  else go to 2.

Thus in worst-case (when the rows in  $T$  are sorted descending) the complexity is  $\mathcal{O}(n^2)!$  To test this we constructed a table with one million rows, sorted descending. Using the obvious SQL statement from above, the query took under a second, while performing the query from NAV took 25 minutes, resulting in the order of one million SQL statements being executed! To make matters even worse, the selection of  $S$  in the algorithm above uses a (seemingly random) sorting, which implies an actual complexity of  $\mathcal{O}(n^2 \log n)$  (provided that the fields being sorted on are not included in an index).

**Hypothesis 5:** *Minimum and maximum have been removed from SIFT indexes in Microsoft Dynamics NAV 5.0 SP1, as they are not maintainable in a materialized view in  $\mathcal{O}(\log n)$ . The replaced algorithm for minimum and maximum has complexity  $\mathcal{O}(n^2)$ .*

### 2.3.4 Table relations

NAV table relations are much like foreign key constraints in SQL. There are however some differences, which makes it impossible to model the NAV constraints directly in SQL. Table relations are defined on fields (cf. the domain of the function *TableRelation* from Figure 2, (6)). The full grammar for table relation expressions is presented in Figure 5 ([.] means optionality).

The most common (and simplest) type of relation has the form

TargetTable.TargetField

---

**Figure 5** BNF grammar for table relations.

---

<i>TableRelationExp</i>	::=	<i>SimpleRelation</i>   <b>IF</b> <i>Condition</i> <sup>+</sup> <i>SimpleRelation</i> [ <b>ELSE</b> <i>TableRelationExp</i> ]
<i>SimpleRelation</i>	::=	<i>TargetTable</i> [. <i>TargetField</i> ] <b>WHERE</b> <i>WhereClause</i> *
<i>Condition</i>	::=	<i>SourceField</i> = <b>CONST</b> ( <i>Constant</i> )   <i>SourceField</i> = <b>FILTER</b> ( <i>Filter</i> )
<i>WhereClause</i>	::=	<i>TargetField</i> = <b>CONST</b> ( <i>Constant</i> )   <i>TargetField</i> = <b>FIELD</b> ( <i>SourceField</i> )   <i>TargetField</i> = <b>FILTER</b> ( <i>Filter</i> )

---

where *TargetTable* is a table and *TargetField* is a column in *TargetTable*. When this relation is set on a field *SourceField*, i.e.

$$TableRelation(SourceField) = TargetTable.TargetField$$

it means that all records in *SourceTable* must have a value in the field *SourceField* which matches a record in *TargetTable* with the same value in the field *TargetField* (unless the value in the source field is null). More formally expressed, the table relation states that the following must hold:

$$\begin{aligned}
\forall s \in SourceTable. \\
\pi_{SourceField}(s) \neq \mathbf{null} &\Rightarrow \exists t \in TargetTable. \\
&\pi_{SourceField}(s) = \pi_{TargetField}(t)
\end{aligned}$$

This simple kind of relation is exactly what is known from SQL as a foreign key. There is however one important difference: Whereas foreign key constraints in SQL are *invariants* (i.e. the relations are guaranteed to hold at all times outside transactions), table relations in NAV are only checked when records are inserted or modified! Thus if  $TableRelation(F) = rel$ , then the predicate specified by *rel* (will be formalized soon) will only be checked when a new value is inserted in the field *F* of a (possibly new) record.

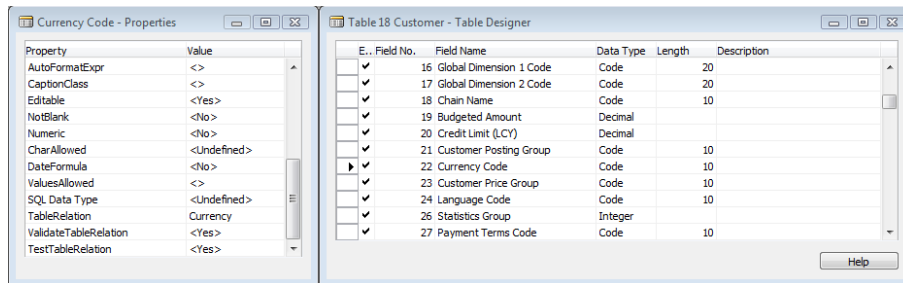
This means that it is possible for a record in *SourceTable* to refer to another record in *TargetTable*, which is later modified or deleted, resulting in a dangling reference. So how can a programmer make sure that a record does not refer data, which may potentially be deleted? The simplest solution is to *copy* the data from the target table; this guarantees *local consistency*, but it also results in an unnormalized database design, which is harder to maintain during upgrades.

**Hypothesis 6:** *Table relations in NAV are not invariants, contrary to e.g. foreign key relations in SQL. This can result in inconsistent data, if*

*data is referred rather than copied. To ensure consistency the developer may therefore resolve to copy data, resulting in unnormalized database design.*

Often the field identifier `TargetField` is left out in the simple relation, in which case the first field of `TargetTable`'s primary key is used. Figure 6 shows the table `Customer` which has a table relation to the table `Currency` set on the field `Currency Code` (i.e.  $TableRelation(Currency\ Code) = Currency$ ). Since the primary key of `Currency` is (`Code`), the table relation is equivalent to  $TableRelation(Currency\ Code) = Currency.Code$ . In effect this means that if `Currency Code` is set on a customer, then a currency must be defined with the corresponding code (as one would expect).

**Figure 6** A simple table relation for the field `Currency Code` in the NAV table `Customer`.



As mentioned above, simple table relations in NAV correspond to SQL foreign keys (modulo invariance), but NAV table relations are more general – in particular NAV table relations can be conditional with respect to the possible target records (via the `WHERE` clause) as well as the possible target table (via the `IF` construction).

We describe the three kinds of conditions in a `WHERE` clause by examples (in the next section we give a formal definition of NAV table relations). In all examples we consider the relation `Currency.Code` from Figure 6 (we include the field identifier `Code` to make the relation explicit):

First consider if we change the relation to

`Currency.Code WHERE EMU Currency = CONST(Yes)`

Then the related record must have `EMU Currency` set to `Yes`, meaning that the currency must be the EMU currency. Second consider if we change the relation to

`Currency.Code WHERE Last Date Modified =  
FILTER(01-01-09..01-02-09)`

Then the related record must have been last modified between 1st of January 2009 and 1st of February 2009. Finally consider if we change the relation to

Currency.Code WHERE Description = FIELD(Name)

Then the related record must have the same value set in the field *Description* as the value in the field *Name* of the *Customer* record (this makes really no sense, and is just meant as an example).

Thus in all three cases the left hand side of the equality in a **WHERE** clause is a field of the related table (the *target* table), and the right hand side is either a constant, a filter or a field in the *source* record. Note that the conditions of a **WHERE** clause in a NAV table relation can be expressed as a **WHERE** clause in a SQL statement, but SQL does not support foreign keys with **WHERE** clauses. Without having investigated it further, we believe one possibility of achieving integrity constraints with **WHERE** clauses directly in SQL could be to use *assertions*. It should however be investigated if this is possible in SQL in general, and in particular whether it is possible in Microsoft SQL Server.

So **WHERE** clauses can be used to put constraints on the records in the target table. But NAV table relations also have the ability to relate to different tables/fields, depending on conditions set on the source table. As can be seen from the grammar, the conditions only include equality with constants and filters – we give an example using both:

Consider again the table relation *Currency.Code* on the field *Currency Code* in table *Customer*. If we change the relation to

```
IF Name = CONST(Bob)
  Currency.Code
ELSE IF Last Date Modified = FILTER(01-01-09..01-02-09)
  Language.Code
```

Then the table relation can have three possible outcomes:

1. If the name of the customer is “Bob” then the relation on the field *Currency Code* is to the field *Code* of the table *Currency* (as before).
2. Otherwise, if the customer record was last modified in the period 1st of January 2009 to 1st of February 2009, the relation is to the field *Code* of the table *Language*.
3. Otherwise there are no table relations.

Conditional table relations is another example of relations that cannot be modeled in SQL using foreign keys. As a consequence, all consistency checks are performed directly in the client, rather than relying on Microsoft SQL Server.

One important thing to note in connection with table relations is that it is not possible to *join* tables in C/AL. One might expect such a facility since tables have relations, and NAV runs directly on Microsoft SQL Server, but this is not the case as NAV did not originally run on a relational database.

As one can imagine, the table relations of NAV can be quite powerful (compared to foreign key constraints in SQL). We claim however that most uses of conditional table relations could be avoided, if C/AL permitted joins: Consider the following (simplified) table relation on the field Bal. Account No. in the table G/L Entry:

```

IF (Bal. Account Type = CONST(G/L Account))
  G/L Account
ELSE IF (Bal. Account Type = CONST(Customer))
  Customer
ELSE IF
...

```

This relation states that if a G/L entry is balanced against a G/L (customer) account, then the account number in the field should be a G/L (customer) account. Figure 7 shows an example where the G/L Entry table contains two records: The first entry is balanced against a G/L account and the last entry is balanced against a customer account (the relations for each record are shown in parentheses).

**Figure 7** NAV table G/L Entry with two records.

G/L Entry no.	Bal. Account Type	Bal. Account No.
1	G/L Account	1010 (→ G/L Account)
2	Customer	1020 (→ Customer)

It is possible to achieve the same effect using only static table relations, if we instead use *mapping tables*, i.e. one table for each branch in the conditional table relation (Figure 8).

**Figure 8** G/L account mapping table (top), customer account mapping table (bottom).

G/L Entry no.	G/L Account no.
1 (→ G/L Entry)	1010 (→ G/L Account)

G/L Entry no.	Customer Account no.
2 (→ G/L Entry)	1020 (→ Customer)

In this way conditional table relations can be replaced by simple table relations that are checkable directly in Microsoft SQL Server, rather than having the client to do the checks. Furthermore, we believe the complex table relations can be an indication of unnormalized database design; It is likely that the target of a (complex) table relation determines which other

fields of the source table should be active. It would hence be better to place these fields in the mapping table (as described above), to eliminate null-values in the source table.

**Hypothesis 7:** *Table relations in NAV are unnecessarily complex: The use of conditionals can be a sign of unnormalized database design. Solution: Only allow foreign key relations (as invariants) and introduce joins (or in general views).*

### 2.3.5 Table relations formalized

In this section we briefly present the (denotational) semantics of table relations, which may be easier to understand to some readers. Assume

$$\text{TableRelation}(\text{SourceField}) = \text{rel}$$

in the table `SourceTable`. Then each time a record  $s$  is either created or modified in `SourceTable`, the NAV client checks that the following holds:

$$\pi_{\text{SourceField}}(s) \neq \text{null} \Rightarrow \mathcal{R}[\text{rel}](\text{SourceField}, s)$$

where  $\mathcal{R}[\text{rel}](\text{SourceField}, s)$  is defined by structural induction on  $\text{rel}$ :

$$\begin{aligned} & \mathcal{R}[\text{TargetTable.TargetField WHERE } cs](\text{SourceField}, s) \\ &= \exists t \in \text{TargetTable}. \pi_{\text{SourceField}}(s) = \pi_{\text{TargetField}}(t) \wedge \bigwedge_{c \in cs} \mathcal{W}[c](s, t) \\ \\ & \mathcal{R}[\text{IF } cs \text{ rel}](\text{SourceField}, s) \\ &= \begin{cases} \mathcal{R}[\text{rel}](\text{SourceField}, s), & \bigwedge_{c \in cs} \mathcal{C}[c](s) \\ \text{true}, & \text{otherwise} \end{cases} \\ \\ & \mathcal{R}[\text{IF } cs \text{ rel ELSE } rel'](\text{SourceField}, s) \\ &= \begin{cases} \mathcal{R}[\text{rel}](\text{SourceField}, s), & \bigwedge_{c \in cs} \mathcal{C}[c](s) \\ \mathcal{R}[rel'](\text{SourceField}, s), & \text{otherwise} \end{cases} \\ \\ & \mathcal{W}[\text{TargetField} = \text{CONST}(C)](s, t) = \pi_{\text{TargetField}}(t) = C \\ & \mathcal{W}[\text{TargetField} = \text{FIELD}(\text{SourceField})](s, t) \\ &= \begin{cases} \pi_{\text{TargetField}}(t) \in_{\text{FlowFilter}} \pi_{\text{SourceField}}(s), & \text{if } \text{SourceField} \in \text{dom}(\Sigma_{\text{FlowFilter}}) \\ \pi_{\text{TargetField}}(t) = \pi_{\text{SourceField}}(s), & \text{otherwise} \end{cases} \\ & \mathcal{W}[\text{TargetField} = \text{FILTER}(F)](s, t) = \pi_{\text{TargetField}}(t) \in_{\text{Filter}} F \\ \\ & \mathcal{C}[\text{SourceField} = \text{CONST}(C)](s) = \pi_{\text{SourceField}}(s) = C \\ & \mathcal{C}[\text{SourceField} = \text{FILTER}(F)](s) = \pi_{\text{SourceField}}(s) \in_{\text{Filter}} F \end{aligned}$$

For filters  $F$  we have used the informal notation  $\pi_{\text{Field}}(r) \in_{\text{Filter}} F$  simply meaning that the value  $\pi_{\text{Field}}(r)$  is within the filter  $F$  (see [18] for a description of filter expressions). Furthermore we have used the notation  $\pi_{\text{TargetField}}(t) \in_{\text{FlowFilter}} \pi_{\text{SourceField}}(s)$  meaning that the value of  $\pi_{\text{TargetField}}(t)$  must be within the value of the `FlowFilter` `SourceField` (more on `FlowFilters`

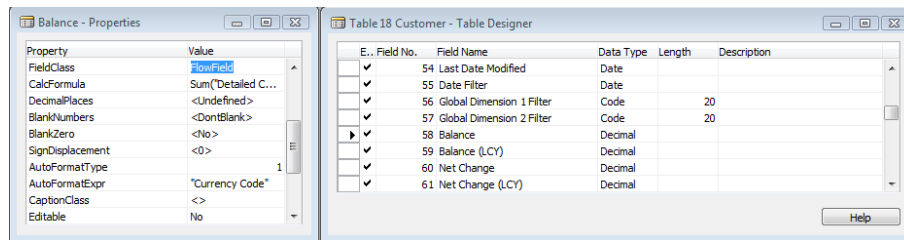
in the next section).

The ability to use FlowFilters in table relations is really somewhat strange: FlowFilters are not persistent in the database and will often have different values on different clients, which may be one of the reasons why NAV does not maintain table relations as invariants.

### 2.3.6 FlowFields and FlowFilters

FlowFields can be thought of as *virtual* fields that are computed/derived values, and thus not actual values in the table. The computation of a value in a FlowField may not only depend on the table in which it is defined, but also *exactly one* other table. We introduce FlowFields by an example:

**Figure 9** A FlowField in the table Customer.



Consider the field **Balance** in the table **Customer** (see Figure 9). It should be evident that the balance of a customer is derived data, based on the transactions for that customer. And indeed this is the case, since **Balance** is defined as a FlowField with the following calculation formula (we have simplified the actual formula):

$$\phi = \text{Sum}(\text{Detailed Cust. Ledg. Entry.Amount} \\ \text{WHERE (Customer No. = FIELD(No.))})$$

This formula expresses that the value of **Balance** is the sum of all **Amount** fields in the **Detailed Cust. Ledg. Entry** table, for which the value of **Customer No.** equals the number of the customer we are currently considering. Cf. Figure 2 (7) this corresponds to having  $\Sigma_{\text{FlowField}}(\text{Balance}) = \phi$ .

So in this example we wish to calculate a sum for each record in the **Customer** table, which can potentially be time consuming, as we are iterating over the **Detailed Cust. Ledg. Entry** table. Fortunately this is where we can use **Sum Index Fields**, which we introduced in Section 2.3.3. In order to utilize **SIFT** in the example, we must define an index on the table **Detailed Cust. Ledg. Entry** – which includes the field **Customer No.** as the first field – such that **Amount** is defined as a **Sum Index Field** for that index. In our semantic model this means that

$$\text{Amount} \in \text{Indexes}_{\text{Detailed Cust. Ledg. Entry}}(\text{Customer No.}, F_1, \dots, \dots, F_n)$$

for some fields  $F_i$ , and indeed such an index is already defined, as the reader can check.

In fact NAV will not allow the computation of a FlowField unless all fields used in the filtering are contained in an index with the field being summed over defined as a Sum Index Field. If the index is not defined, the NAV client will produce a runtime error.

**Hypothesis 8:** *All fields used in the filter for a FlowField must be included in a NAV key with the field being aggregated over defined as a Sum Index Field. If this is not the case, the NAV client will produce a runtime error. It should be (partially) possible to verify this statically<sup>4</sup>.*

In our example we were interested in computing a sum, but FlowFields may compute other functions as well (all with the property that they can be maintained in a B+ tree with logarithmic update/insertion, as was the case for sum). The complete pseudo grammar for FlowField formulae is presented in Figure 10.

---

**Figure 10** BNF grammar for FlowField expressions.

---

```

FlowFieldExp ::= Function(TargetTable[.TargetField] WHERE WhereClause*)

Function      ::= [-] Exist | Count | [-] Sum | [-] Average |
                Min | Max | Lookup

WhereClause   ::= TargetField = CONST(Constant)
                | TargetField = FIELD(SourceField)
                | TargetField = FIELD(UPPERLIMIT(SourceField))
                | TargetField = FIELD(FILTER(SourceField))
                | TargetField = FIELD(UPPERLIMIT(FILTER(SourceField)))
                | TargetField = FILTER(Filter)

```

---

The formal semantics of FlowFields is as follows: Assume we are defining SourceField as a FlowField on the table SourceTable with the FlowField formula  $\phi$  (i.e.  $\Sigma_{\text{FlowField}}(\text{SourceField}) = \phi$  in SourceTable). Then the value of  $\pi_{\text{SourceField}}(s)$  for a given  $s \in \text{SourceTable}$  is  $\llbracket \phi \rrbracket(s)$ :

$$\begin{aligned} & \llbracket f(\text{TargetTable.TargetField WHERE } cs) \rrbracket(s) \\ &= \mathcal{F} \llbracket f \rrbracket (\{ \pi_{\text{TargetField}}(t) \mid t \in \text{TargetTable} \wedge \bigwedge_{c \in cs} \mathcal{W} \llbracket c \rrbracket(s, t) \}) \end{aligned}$$

$$\mathcal{F} \llbracket \text{Exist} \rrbracket (S) = S \neq \emptyset$$

$$\mathcal{F} \llbracket \text{Count} \rrbracket (S) = |S|$$

$$\mathcal{F} \llbracket \text{Sum} \rrbracket (S) = \sum_{s \in S} s$$


---

<sup>4</sup>The reason that we can only partially verify the existence of NAV keys is because NAV keys can be disabled at runtime. Thus the existence of a NAV key does not necessarily imply that it is active.

$$\begin{aligned} \mathcal{F}[\text{Average}](S) &= \frac{\mathcal{F}[\text{Sum}](S)}{\mathcal{F}[\text{Count}](S)} \\ \mathcal{F}[\text{Min}](S) &= \min(S) \\ \mathcal{F}[\text{Max}](S) &= \max(S) \\ \mathcal{F}[\text{Lookup}](S) &= \text{First element}^5 \text{ of } S \end{aligned}$$

$$\begin{aligned} \mathcal{W}[\text{TargetField} = \text{CONST}(C)](s, t) &= \pi_{\text{TargetField}}(t) = C \\ \mathcal{W}[\text{TargetField} = \text{FIELD}(\text{SourceField})](s, t) &= \begin{cases} \pi_{\text{TargetField}}(t) \in \text{FlowFilter } \pi_{\text{SourceField}}(s), & \text{if } \text{SourceField} \in \text{dom}(\Sigma_{\text{FlowFilter}}) \\ \pi_{\text{TargetField}}(t) = \pi_{\text{SourceField}}(s), & \text{otherwise} \end{cases} \\ \mathcal{W}[\text{TargetField} = \text{FIELD}(\text{UPPERLIMIT}(\text{SourceField}))](s, t) &= \pi_{\text{TargetField}}(t) \leq \max(\{x \mid x \in \text{FlowFilter } \pi_{\text{SourceField}}(s)\}) \\ \mathcal{W}[\text{TargetField} = \text{FIELD}(\text{FILTER}(\text{SourceField}))](s, t) &= \pi_{\text{TargetField}}(t) \in \text{Filter } \pi_{\text{SourceField}}(s) \\ \mathcal{W}[\text{TargetField} = \text{FIELD}(\text{UPPERLIMIT}(\text{FILTER}(\text{SourceField})))](s, t) &= \pi_{\text{TargetField}}(t) \leq \max(\{x \mid x \in \text{Filter } \pi_{\text{SourceField}}(s)\}) \\ \mathcal{W}[\text{TargetField} = \text{FILTER}(F)](s, t) &= \pi_{\text{TargetField}}(t) \in \text{Filter } F \end{aligned}$$

The optional “-” allowed in front of some of the functions (which we have left out for simplicity) is negation (with respect to integers and booleans respectively).

FlowFilters are – like FlowFields – virtual. But whereas FlowFields compute a value for each record in the table, FlowFilters are used as *parameters* in the calculation of FlowFields (which is already evident from the semantics of FlowFields).

Again we illustrate the concept by an example: Consider the NAV table **Currency**, which defines the different currencies in the company. This table contains a FlowField **Customer Balance**, which computes the total balance of all customers working in the given currency. If we were only interested in computing the total balance for *all* customers, the following FlowField formula would be sufficient:

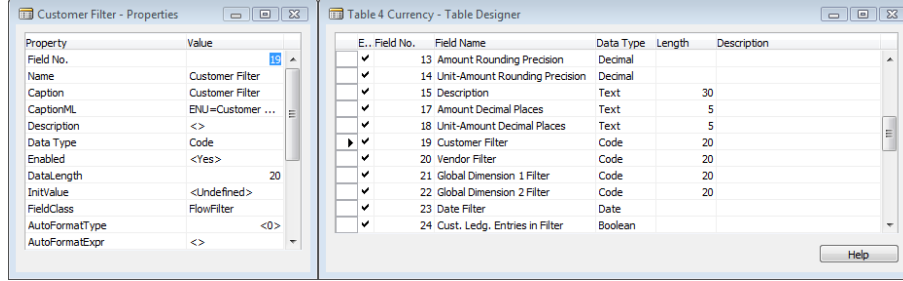
$$\begin{aligned} &\text{Sum}(\text{Detailed Cust. Ledg. Entry.Amount} \\ &\quad \text{WHERE Currency Code} = \text{FIELD}(\text{Code})) \end{aligned}$$

If, however, we only wish to compute the balance for a subset of customers, we need somehow to restrict the ledger entries in the **WHERE** clause, and preferably this restriction should be parametrized: This is exactly what FlowFilters do. Returning to the example, one defines a FlowFilter **Customer Filter** which has the same type as the destination field we want to filter on. In this example we want to filter on the field **Customer No.**, thus **Customer Filter** must have type **Code** [20] (see Figure 11). In our semantic model, Figure 2 (8), we hence have  $\Sigma_{\text{FlowFilter}}(\text{Customer Filter}) = \text{Code}$  [20].

---

<sup>5</sup> $S$  being a set it does not really make sense to talk about “the first element of  $S$ ”. When we write this anyway it is because the selection is with respect to a given ordering of  $S$  – typically the order specified by the primary key on the underlying table

**Figure 11** A FlowFilter in the table Currency.



The FlowFilter Customer Filter can now be used in the formula from earlier to restrict the set of customers, from which to compute the total balance (we call this formula  $\psi$ ):

$$\begin{aligned} & \text{Sum}(\text{Detailed Cust. Ledg. Entry. Amount} \\ & \quad \text{WHERE (Customer No. = FIELD(Customer Filter),} \\ & \quad \quad \text{Currency Code = FIELD(Code))}) \end{aligned}$$

Using the semantics we see that  $\psi$  denotes the following function:

$$\llbracket \psi \rrbracket (s) = \sum_{t \in \text{Detailed Cust. Ledg. Entry, where } P(s,t)} \pi_{\text{Amount}}(t)$$

where

$$\begin{aligned} P(s, t) = & \pi_{\text{Customer No.}}(t) \in_{\text{FlowFilter}} \pi_{\text{Customer Filter}}(s) \wedge \\ & \pi_{\text{Currency Code}}(t) = \pi_{\text{Code}}(s) \end{aligned}$$

which is what we would expect.

Even though the FlowFilter from Figure 11 is presented as a field in the table, it has actually nothing to do with normal fields: It is not part of the underlying SQL table schema, and when a user sets a FlowFilter (via a mutator, cf. Figure 2 (13)) it is never stored in the database. Also note that FlowFilters are properties on tables and not on records, thus the value of a FlowFilter is the same for all records (i.e. in the predicate  $P$  above, the value  $\pi_{\text{Customer Filter}}(s)$  is the same for all  $s \in \text{Currency}$ ).

Figure 12 illustrates this way of thinking about FlowFilters, where the FlowField formula from the last example ( $\psi$ ) is used to compute customer balance based on the Customer Filter FlowFilter (in this example set to the range 5..10). The reader is encouraged to check that the FlowField values are correct with respect to the semantics of  $\psi$ .

One thing that strikes us is the mixing of data definitions (fields) and derived data (FlowFields and FlowFilters) in one NAV object. This makes

**Figure 12** Detailed Cust. Ledg. Entry (top) and Currency (bottom).

	Customer No.	Currency Code	Amount
$t_1 =$	1	DKK	10
$t_2 =$	5	DKK	7
$t_3 =$	10	EUR	2
$t_4 =$	7	EUR	6
$t_5 =$	13	EUR	10
$t_6 =$	6	DKK	4

	Customer Filter	Code	Customer Balance
$s_1 =$	5..10	DKK	$\llbracket \psi \rrbracket(s_1) = \Sigma\{\pi_{\text{Amount}}(t_2), \pi_{\text{Amount}}(t_6)\}$ $= 7 + 4 = 11$
$s_2 =$	5..10	EUR	$\llbracket \psi \rrbracket(s_2) = \Sigma\{\pi_{\text{Amount}}(t_3), \pi_{\text{Amount}}(t_4)\}$ $= 2 + 6 = 8$
$s_3 =$	5..10	USD	$\llbracket \psi \rrbracket(s_3) = \Sigma\emptyset$ $= 0$

table definitions complex, and we believe it also makes maintenance harder. An alternative approach is to define raw data definitions in one NAV object (which is more like a SQL table), and derived data in another. The derived NAV object may then not only extract data from one table, but possibly multiple tables, making it essentially a database view.

**Hypothesis 9:** *Mixing raw data (fields) and derived data (FlowFields) in the same NAV object makes table definitions complex and harder to maintain. Instead: Separate raw data (i.e. SQL table) from derived data (views).*

### 2.3.7 Iteration

The final entry in the class interface for NAV tables (Figure 2, (14)) is an *iterator*. The iterator of a NAV table is a cursor-based implementation of the well-known Iterator Design Pattern [6]. When the cursor of an iterator points to a record in the underlying SQL table, the fields of the record can be accessed, and the record can be modified/deleted.

The most important methods in connection with iteration are: **FIND**, **NEXT**, **INSERT**, **MODIFY** and **DELETE**. **FIND** takes as parameter a *filter expression* (i.e. a predicate on records, see [15]) and sets the cursor to the first record within the filter. **NEXT** moves the cursor to the next record within the current filter, and returns an indication of how many steps the cursor has moved (zero means “no records left”, and the number of steps is with respect to an ordering of the records based on a NAV key<sup>6</sup>). This is illustrated in

<sup>6</sup>It is possible to change the NAV key using the **SETCURRENTKEY** method [15]. The

Figure 13.

**Figure 13** FIND and NEXT operation. NAV table instance (left) and underlying SQL table (right).

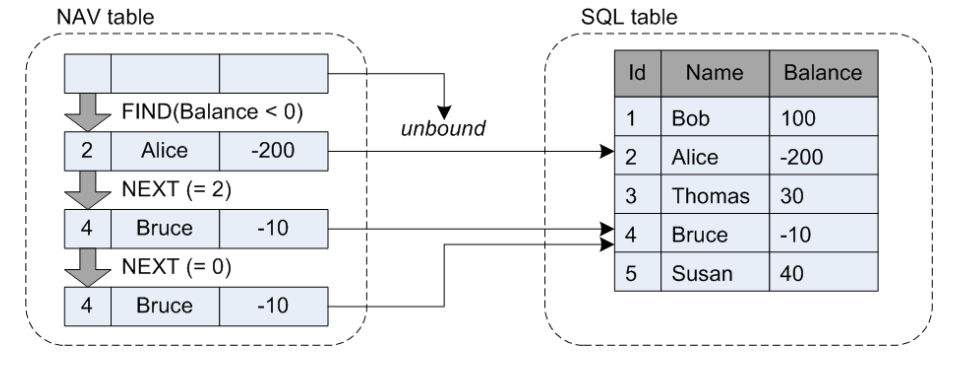


Figure 13 also illustrates how NAV tables maintain a local in-memory copy of the current SQL column. Any (local) changes to this record will not have persistent effect until the changes are committed, either via `MODIFY`, `INSERT` or `DELETE`<sup>7</sup> (see Figure 14, where “time moves vertically”)

We conclude this Section with a brief example on how to use iterators (and in particular how not to use them). Consider a table with  $\Sigma(\text{Name}) = \text{TEXT}[n]$ , for some suitable  $n \in \mathbb{N}$ . If we want to duplicate all entries in this table and add “duplicated” to the name, then the following piece of code will not always work (`t` is an instance of the table we want to duplicate):

```
t.FIND('-'); // Get all records
Repeat
  t.Name := 'Duplicated: ' + t.Name; // Rename
  t.INSERT; // Duplicate current record
UNTIL t.NEXT = 0 // Stop when no records left
```

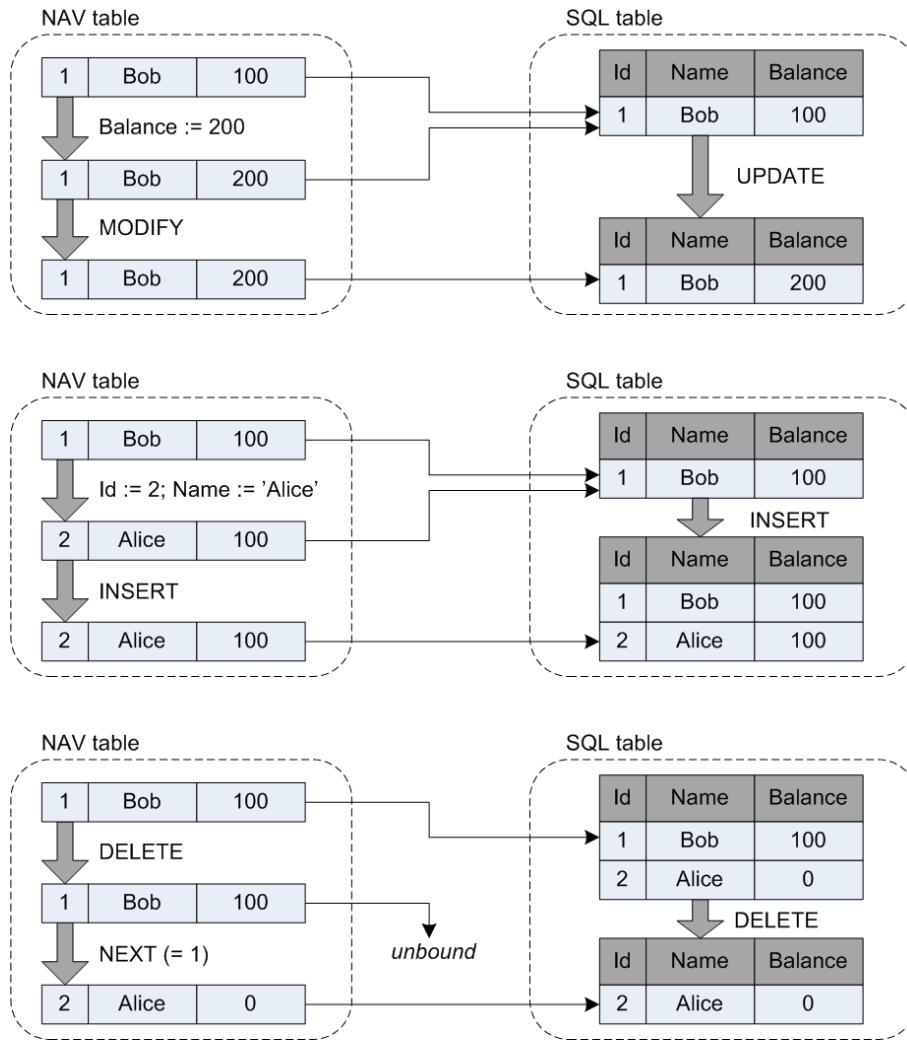
The reason why this piece of code is incorrect is because the invocation of `INSERT` will automatically move the cursor to the newly created record (cf. Figure 14), and thus skip all records in between. To solve this problem, we need two instances of the table (one for iteration and one for insertion), yielding the following correct code:

```
t.FIND('-'); // Get all records
REPEAT
  t2.TRANSFERFIELDS(t); // Copy in-memory fields
  t2.Name := 'Duplicated: ' + t2.Name; // Rename
  t2.INSERT; // Duplicate current record
UNTIL t.NEXT = 0 // Stop when no records left
```

fact that iterators always use a NAV key means in particular that the iteration order is deterministic, unlike the SQL standard.

<sup>7</sup>Followed by an implicit or explicit `COMMIT`, more on this in Section 2.7.1.

**Figure 14** MODIFY, INSERT and DELETE operations. NAV table instances (left) and underlying SQL tables (right).



## 2.4 Codeunits

In this section we will look more into how business logic is defined in Microsoft Dynamics NAV. A codeunit is a collection of methods (procedures in NAV terminology) written in C/AL that are combined in one NAV object, making the logic therein available to other NAV objects. Strictly speaking, codeunits could be removed from NAV, since it is possible to define C/AL procedures on e.g. tables and forms as well, but the idea is to separate business logic from data definition and user interface.

### 2.4.1 Interface

Like NAV tables, codeunits correspond to classes in OOP that can be instantiated. The class interface for codeunits is given in Figure 15, where the only constant entry is the name of the codeunit.

**Figure 15** The semantic model of a NAV codeunit.

<b>Constant</b>	
1. $Name \in String$	(codeunit name)
<b>Per instance</b>	
2. Built-in method	( <b>OnRun</b> )
3. $Vars : String \rightarrow_{\text{fin}} Type$	(user-defined instance variables)
4. $Methods : String \rightarrow_{\text{fin}} Procedure$	(user-defined methods)

Codeunits have one built-in trigger: **OnRun**. This means that codeunits can be executed (e.g. as the result of pushing a button on a form), but from a conceptual point of view the **OnRun** trigger is nothing but a parameterless method.

Apart from procedures (*Methods*), codeunits may contain global variables (*Vars*), which span all procedures within the codeunit. The body of a procedure may be any C/AL *statement* – we describe C/AL in the following section.

### 2.4.2 C/AL

C/AL is an imperative, statically typed programming language, similar to Pascal [9]. *Expressions* in C/AL evaluate to values and *statements* are computations with effects – in particular modifications of (persistent) store. The BNF grammar for expressions (*Exp*) and statements (*Statement*) are given in Figure 16 and Figure 17 respectively – the grammar is simplified in some places to make it more uniform.

We will not explain the semantics of expressions in detail, rather we will explain some of the constructs informally. The first non-selfexplanatory expression is *member access*:

$$Exp_1 . Exp_2$$

Here  $Exp_1$  must evaluate to an object instance (e.g. a table), and  $Exp_2$  must be a member of that objects interface. If e.g.  $Exp_1$  is a variable pointing to an instance of table  $T$  and  $Exp_2 = F \in \text{dom}(\Sigma_T)$ , then  $Exp_1 . Exp_2$  evaluates to the value  $\pi_F(r)$ , where  $r$  is the in-memory copy of the current record (cf. Section 2.3.7).

One important thing to note is that  $Exp_2$  may not be any arbitrary expression: If e.g.  $Exp_1$  is a table variable, then the value of  $Exp_2$  must be known at compile time! This means that one cannot access fields dynamically, which makes it possible to statically type check accesses to the

---

**Figure 16** BNF grammar for expressions.

---

$Exp$	$::= Constant$ $  Identifier$ $  UnaryExp$ $  BinaryExp$ $  NaryExp$	Constants (strings, integers, float- ing points, booleans, etc.) Identifiers (variable names, sys- tem calls, etc.) Unary expression Binary expression <i>n</i> ary expression
$UnaryExp$	$::= (Exp)$ $  NOT Exp$ $  - Exp$	Grouping Negation Unary minus
$BinaryExp$	$::= Exp \oplus Exp$ $  Exp \odot Exp$ $  Exp < Exp$ $  Exp . Exp$ $  Exp .. Exp$ $  Exp :: Exp$	Arithmetic ( $\oplus \in \{+, -, *, /, MOD\}$ ) Boolean combinators ( $\odot \in \{AND, OR, XOR\}$ ) Relations ( $< \in \{<, =, <=, >, >=, <>\}$ ) Member access Range Scope
$NaryExp$	$::= Exp(Exp_1, \dots, Exp_n)$ $  Exp[Exp_1, \dots, Exp_n]$ $  Exp \text{ IN } [Exp_1, \dots, Exp_n]$	Function call Array reference Membership test

---

database (remember that the signature  $\Sigma$  of each table is known at compile time).

This restriction means that it is for instance not possible to write a procedure which takes as input *any* table with a field named `ld` of type  $T$ , and outputs  $\pi_{ld}(r)$ , where  $r$  is the first record in the table (this can almost be achieved using variables of type `RecordRef` – more on this later – however, then no statical type checking is done).

The next non-selfexplanatory expression is *range*:

$$Exp_1 .. Exp_2$$

Here  $Exp_1$  and  $Exp_2$  must evaluate to values of the same type, and the type must be *ordered*. If  $v_i$  is the value of  $Exp_i$  then  $Exp_1 .. Exp_2$  simply denotes the range  $[v_1; v_2]$  (which is why the type must be ordered). Ranges can e.g. be used when setting FlowFilters.

Next we consider *scopes*:

$$Exp_1 :: Exp_2$$

Scopes are used for fields of type `Option`, which are enumeration types in the database. Consider a table signature  $\Sigma$  for a table  $T$ . If  $\pi_1(\Sigma(F)) =$

`Option` and `(OptionString, 'a,b,c')`  $\in \pi_2(\Sigma(F))$ , then `F` is defined as an enumeration with possible values `a`, `b` and `c`. Letting `x` denote an instance of table `T`, we can then access the enumeration values of `F` via `x.F::a`, `x.F::b` and `x.F::c`. This can then be used in a test for the actual enumeration value in `F` for the current in-memory record via:

```
IF x.F = x.F::a THEN
  <Code for case a>
ELSE IF x.F = x.F::b then
  <Code for case b>
ELSE
  <Code for case c>
```

Function calls in C/AL have the form:

$$Exp(Exp_1, \dots, Exp_n)$$

C/AL uses call-by-value function calls, and parameters can either be passed by reference or as values (how each parameter is passed is specified in the functions formal parameters definition). Each expression  $Exp_1, \dots, Exp_n$  must evaluate to a simply-typed value (e.g. integer or bool) or be a table/codeunit instance, thus it is not possible to pass e.g. a function as a parameter (functions are not first class values). The value of  $Exp$  must be known at compile time in order to statically type check function calls. Functions can be called recursively.

C/AL includes fixed size, multidimensional arrays (i.e. the size must be known at compile time). Array access has the form

$$Exp[Exp_1, \dots, Exp_n]$$

which makes it possible to retrieve and store values in the array.

The semantics of statements should be somewhat clear, so we will not get into detail (again we refer to [15] for an informal description). Having defined expressions and statements informally, we can now define procedures:

```
Procedure ::= [LOCAL] PROCEDURE Identifier([VAR] Var1, ..., [VAR] Varn) [: Type];
            VAR Var'1, ..., Var'k
            BEGIN Statement END;
Var        ::= Identifier : Type
```

The optional keyword `LOCAL` makes a procedure available only to the NAV object in which it is defined (private method in OOP terminology). The optional keyword `VAR` in the formal parameter list specifies whether the parameter should be passed by reference (`VAR` set) or as a value. The optional type identifier specifies the return type of the function. Variables  $Var'_1, \dots, Var'_k$  are local variables, which are instantiated each time the procedure is invoked.

---

**Figure 17** BNF grammar for statements.

---

$Statement ::=$	Empty statement
<code>BEGIN Statement END</code>	Block
<code>Statement; Statement</code>	Sequence
<code>Exp ::= Exp</code>	Assignment
<code>Exp</code>	Expression
<code>IF Exp THEN Statement [ELSE Statement]</code>	Conditional
<code>CASE Exp OF Exp<sub>11</sub>, ..., Exp<sub>1k<sub>1</sub></sub> : Statement<sub>1</sub>;</code>	
	⋮
	$Exp_{n1}, \dots, Exp_{nk_n} : Statement_n$
<code>[ELSE Statement<sub>n+1</sub>] END</code>	Case split
<code>WHILE Exp DO Statement</code>	Loop
<code>REPEAT Statement UNTIL Exp</code>	Loop
<code>FOR Exp ::= Exp TO/DOWNTO Exp DO Statement</code>	Bounded loop
<code>WITH Exp DO Statement</code>	Open table
<code>EXIT(Exp)</code>	Function return

---

All variables in C/AL are explicitly typed, which makes type checking easy, and as mentioned earlier, this makes it possible to type check database access. The BNF grammar for types can be seen in Figure 18. We have included a small description for some of the types – for a (more or less) complete description of types see [15].

We mentioned earlier that it is not possible to access fields dynamically; This is however not entirely true. The type `RecordRef` above basically means that a variable points to *any* record of *any* table. It is then possible to iterate over the fields of this record, but then the type system can make no guarantees with respect to correct access.

**Hypothesis 10:** *The strict type annotations of table instance variables imply that code written for a table cannot be used for other tables. If similar code is needed for two different tables, then the programmer must either duplicate the code, or merge the two tables. Solution: Introduce polymorphism. Rather than having strict type annotations, for instance “ $x : \text{Record } 18$ ”, one could instead use type inference, or explicit subtyping (i.e. “ $x : \text{Record}\langle F : T \rangle$ ” – meaning  $x$  must point to any table with  $\Sigma(F) = T$ ).*

The last thing to note in connection with the type system of C/AL is the type `Variant`. `Variant` is a *dynamic* type, which means that it can be instantiated with values of different types (see [15] for a list of allowed types). This means that runtime type errors can occur, and thus C/AL is not strongly typed (as was already witnessed via the `RecordRef` type).

Apart from the conceptual grammar given in this section, C/AL contains tons of built-in system calls (for instance GUI-messaging functionality). We

---

**Figure 18** BNF grammar for types.

---

<i>Type</i>	::=	<i>SimpleType</i>	
		<i>ComplexType</i>	
<i>SimpleType</i>	::=	<b>Integer</b>	
		<b>Text</b> [ <i>n</i> ]	A string of maximum length <i>n</i>
		<b>Code</b> [ <i>n</i> ]	A code of maximum length <i>n</i>
		<b>Char</b>	
		<b>Decimal</b>	
		<b>Option</b>	Enumeration type
		<b>Boolean</b>	
		<b>Date</b>	
		<b>Time</b>	
		<b>DateTime</b>	
		<b>Binary</b> [ <i>n</i> ]	Binary data of maximum length <i>n</i>
		<b>BLOB</b>	Large binary data
		<b>DateFormula</b>	Date formulae
		<b>TableFilter</b>	<i>Unknown</i>
		<b>BigInteger</b>	
		<b>Duration</b>	Difference between two <b>DateTime</b> 's
		<b>GUID</b>	Global Unique Identifier
		<b>RecordID</b>	Id (name) of a NAV table
<i>ComplexType</i>	::=	<b>Action</b>	<i>Unknown</i>
		<b>RecordRef</b>	Reference to <i>any</i> record
		<b>Dialog</b>	Dialog window
		<b>Variant</b>	Dynamic type (!)
		<b>InStream</b>	
		<b>OutStream</b>	
		<b>FieldRef</b>	Reference to <i>any</i> field
		<b>KeyRef</b>	Reference to a key definition in <i>any</i> table
		<b>File</b>	
		<b>[TEMPORARY] Record</b> <i>n</i>	A (temporary) instance of table <i>n</i>
		<b>Codeunit</b> <i>n</i>	An instance of codeunit <i>n</i>
		<b>Form</b> <i>n</i>	An instance of form <i>n</i>
		<b>Report</b> <i>n</i>	An instance of report <i>n</i>
		<b>XMLport</b> <i>n</i>	An instance of XMLport <i>n</i>
		<b>Automation</b> <i>identifier</i>	Automation object (see [15], Chapter 19)
		<b>OCX</b> <i>identifier</i>	OCX object (see [15], Chapter 19)
		<b>ARRAY</b> [ <i>n</i> <sub>1</sub> , ..., <i>n</i> <sub><i>k</i></sub> ]	
		<b>OF</b> <i>Type</i>	Multidimensional, fixed size array
		<b>'enum</b> <sub>1</sub> , ..., <i>enum</i> <sub><i>n</i></sub> <b>'</b>	(Constant) enumeration type
		<b>TextConst</b> <i>String</i>	Multilanguage string

---

have deliberately left these out in this document, and we refer to [15] and [18] for a more complete description. A complete grammar for C/AL (including complete grammars for tables, codeunits, forms and reports) can be found in Appendix B. The grammar has been constructed by reverse engineering in collaboration with Michael Nissen (DIKU) – to our surprise no official grammar existed before our attempt! The grammar is specified in F#'s parser generator language [19]. The tedious work on constructing the parser has opened up for a wide range of interesting topics for future work (Section 5).

## 2.5 Forms

In this section we briefly discuss forms in NAV. A form in NAV is much like a Windows Form [13]: A form contains various controls for displaying and altering data. Like tables and codeunits, forms have triggers (but unlike tables these are “proper” triggers) – some of which have form scope (i.e. one per form) and some of which have control scope. Triggers can be used to activate C/AL code (e.g. when the user pushes a specific button), and like tables and codeunits, forms can contain user-defined methods and variables as well.

### 2.5.1 Interface

The class interface for a NAV form is presented in Figure 19.

**Figure 19** The semantic model of a NAV form.

<b>Constant</b>	
1. $Name \in String$	(form name)
2. $SourceTable \in String + \{unbound\}$	(source table)
3. $C : \mathbb{N} \rightarrow_{\text{fin}} Control \times \mathcal{P}(Property)$	(controls)
<b>Per instance</b>	
4. Built-in methods	( <code>OnOpenForm</code> , <code>OnNextRecord</code> , etc.)
5. $Vars : String \rightarrow_{\text{fin}} Type$	(user-defined instance variables)
6. $Methods : String \rightarrow_{\text{fin}} Procedure$	(user-defined methods)

All forms have a name (1) and can be bound to *at most one* table (2). Having a form bound to a table means that each instance of the form implicitly has an instance of the bound table, from which data can be shown. Besides from making the table instance visible in the source code of the form, the runtime system automatically generates GUI access to filters, and automatically invokes table triggers (this is why the “triggers” of Section 2.3.2 are really GUI triggers).

Figure 20 shows the NAV form `Customer Card`, which shows information about a given customer. The form is bound to the table `Customer`, which means that one can iterate through all customers using `[Pg Up]` and `[Pg Dn]`. But it also makes it possible to change customer data, by changing the value in one of the controls – and this will automatically invoke the appropriate NAV trigger(s) on the `Customer` table. If we e.g. change the value in the text box `Name`, then the runtime system will automatically invoke the `OnModify` trigger for the field `Name`. Similarly, if we change the value in the text box `No.`, then the `OnRename` trigger will automatically be invoked, as `No.` is part of the primary key on the `Customer` table.

Having a form bound to a table does not automatically imply that all fields of the table are presented on the form. All visible controls on the form must (in principle) be added manually, and it is then possible to hook up e.g. a text box to a field of the table. In Figure 20 we have for instance

**Figure 20** NAV form Customer Card

that  $\pi_1(\mathcal{C}(2)) = \text{TextBox}$  and  $(\text{SourceExpr}, \text{No.}) \in \pi_2(\mathcal{C}(2))$ , meaning that the control with identifier 2 is a TextBox, and it is bound to the field No. of the underlying table, Customer.

It is possible to auto generate the controls for a new form, based on the table it is bound to. But when an underlying table is updated, there is no means of automatically updating the existing form(s) bound to the table.

**Hypothesis 11:** *Forms can only be bound to a single table, which exacerbates the use of unnormalized database design: If a developer wishes to show the compound data from two tables in a form, then the easiest solution is to simply merge the two tables into a (Cartesian) product table. Solution: Generalize source tables to source views instead.*

## 2.6 Reports

The last NAV object type we will describe in this section is reports. We will not discuss layout issues, instead we will focus on how data retrieved and calculated.

### 2.6.1 Interface

The semantic model of a NAV report is presented in Figure 21.

Like forms, reports are bound to tables. But unlike forms, reports can in general be bound to several tables, which are called data items. Data items thus define the data from which reports are calculated. When there are no dependencies between the data items, nor any post-processing of the retrieved data, then the output from a report is simply

$$\bigcup_{n \in \text{dom}(\text{DataItem})} \sigma(\pi_1(\text{DataItem}(n)))$$

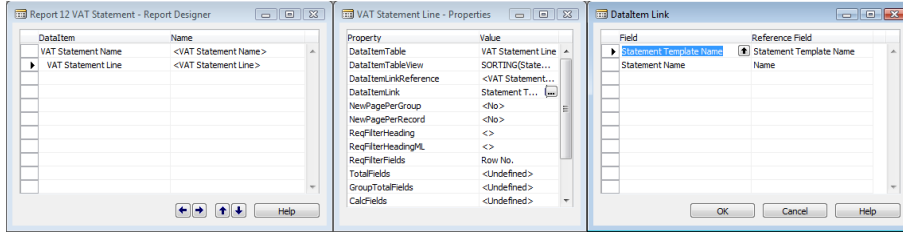
**Figure 21** The semantic model of a NAV report.

<b>Constant</b>	
1. $Name \in String$	(table name)
2. $DataItem : \mathbb{N} \rightarrow_{\text{fin}} String \times \mathcal{P}(Property)$	(data items)
<b>Per instance</b>	
3. Built-in methods	( <code>OnPreReport</code> , <code>OnAfterGetRecord</code> , etc.)
4. $Vars : String \rightarrow_{\text{fin}} Type$	(user-defined instance variables)
5. $Methods : String \rightarrow_{\text{fin}} Procedure$	(user-defined methods)

(here we use the informal notation  $\sigma(\pi_1(DataItem(n)))$  denoting all records in the table with the name  $\pi_1(DataItem(n))$ ).

It is possible to nest data items, which means that a data item can depend on *exactly* one other data item. We illustrate this by an example: Consider the NAV report VAT Statement in Figure 22.

**Figure 22** NAV report VAT Statement.



This report has one *root* data item, VAT Statement Name, and one nested data item, VAT Statement Line. In the semantic model this means that

$$\begin{aligned}\pi_1(DataItem(1)) &= \text{VAT Statement Name} \\ \pi_1(DataItem(2)) &= \text{VAT Statement Line}\end{aligned}$$

Furthermore, the second data item is nested under the first data item, which in the semantic model means that

$$(\text{DataItemLinkReference}, 1) \in \pi_2(DataItem(2))$$

i.e. the second data item *links* to the first data item. This means that the VAT Statement Line table is iterated (inner loop) for each record in the VAT Statement Name (outer loop). However, we do not iterate through all records in VAT Statement Line, instead we only iterate through those records that meet certain join conditions.

In the example above, the join condition is that for each record  $n$  in the VAT Statement Name table, we iterate through those records  $l$  in the VAT Statement Line table that satisfy

$$\begin{aligned}\pi_{\text{Statement Template Name}}(l) &= \pi_{\text{Statement Template Name}}(n) \\ \pi_{\text{Statement Name}}(l) &= \pi_{\text{Name}}(n)\end{aligned}$$

(this is specified in the `DataItemLink` property, cf. Figure 22). But this means that we wish to process exactly the records (again using notation from relational algebra)

$$\text{VAT Statement Name} \bowtie_P \text{VAT Statement Line}$$

where  $P$  is the predicate

$$\text{Statement Template Name} = \text{Statement Template Name} \wedge \text{Name} = \text{Statement Name}$$

In general the `DataItemLink` property can only depend on the immediate parent data item, thus all data in NAV reports correspond (in principle) to pairwise joined tables.

In the example above, one would imagine that the following query would be sent to Microsoft SQL Server to generate the data set:

```
SELECT * FROM [VAT Statement Name], [VAT Statement Line]
INNER JOIN [VAT Statement Line] ON
    [VAT Statement Name].[Statement Template Name] =
    [VAT Statement Line].[Statement Template Name] AND
[VAT Statement Name].Name =
    [VAT Statement Line].[Statement Name]
```

But after doing a database trace, it turned out that all joins were performed manually via nested loops, instead of utilizing joins and automatic query optimization.

**Hypothesis 12:** *The implementation of reports in Microsoft SQL Server does not utilize joins and the performance gains possible if the joined fields are indexed. Instead data is retrieved via nested looping making it asymptotically slower.*

As mentioned earlier, it is possible to do post-processing of the data retrieved via data items. This is done via triggers, which can have either report, data item or record scope. For example, all data items have an `OnAfterGetRecord` trigger, which is executed after the retrieval of a record from the corresponding table. Since this trigger can contain arbitrary C/AL code, one could for instance access another table (not specified in the *DataItem* map), and include data from this table in the report. However, we do not believe this is often used, as all data relevant to a report should really be defined as data items – all other parameters used in the report should contain static values. This suggests that C/AL is perhaps too expressive for post processing, and maybe a less expressive language should be used ([16] is an example of a declarative, domain specific language for ERP reporting).

**Hypothesis 13:** *The underlying data set used in a NAV report can be expressed directly as a statement in SQL using joins. However, the post-*

processing can contain arbitrary C/AL code, making the report infeasible for direct conversion to e.g. SQL.

**Hypothesis 14:** NAV developers are already familiar with table joins from reports. The same user interface could be used to introduce views in NAV, making it more feasible to use normalized database design (the developer would not need to hand-write table joins).

## 2.7 Runtime system (C/SIDE)

C/SIDE is the runtime system of NAV. By runtime system we mean the *client*, since it is the client that is running all business logic (cf. the introduction). Having described the “building blocks” of NAV as classes in OOP, the runtime system corresponds to e.g. the virtual machine of Java, since it executes instances of compiled NAV objects.

More specifically, the C/SIDE client interprets compiled C/AL code for codeunits, generates GUI for forms and reports, and links NAV tables to their underlying SQL tables.

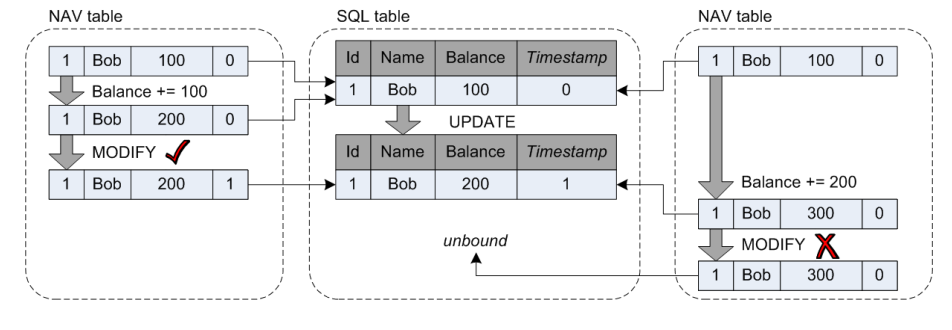
### 2.7.1 Concurrency

Microsoft Dynamics NAV is – as mentioned in Section 1 – a multiuser system, thus it has to deal with concurrency. We shortly describe the facilities in NAV used to avoid concurrency problems (again we will only consider the case where NAV is running on Microsoft SQL Server).

The first thing we describe is how NAV handles *lost updates*. Consider the following example: Two processes (object instances) wish to update the same field in the same record, by incrementing the field by 100 and 200 respectively. When processed in serial, the result will be a total incrementation of 300. However, when processed in parallel, the total result can be an incrementation of either 100, 200 or 300 – the two first cases are due to lost updates. NAV handles this by adding time stamps to all(!) records in the database. Each time the record is updated, the time stamp is updated as well, and the update is only carried out if the time stamp has not been updated in the meantime. This scenario is described in Figure 23, where the right process will not be able to commit its update, as the left process has made an intermediate update (we have used integers for time stamps).

In other words, Microsoft Dynamics NAV uses *optimistic concurrency control* [2]. If an update fails due to an incorrect version, then all changes to the database in the current transaction are not committed. C/SIDE has built-in support for transactions: each execution of C/AL code from the user interface starts with an implicit “start write transaction”, and the transaction is either committed explicitly via `COMMIT` or implicitly upon completion of the execution (when control is given back to the user in the

**Figure 23** Time stamps are used to prevent lost updates.



user interface). This means that either *all* updates take place, or *no* updates take place. Thus if e.g. the right process in Figure 23 had made another update prior to the failing MODIFY without first COMMIT'ing the update, then that update would not be executed.

Transactions are implemented by acquiring *locks*, however it is unclear to us exactly what is locked and when it is locked. [15] has some information on this, but it is still very unclear what locking principles are used.

**Question:** *Microsoft Dynamics NAV does not scale well beyond 250 users. Is this due to the specific concurrency control protocol used in the database? If so, is it then because of the method used for locking or because of the unnormalized database design (decomposition of tables could minimize the data that actually needs to be locked)?*

### 2.7.2 NAV 2009

As mentioned in the beginning of this document, we have used Microsoft Dynamics NAV 5.0 SP1 throughout our analysis. A new version of NAV – Microsoft Dynamics NAV 2009 – has however recently been released. So will our analysis make any sense in the new setting?

One of the new things in NAV 2009 is the use of a three-tier architecture, i.e. all business logic is moved from the client to running centrally on an application server. It is, however, the same C/AL code that is being executed as in the old version of NAV. So even though NAV 2009 introduces some new features (such as a “Role-tailored client” and SQL reports) it still uses the exact same database scheme and business logic (and data source for reports).

We therefor claim that the observations made in this document are still valid for the new version of NAV.

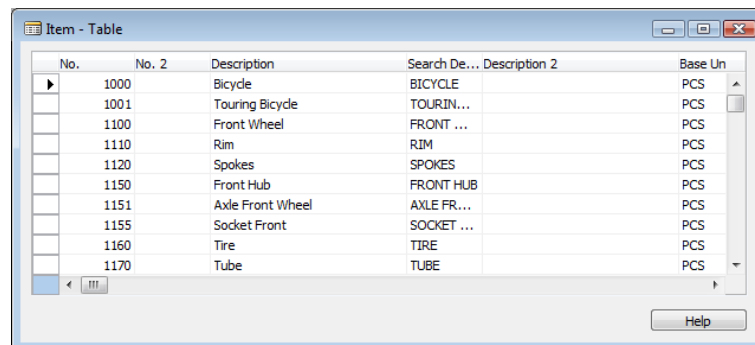
## 3 NAV Design Patterns

In the last section we described what the different NAV objects are. In this section we wish to describe how these object are (typically) used (i.e. NAV design patterns). We classify different kinds of tables, codeunits, forms and reports – note however that these are only design *guidelines*; In principle the programmer can design NAV objects in any way desirable, using the possibilities described in the previous section. The division of NAV objects described in this section is based on a similar division in [18].

### 3.1 Tables

#### 3.1.1 Master tables

Master tables contain primary/master data, such as customers (table 18) and items (table 27). Rows in a master table can be thought of as *entities*, and since NAV lacks joins, these tables will often contain many columns and be sparse (many null values). The Items table for instance contains 175 columns! Master tables typically use card forms as primary input (will be explained later).



No.	No. 2	Description	Search De...	Description 2	Base Un
1000		Bicycle	BICYCLE		PCS
1001		Touring Bicycle	TOURIN...		PCS
1100		Front Wheel	FRONT ...		PCS
1110		Rim	RIM		PCS
1120		Spokes	SPOKES		PCS
1150		Front Hub	FRONT HUB		PCS
1151		Axle Front Wheel	AXLE FR...		PCS
1155		Socket Front	SOCKET ...		PCS
1160		Tire	TIRE		PCS
1170		Tube	TUBE		PCS

#### 3.1.2 Templates

Templates are used in connection with journals (below). A row in a template table characterizes a set of rows in a journal, by describing common control information such as type of journal, numbering series, account numbers, etc. An example of a template table is table 80 (Gen. Journal Template). This table divides general journals into classes, such as “assets”, “payments”, etc. Templates can be thought of as *configuration*, as they are not often changed.

Name	Description	Test Repo...	Form ID	Posting R...	Source Code	Recurring	Force D
ASSETS	Fixed Asset G/L Journal	2	5628	3	FAGLJNL		✓
CASHRCPT	Cash receipts	2	255	3	CASHRE...		✓
GENERAL	GENERAL	2	39	3	GENJNL		✓
INTERCOMP	Intercompany	2	610	3	INTERCO...		✓
JOB	Job G/L Journal	2	1020	3	JOBGLJNL		✓
PAYMENT	Payments	2	256	3	PAYMEN...		✓
PURCH	Purchases	2	254	3	PURCHJNL		✓
RECURRING	Recurring General Journal	2	283	3	GENJNL	✓	✓
SALES	Sales	2	253	3	SALESJNL		✓

### 3.1.3 Journals

Journals contain *unposted* data, i.e. information that may possibly change before it is posted to the ledger. Each row in a journal is associated with a row in a template table (above). Table 81 (Gen. Journal Line) is an example of a journal table, which contains all lines not yet posted to the general ledger. Journals typically use tabular forms as input (will be explained later).

Journal T...	Journal B...	Line No.	A... Account No.	Posting Date D...	Document...	Description
GENERAL	DEFAULT	460000	G.. 5750	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	470000	G.. 8510	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	480000	G.. 5710	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	490000	G.. 5750	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	500000	G.. 8510	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	510000	G.. 5710	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	520000	G.. 5750	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	530000	G.. 8510	25-01-08	G00002	Invoice no. 156786 for Gasc
GENERAL	DEFAULT	540000	B.. WWB-OP...	25-01-08	G00002	Invoice no. 156786 for Gasc
JOB	DEFAULT	20000	G.. 8430	25-01-08	G07001	Road Toll

### 3.1.4 Ledgers

Ledger tables contain *posted* data, i.e. events that cannot – and in fact, by law, should not – be deleted (in some cases, however, they can be *reversed*). Data in ledger tables is considered “accounting data”, i.e. data which is used for auditing. An example of a ledger table is G/L Entry (table 17). Data cannot be inserted manually in a ledger table, instead journal lines are posted via a *posting routine* to become ledger lines. The posting routine then makes sure that all entries are balanced against dual entries, maintaining the invariants of *double-entry bookkeeping* [20, 11].

Entry No.	G/L Account	Posting Date	Document	Description	Bal. Account	Amount	Department
30	5830	C31-12-06	START	Opening Entry		-14.702,43	
31	5840	C31-12-06	START	Opening Entry		-4.518,40	
32	5920	C31-12-06	START	Opening Entry		-45.552,00	
33	3120	C31-12-06	START	Opening Entry		-488.329,71	
34	1320	01-01-07 I...	108009	Order 106015		30.000,00	ADM
35	5630	01-01-07 I...	108009	Order 106015		7.500,00	ADM
36	5410	01-01-07 I...	108009	Order 106015		-37.500,00	
37	2940	01-01-07 P..	108009	Order 106015	44127914	-37.500,00	
38	5410	01-01-07 P..	108009	Order 106015	GIRO	37.500,00	
39	1220	01-01-07 I...	108010	Order 106018		6.600,00	PROD

### 3.1.5 References

Reference tables are somewhat like master tables, only data is always referred rather than copied. An example of a reference table is Post Code (table 225), which contains a list of postal codes. If this table for instance contains the record (DK-2300, Copenhagen South), and an address refers to the code DK-2300 (which is the primary key), then any changes to the postal code will automatically be reflected in the address.

Opposed to this, data from the customer table is copied, for instance when a sales order is created. This is because the address on a sales order must be that of the time when the order was established, and not the present address (which may be different).

Code	City	Search City
AT-1100	Wien	WIEN
AT-1230	Wien	WIEN
AT-2355	Wr. Neudorf	WR. NEU...
AT-4810	Gmunden	GMUNDEN
AT-5730	Mittersill	MITTERSILL
AT-8850	Murau	MURAU
AU-2000	Sydney, NSW	SYDNEY, ...
AU-2500	Wollongong, NSW	WOLLON...
AU-3000	Melbourne, VIC	MELBOU...
AU-4000	Brisbane, QLD	BRISBAN...
AU-6800	Perth, WA	PERTH, WA

### 3.1.6 Registers

Register tables are used to group entries in ledger tables that belong to the same posting. Whereas ledger entries contain *posting dates*, registers contain the actual date the entries were posted (which need not be the same). Registers are used for auditing – an example is table 45 (G/L Register), which groups entries in the G/L Entry table.

No.	From Entr...	To Entry No.	Creation...	Source Code	User ID	Journal B...	From VAT...	To VAT En...
1	1	33	31-12-07	START		G/L OPEN	1	
2	34	38	31-12-07	PURCHA...			1	
3	39	43	31-12-07	PURCHA...			2	
4	44	200	31-12-07	START		PERIODIC	3	
5	201	205	31-12-07	PURCHA...			60	
6	206	209	31-12-07	START		DEPR	61	
7	210	214	31-12-07	PURCHA...			61	
8	215	219	31-12-07	PURCHA...			62	
9	220	380	31-12-07	GENJNL		DEFAULT	63	
10	381	385	31-12-07	PURCHA...			122	

### 3.1.7 Posted documents

Posted documents are – like registers – used to group ledger entries. But whereas registers serve auditing purposes, posted documents are only used for easier reference to the posted data (and can thus in principle be deleted).

Posted documents are examples of the “history problem”. By history problem is meant the fact that some data needs to be copied for historical reasons, e.g. the address of a customer or the name of an item on an invoice, as explained earlier.

**Hypothesis 15:** *The history problem is handled in NAV by explicit copying of data. This implies that a change in the data definition of historic data must be changed in all places where it is copied (since the data definition/schema is copied as well). An alternative solution is to use versioning, where historical data refers to a given version of data rather than copying it – avoiding propagation of changes when upgrading.*

An example of a table used for posted documents is Sales Invoice Line (table 113), which describes items that were sold. Ledger entries are linked to their originating documents via a document number, making it possible to navigate from a ledger entry on a sales account to the sales invoice from which it originated.

Document...	Line No.	Sell-to Cu...	T... No.	Location...	Posting Gr...	Shipment...	Description
103001	10000	10000	R.. TIMOTHY	BLUE			21-01-08 Assembling Furnitur
103001	20000	10000	R.. TIMOTHY	BLUE			21-01-08 Assembling Furnitur
103002	10000	20000	R.. TIMOTHY				21-01-08 Assembling Furnitur
103002	20000	20000	R.. TIMOTHY				21-01-08 Assembling Furnitur
103003	10000	30000	R.. TIMOTHY				21-01-08 Assembling Furnitur
103003	20000	30000	R.. TIMOTHY				21-01-08 Assembling Furnitur
103005	10000	10000	I... 1968-S	RED	RESALE		03-01-08 MEXICO Swivel Che
103005	20000	10000	I... 1996-S	RED	RESALE		03-01-08 ATLANTA Whiteboz
103006	10000	42147258	I... 1972-S	RED	RESALE		11-01-08 MUNICH Swivel Che
103006	20000	42147258	I... 1968-S	RED	RESALE		11-01-08 MEXICO Swivel Che



Examples of table independent libraries include codeunit 365 (Format Address), codeunit 397 (Mail) and codeunit 412 (Common Dialog Management).

### **3.2.2 Table dependent libraries**

The second codeunit pattern is the dual of table independent libraries, namely “table dependent libraries”. Table dependent libraries are somewhat similar to bound forms (Section 2.5) in that they provide functionality for a set of tables which are logically related. What we mean by this is tables that typically belong to the same functional area (e.g. accounting, orders, etc.).

Examples include codeunit 3 (G/L Account-Indent, using tables G/L Account and IC G/L Account), codeunit 60 (Sales-Calc. Discount, using e.g. tables Cust. Invoice Disc., Sales Header and Sales Line) and codeunit 220 (Resource-Find Cost, using table Resource Cost).

### **3.2.3 Posting routines**

The final pattern we have identified is posting routines, which is a special case of table dependent libraries. Posting routines are typically divided into three codeunits:

1. Check line
2. Post line
3. Post lines

A posting routine takes a set of journals (Section 3.1.3) and posts them as ledger entries (Section 3.1.4), and group them together in one register (Section 3.1.6). This is done by calling (3), which for each unposted line calls (1) to check that the line is valid for posting, and if so calls (2) to do the actual posting.

Codeunit 11 (Gen. Jnl.-Check Line), codeunit 12 (Gen. Jnl.-Post Line) and codeunit 13 (Gen. Jnl.-Post Batch) make up a posting routine for posting general journals.

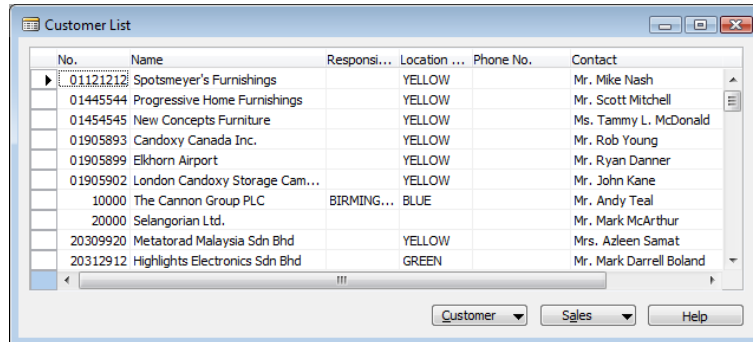
## **3.3 Forms**

### **3.3.1 Card forms**

A card form is used to display a single record from the table to which it is bound. Card forms typically show data from master tables or setup tables. Form 21 (Customer Card, cf. Figure 20) is an example of a card form, which is bound to the Customer table.

### 3.3.2 Tabular/list forms

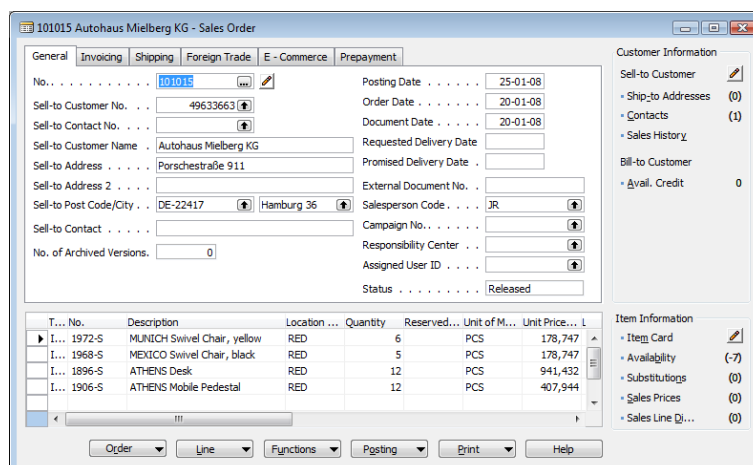
Tabular/list forms are used to show a list of records from the table to which it is bound. Typically one has a list form for a table exactly when there is also a card form for that table. Form 22 (Customer List) is an example of a list form.



### 3.3.3 Header/detail forms

Header/detail forms are somewhat different from card and list forms since they can show data from two tables. The form is still bound to a single table T1 (the header information), but it contains a sub form (a list), which is bound to another table T2 (the detail information). This list contains those rows of T2 that are related to the current row of T1, i.e. there is a one-to-many relation between T1 and T2.

Form 42 (Sales Order) is an example of a header/detail form, which is bound to the table Sales Header. The detailed information is *sales lines*, which are retrieved from the Sales Line table (the one-to-many mapping is on the field No. of Sales Header and Document No. on Sales Line).



### 3.4 Reports

In our description of reports in Section 2.6 we were only interested in the data aspects of a report, and not the visualization of data. In this section we will follow that approach, and make a division only on how/what data is retrieved, and not in what ways it can be visualized.

In [18] reports are divided into “lists”, “documents”, “transactions”, “tests” and “postings”. In our opinion this distinction is entirely on the visualization level, and data is retrieved in the exact same way in all cases (as mentioned in Section 2.6, the data used in reports is in fact always a selection with joins).

There is however one important distinction in reports, namely those that are *read-only*, and those that are *processing-only*. Read-only are the most normal reports, and one can argue that these are in fact the only “proper” reports, in the sense that a report should be a side-effect free function on data, and in particular not modify data. An example of a read-only report is report 111 (Customer - Top 10 List), which calculates a list of customers who bought most.

Customer - Top 10 List					24. September 2008	
Period:					Page 1	
CRONUS International Ltd.					sa	
Ranked according to Sales (LCY)						
Rank	No.	Name	Sales (LCY)	Balance (LCY)	Portion of Sales (LCY)	
1	10000	The Cannon Group PLC	17,100.96	168,364.41	.....	
2	47563218	Klubben	11,772.20	11,772.20	.....	
3	20000	Selangorian Ltd.	6,510.64	90,049.99	.....	
4	30000	John Haddock Insurance Co.	6,142.90	349,615.40	.....	
5	32656665	Antarcticoopy	2,592.81	2,592.81	.....	
6	43687129	Designstudio Gmunden	2,498.10	2,498.10	.....	
7	35963652	Heimilispriidi	2,024.21	2,024.21	.....	
8	42147258	BYT-KOMPLET s.r.o.	1,602.90	1,602.90	.....	
9	01445544	Progressive Home Furnishings	1,499.02	1,499.02	.....	
10	40000	Deerfield Graphics Company	1,063.10	1,328.88	.....	
<b>Total</b>			<b>52,796.84</b>	<b>637,337.92</b>		
Total Sales			55,162.67	872,281.98		
% of Total Sales			95.7	73.1		

Processing-only reports are used to update data, and will not produce an output, and as mentioned we find the “report” terminology misleading.

**Hypothesis 16:** *Processing-only reports exist because reports have the ability to join tables (via data items), making it easier to traverse a set of tables, rather than having to hand-write join procedures in a codeunit. However, processing-only reports have the same performance problems as mentioned in Hypothesis 12.*

Report 794 (Adjust Item Costs/Prices) is an example of a processing-only report, which updates the price of a set of items with respect to an adjustment factor.

## 4 Architectural Redesign

In this last section we describe our attempt at transforming the existing NAV architecture into a modularized design, and why that approach did not work. We conclude with proposing another method, which relies on expanding the existing functionality described in Section 2.

One might ask why we are at all interested in having a modularized design, and what exactly is meant by such a design. Today all NAV functionality is spread across tables, codeunits, forms and reports – as described in Section 2 – with no logical grouping of functionality (e.g. customers, currency management, VAT management, etc.). This means that for instance VAT functionality is spread across multiple NAV objects, and it is not possible to see exactly which NAV objects are involved without inspecting the code. This is the first reason why modularization is desirable; simply to make it easier for developers to find the source code for a particular functionality, which will ultimately lower the time needed for customizations/upgrades and thus lower *total cost of ownership* (TCO).

Another aspect is *updates*. Having a modularized design in which all modules have a formally defined interface (e.g. a class interface), it should be possible to update such a module – preserving the interface – and then plug in the updated version in the ERP system, without having to worry about the other modules. And to make the update even more reliable, we suggest using *behavioral interfaces* (e.g. session types [8]), which not only specify the legal module invocations, but also the order in which methods should be invoked.

But in fact we believe modularization is not just desirable, we believe it is necessary. Extending and maintaining a big system such as Microsoft Dynamics NAV means that a lot of developers must work on the source code simultaneously. Without a modularized design one can easily imagine the need for simultaneous updates of the same NAV objects, which means that the updates need to be merged (and re-tested). As the tightly-coupled system grows even more complex, this procedure will become even more troublesome, if not impossible. Using a modularized design it should be easier to make non-overlapping updates by having developers working only on disjoint modules (which are in a sense self-contained).

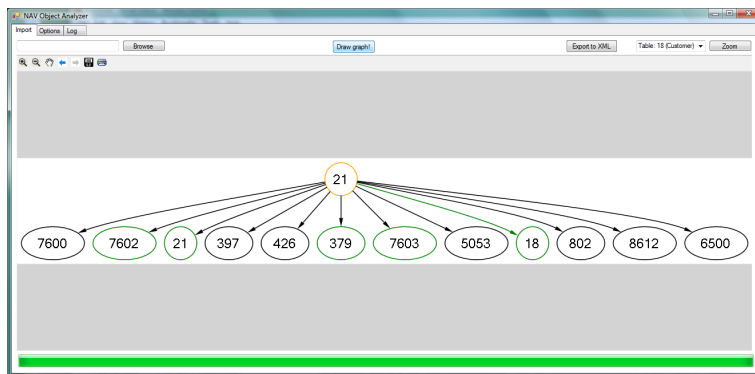
**Hypothesis 17:** *Modularization of the NAV architecture is desirable with respect to maintenance and upgradability, and a necessity when NAV is to be extended with even more features in the future.*

Our approach was based on grouping existing NAV objects into modules. The objects within a module were allowed to have a high level of interdependency, but should be somewhat more loosely coupled from the other modules. In the analysis we needed a tool for computing *dependency graphs*, i.e. a graphical visualization of dependencies between NAV objects.

Since no such tool existed, we built one ourselves, which included object references (in C/AL), table relations and source tables (forms) – but omitting e.g. FlowFields (the tool was created before the development of the C/AL parser, which is needed to include FlowField relations). The tool was restricted to tables, codeunits and forms as well.

Figure 24 shows how the tool is used to compute all dependencies for the Customer Card form (forms are displayed in orange, codeunits in black and tables in green).

**Figure 24** “NAV Object Analyzer” example.



Besides from rendering dependencies between NAV objects, the tool has support for *components*, which are collections of NAV objects (and components). The dependencies of a component is the union of the dependencies of the objects in the component. For instance one can define a “General ledger setup” component (in XML-style notation) via

```
<Component>
  <Name>General ledger setup</Name>
  <Id>1</Id>
  <NAVObjects>
    <NAVObject id="118" type="Form" name="General Ledger Setup" />
    <NAVObject id="98" type="Table" name="General Ledger Setup" />
    <NAVObject id="426" type="Codeunit" name="Payment Tolerance Management" />
  </NAVObjects>
</Component>
```

Using this facility we tried to modularize the architecture via components, but it turned out that the interdependency levels were in general far to high to make a reasonable modularization. For instance the Customer table has around 50 dependencies and around 150 dependants. On average the tool reported around ten dependencies per object – and again only taking into account a subset of dependency types. Thus we believe modularization via simply collecting existing NAV objects in components is not feasible.

So another method is needed in order to modularize the functionality of Microsoft Dynamics NAV. The problem with our approach is that NAV objects often contain functionality/data that is not logically related: Consider

for instance the `Customer` table. This table contains customer data, but it also contains a lot of other data which is not directly customer related. An example is shipping information (ten fields), which is used to describe various information on how to ship goods to the customer. A better approach is to factorize the shipping information into another table (which must be part of the “shipping module”), and then add a relation between customers and shipping information.

The problem with decomposition of tables is that it is not possible to join tables in NAV (as mentioned in Section 2.3). Thus the introduction of joins (or in general views) into NAV is crucial if such decomposition should be made possible. We mentioned earlier the desire for a normalized database design. Besides from ensuring data integrity, this also makes customization/upgrading easier, as changes to table definitions need not be propagated to other copies of the same definitions.

Another problem is *backwards compatibility*. Since NAV is customized by partners, it must be backwards compatible (or at least not hard to update existing customizations to the new architecture). This suggests introduction of a new object type, *view*, which can be used to resemble the old `Customer` table, but such that data is retrieved from multiple underlying tables. This also makes it possible to move all `FlowField` and `FlowFilter` definitions from tables to views, such that tables only contain proper data definitions, and not derived data. It should then be possible to use views in all places where a table was previously used – in particular reports could use a view as data set rather than data items.

In a similar fashion, codeunits need to be modularized as well. This, however, does not imply the need for new object types, as one existing codeunit should be divided into logically different codeunits, which need no relation to each other. For backwards compatibility it is possible to create a resembling codeunit, which calls the functionality in the divided units. This kind of modularization is also referred to as *code refactorization*, for which tools exist. An example is [7], which may also be useful in decomposition of tables.

Finally we must consider forms. The only thing needed is to accommodate the changes made in tables and codeunits. Changes to codeunits is again an instance of code refactorization, and decomposition of tables should be accommodated by allowing views (the new object type) to be the underlying data source, rather than just tables.

Now the question is to what extent this decomposition/refactorization can be done automatically. We do not believe it is possible to fully mechanize the procedure, but we believe the process will be tool-supported. The first thing that is needed is a more fine-grained dependency analysis; rather than registering object dependencies (as is the case for our “NAV Object Analyzer” tool), we need to register dependencies on procedure level (codeunits) and field level (tables). In order to do this analysis, the C/AL parser

from Appendix B is needed.

For tables it would furthermore be interesting to divide dependencies into read-only and read/write, for it is likely that read/write dependencies are logically more related than those that are read-only. This again involves the C/AL parser and an access pattern analysis (`INSERT`, `MODIFY` and `DELETE`, cf. Section 2.3.7).

A final interesting experiment is to apply a *tree decomposition* algorithm [17] to the fine-grained dependency graph, in order to get a first approximation of a modularized design. With such an analysis we will also get an idea of the dependency complexity, which in the graph theoretic setting is known as the *tree width*.

## 5 Conclusion & Future Work

We have presented a detailed, high-level description of the ERP system Microsoft Dynamics NAV. Our presentation is different from existing literature on NAV, as it targets mainly computer scientist. The presentation is given in an object-based fashion, making it useful as a starting point if C/AL is to be ported to an object oriented language.

We have presented some problems regarding performance and upgradability, and given concrete ideas for how these problems can be solved. Our solutions take into account that any modifications of the base product must be backwards compatible. The main contributions of this report are:

- The current implementation of Sum Index Field Technology in Microsoft SQL Server is non-optimal. SIFT is considered one of the main technologies (i.e. selling points) in Microsoft Dynamics NAV, but we believe the competitive advantage has been lost. We propose an asymptotically faster solution.
- Table relations are rather complex, and may result in unintended dangling references. We propose using simple foreign key constraints that are checked automatically by the SQL server (and which prevent dangling references).
- The design of NAV exacerbates an unnormalized database design, witnessed by the strict type annotation in C/AL, lack of database joins and tailoring of forms to single tables. We propose introduction of polymorphism/type inference (or explicit subtyping) to C/AL, and introduction of joins/views.
- The unnormalized database design of NAV makes it hard to modularize functionality into “logically related” components. Introducing views and joins (above) will enable easier decomposition of tables, and provide means for backwards compatibility.

- Data retrieval in reports is sub-optimal. Rather than processing data by means of coded joins (nested looping) the query optimizer of Microsoft SQL Server should be utilized.

## 5.1 Future Work

What is perhaps most interesting about our work in this report, is the possibilities that it opens for future work. In a sense, this report provides a lot of the “hard work” on understanding what is going on in NAV, and in particular in the construction of a context-free grammar (which did not exist prior to this report). Below we list some of the interesting topics for future work (some of which have already been initiated), furthermore we categorize these into “immediate<sup>8</sup>” changes and “deep” changes:

- Dependency analysis (immediate). Extension of the “NAV Object Analyzer” tool to include more fine-grained dependencies. Dependencies at field-level (with read/write analysis) for table objects is useful in the process of decomposing tables. Relies on the C/AL parser.
- Clustering analysis/modularization of NAV (deep). Apply a tool like the above in modularizing a realistic part of NAV (for instance accounting). A proof-of-concept would be interesting for evaluating the possibility – and complexity – of a complete system modularization.
- Views (deep). Investigate what is needed in order to extend NAV with a *view* object type. In particular, it should be investigated how views with *update* can be handled (read-only views almost come for free). Updatable views are needed if views are to be used for backwards compatibility.
- Reimplementation of SIFT (immediate). Investigate possibilities of implementing SIFT using the augmented search tree algorithm described in this report. We label this change “immediate”, as it will not affect the users or developers of NAV directly (though they should be able to see a speedup!).
- Reimplementation of data retrieval in reports (immediate). Investigate possibilities of retrieving data from Microsoft SQL Server using SQL joins, rather than nested looping. The same comment about “immediate” above applies here.
- Extend C/AL with polymorphism/type inference or explicit subtyping (deep). We already described how the strict type annotations in C/AL prevent code to be used on similar table objects. An interesting

---

<sup>8</sup>By immediate we mean changes that will not require extensive adaption by NAV users and NAV partners.

project – which should be carried out before extending C/AL – is to classify/identify the various kinds of code duplication in C/AL. It is our thesis that most of the code duplication in NAV is due to lack of polymorphism – and not just because of bad programming style. This thesis should be investigated.

- Translation of C/AL to an object oriented programming language, for instance C# (deep). We have provided a high-level class interface for each NAV object type. With this description as basis, it would be interesting to investigate possibilities of translating the (old-fashioned) C/AL code to a more up-to-date programming language like for instance C#.
- Investigate possible concurrency- and locking issues in NAV (deep). As described in the question on page 35, NAV does not scale well beyond 250 simultaneous users. Supposedly this is due to locking of tables, so it would be interesting to investigate how locking is done (and when). According to [10], the maintenance of SIFT indexes plays a big role in database locking – hence this investigation may overlap with the SIFT reimplementations mentioned earlier.

## Acknowledgements

I am grateful to Fritz Henglein for fruitful discussions and proof reading of the report. I thank Peter Büniger, Thomas Jensen, Jesper Kiehn, and Hans Kierulff of Microsoft Dynamics, Vedbæk, Denmark for their inputs to the report, and for helping me understand the more *exotic* parts of Microsoft Dynamics NAV.

## References

- [1] David J. Barnes. *Object-Oriented Programming with Java: A First Programming Text*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
- [2] Arthur J. Bernstein and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [3] Jason Bloomberg and Ronald Schmelzer. *Service Orient or Be Doomed!: How Service Orientation Will Change Your Business*. John Wiley & Sons, Inc., New York, NY, USA, 2006.

- [4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill Science/Engineering/Math, July 2001.
- [5] Yvonne Dittrich and Sebastien Vaucouleur. Customization and Upgrade of ERP systems. An empirical Perspective. Technical report, ITU, Copenhagen, 2008.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [7] Google. DepAn: Dependency visualization and analysis.  
<http://code.google.com/p/google-depan/>.
- [8] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag, 1998.
- [9] Kathleen Jensen and Niklaus Wirth. *PASCAL user manual and report*. Springer-Verlag New York, Inc., New York, NY, USA, 1974.
- [10] Jesper Kiehn, Peter Bünger, Hans Kierulff, and Tom Hvitved. Personal conversation with Jesper Kiehn, Peter Bünger and Hans Kierulff, 2008-09-25, 2008.
- [11] Joel J. Lerner. *Bookkeeping and Accounting*. McGraw-Hill Education, 2004.
- [12] Microsoft. Microsoft Dynamics NAV product information.  
<http://www.microsoft.com/dynamics/nav/product/default.aspx>.
- [13] Microsoft. The Official Microsoft WPF and Windows Forms Site.  
<http://windowsclient.net/>.
- [14] Microsoft. *Course 8871 - Finance In Microsoft Dynamics NAV 5.0*, 2007.
- [15] Microsoft. *Microsoft Dynamics NAV 5.00 - Application Designer's Guide*, 2007.
- [16] Michael Nissen and Ken Friis Larsen. FunSETL - Functional Reporting for ERP Systems. In *Draft Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP), Nijmegen, The Netherlands*, Technical Report ICIS-R08007, Nijmegen, The Netherlands, 5 2008. Radboud University Nijmegen.

- [17] O. A. Shcherbina. Tree decomposition and discrete optimization problems: A survey. *Cybernetics and Sys. Anal.*, 43(4):549–562, 2007.
- [18] David Studebaker. *Programming Microsoft Dynamics NAV*. Packt Publishing, 2007.
- [19] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.
- [20] Jerry J. Weygandt, Donald E. Kieso, and Paul D. Kimmel. *Financial Accounting, with Annual Report*. Wiley, 2004.
- [21] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

## A Glossary

The table below summarizes a list of the most important terminology used in Microsoft Dynamics NAV. For each term we put a reference to the page where it is introduced, and a small description.

NAV terminology	Page	Description/CS terminology
Object	3	Class (OOP)
Table	4	Class (OOP – database layer)
Field	4	Column (relational databases)
Trigger	6	Method (OOP)
Record	5	Row (relational databases)
Key	7	Key or index
Primary key	7	Primary key (relational databases)
Secondary key	7	Index (relational databases) with aggregated data (Sum Index Field Technology)
Table relation	12	Generalized foreign key constraints (relational databases), but not invariants
FlowField	18	Virtual column in a table (typically containing aggregated value from some other table)
FlowFilter	20	Parameter used in the calculation of FlowFields (instantiated at runtime)
Codeunit	24	Class (OOP – business logic layer)
Procedure	24	Method (OOP)
C/AL	24	The domain specific language of NAV
Form	30	The GUI through which the end-user interacts with NAV. Forms are typically bound to a table, from which data is presented.
Report	31	Normally; derived data from (pairwise joined) tables. Sometimes; update of data in tables (batch run)
Data item	31	Table (relational databases)
C/SIDE	34	The NAV client

## B C/AL Parser

### B.1 Abstract Syntax Tree

```
1 #light "off"
  //////////////////////////////////////////////////
  // C/AL AST //
  //////////////////////////////////////////////////
6
  module CALAST

  type 'a indexnumber' =
    | IndexPos of int
    | IndexNeg of int
11 and 'a indexnumber = 'a indexnumber * 'a

  type 'a identifier' =
    | Id of string
    | IdentOnlyIndex of 'a indexnumber
    | IdentIndex of 'a identifier * 'a indexnumber
    | SysCall of string
16 and 'a identifier = 'a identifier * 'a

21 // Expressions
  and 'a exp' =
    | Int of int
    | Real of string
    | Date of string
    | Time of string
    | DateTime of string
    | String of string
    | Ident of 'a identifier
    | True
    | False
31 | Opr of string * 'a exp list
    | Call of 'a identifier * 'a exp list
  and 'a exp = 'a exp * 'a

36 // Statements
  type forop = string

  type 'a stmt' =
41 | StmtEmpty
    | StmtBlock of 'a stmts
    | StmtAssign of 'a exp * string * 'a exp
    | StmtExp of 'a exp
    | StmtIfThenElse of 'a exp * 'a stmt * 'a stmt
    | StmtIfThen of 'a exp * 'a stmt
46 | StmtCase of 'a exp * ('a exp list * 'a stmt) list * 'a caseelse
    | StmtWhile of 'a exp * 'a stmt
    | StmtRepeat of 'a stmts * 'a exp
    | StmtFor of 'a exp * 'a exp * forop * 'a exp * 'a stmt
51 | StmtWith of 'a exp * 'a stmt
    | StmtReturnNone
    | StmtReturn of 'a exp
  and 'a stmt = 'a stmt * 'a
  and 'a stmts = 'a stmt list
56 and 'a caseelse' =
    | CaseElseEmpty
    | CaseElse of 'a stmts
  and 'a caseelse = 'a caseelse * 'a

61 // Types
  type 'a typedef' =
    | TypeAction
    | TypeDate
    | TypeTime
    | TypeInteger
    | TypeBoolean
    | TypeDateTime
    | TypeDecimal
    | TypeRecordRef
    | TypeGUID
    | TypeDialog
    | TypeChar
    | TypeDateFormula
    | TypeOption
    | TypeVariant
76 | TypeInStream
```

```

| TypeOutputStream
| TypeFieldRef
81 | TypeKeyRef
| TypeFile
| TypeRecordID
| TypeBigInteger
| TypeDuration
86 | TypeCode of int
| TypeText of int
| TypeRecord of int
| TypeTemporaryRecord of int
| TypeCodeunit of int
91 | TypeForm of int
| TypeReport of int
| TypeXMLport of int
| TypeAutomationEvents of 'a identifier
| TypeAutomation of 'a identifier
96 | TypeOCX of 'a identifier
| TypeArray of (int list) * 'a typedef
| TypeEnum of string
| TypeTextConst of string
and 'a typedef = 'a typedef' * 'a
101
type 'a returntype =
| NoReturnType
| UnNamedReturn of 'a typedef
| NamedReturn of 'a identifier * 'a typedef
106
type 'a vardecl = 'a identifier * 'a typedef
type varopt = string
type 'a formaltyp = 'a typedef
type 'a fsection = varopt * 'a identifier * 'a formaltyp
111 type 'a procedurebody = 'a vardecl list * 'a stmts
type 'a procedureheading = string * 'a identifier * 'a fsection list * 'a returntype
type 'a proceduredecl = 'a procedureheading * 'a procedurebody

type 'a eventheader = 'a identifier * 'a identifier * 'a fsection list
116 type 'a eventdecl = 'a eventheader * 'a procedurebody
type 'a cucomment = string

type 'a code = 'a vardecl list * 'a proceduredecl list * 'a eventdecl list * 'a cucomment
121 // Permissions
type 'a perm = string * int * string

// Properties allowed on a codeunit
type 'a cuprop' =
126 | CUPTrig of string * 'a procedurebody
| CUPPerm of 'a perm list
| CUPCFRONTMayUsePermissions of bool
| CUPSingleInstance of bool
| CUPTableNo of int
131 and 'a cuprop = 'a cuprop' * 'a
type 'a codeunitproperties = 'a cuprop list

// Properties allowed on NAV objects (in general)
type 'a objprop' =
136 | OBPDate of string
| OBPModified of bool
| OBPTIME of string
| OBPVersionList of string
type 'a objprop = 'a objprop' * 'a
141 type 'a objproperties = 'a objprop list

type 'a codeunitbody = 'a objproperties * 'a codeunitproperties * 'a code
// NAV codeunit object
type 'a codeunit = string * string * 'a codeunitbody
146

// Properties allowed on a key
type 'a kprop' =
| KEPClustered of bool
151 | KEPKeyGroups of string
| KEPMaintainSIFTIndex of bool
| KEPMaintainSQLIndex of bool
| KEPSIFTLevelsToMaintain of string
| KEPSQLIndex of string
156 | KEPSumIndexFields of string
and 'a kprop = 'a kprop' * 'a
type 'a keyproperties = 'a kprop list

type 'a key = string * 'a keyproperties
161 type 'a keys = 'a key list

```

```

// Properties allowed on a field
type 'a fprop' =
166 | FIPTrig of string * 'a procedurebody
    | FIPAltSearchField of string
    | FIPAutoFormatExpr of 'a exp
    | FIPAutoFormatType of int
    | FIPAutoIncrement of bool
    | FIPBlankNumbers of string
171 | FIPBlankZero of bool
    | FIPCalcFormula of string
    | FIPCaptionClass of 'a exp
    | FIPCaptionML of string
    | FIPClosingDates of bool
176 | FIPCompressed of bool
    | FIPDateFormula of bool
    | FIPDecimalPlaces of string
    | FIPDescription of string
    | FIPEditable of bool
181 | FIPFieldClass of string
    | FIPInitValue of string
    | FIPMaxValue of string
    | FIPMinValue of string
    | FIPNotBlank of bool
186 | FIPNumeric of bool
    | FIPOptionCaptionML of string
    | FIPOptionString of string
    | FIPSubType of string
    | FIPTableIDExpr of 'a exp
191 | FIPTableRelation of string
    | FIPTestTableRelation of bool
    | FIPValidateTableRelation of bool
    | FIPValuesAllowed of string

196 and 'a fprop = 'a fprop' * 'a
type 'a fieldproperties = 'a fprop list

type 'a field = string * 'a fieldproperties
type 'a fields = 'a field list

201 // Properties allowed on a table
type 'a tprop' =
    | TAPTrig of string * 'a procedurebody
    | TAPPerm of 'a perm list
206 | TAPCaptionML of string
    | TAPDataCaptionFields of string
    | TAPDataPerCompany of bool
    | TAPDrillDownFormID of int
    | TAPLinkedObject of bool
211 | TAPLookupFormID of int
    | TAPPasteIsValid of bool
and 'a tprop = 'a tprop' * 'a
type 'a tableproperties = 'a tprop list

216 type 'a tablebody = 'a objproperties * 'a tableproperties * 'a fields * 'a keys * 'a code
// NAV table object
type 'a table = string * string * 'a tablebody

221 // Properties allowed on a menu item
type 'a mprop' =
    | MIPTrig of string * 'a procedurebody
    | MIPCaptionML of string
    | MIPEllipsis of bool
226 | MIPEnabled of bool
    | MIPID of int
    | MIPMenuItemType of string
    | MIPMenuLevel of int
    | MIPPushAction of 'a exp
231 | MIPRunFormLink of string
    | MIPRunFormLinkType of string
    | MIPRunFormOnRec of bool
    | MIPRunFormView of string
    | MIPRunObject of string
236 | MIPShortCutKey of string
    | MIPUpdateOnAction of bool
    | MIPVisible of bool
and 'a mprop = 'a mprop' * 'a
type 'a mproperties = 'a mprop list

241 type 'a menuitem = 'a mproperties
type 'a menuitems = 'a menuitem list

// Properties allowed on a control
246 type 'a cprop' =

```

```

| COPTrig of string * 'a procedurebody
| COPAssistEdit of bool
| COPAutoCalcField of bool
| COPAutoEnter of bool
251 | COPAutoFormatExpr of 'a exp
| COPAutoFormatType of int
| COPAutoRepeat of bool
| COPBackColor of int
| COPBackTransparent of bool
256 | COPBitmap of int
| COPBitmapList of string
| COPBitmapPos of string
| COPBlankNumbers of string
| COPBlankZero of bool
261 | COPBorder of bool
| COPBorderColor of int
| COPBorderStyle of string
| COPBorderWidth of string
| COPCancel of bool
266 | COPCaptionClass of 'a exp
| COPCaptionML of string
| COPClearOnLookup of bool
| COPClosingDates of bool
| COPDateFormula of bool
271 | COPDecimalPlaces of string
| COPDefault of bool
| COPDivisor of int
| COPDrillDown of bool
| COPDrillDownFormID of string
276 | COPDropDown of bool
| COPEditable of bool
| COPEllipsis of bool
| COPEnabled of bool
| COPFocusable of bool
281 | COPFocusOnClick of bool
| COPFontBold of bool
| COPFontName of string
| COPFontSize of int
| COPForeColor of int
286 | COPHeadingHeight of int
| COPHorzAlign of string
| COPHorzGlue of string
| COPInColumn of bool
| COPInColumnHeading of bool
291 | COPInFrame of bool
| COPInlineEditing of bool
| COPInMatrix of bool
| COPInMatrixHeading of bool
| COPInPage of int
296 | COPInvalidActionAppearance of string
| COPLeaderDots of bool
| COPLookup of bool
| COPLookupFormID of string
| COPMatrixColumnWidth of int
301 | COPMatrixSourceTable of int
| COPMaxLength of int
| COPMaxValue of string
| COPMenuItems of string * 'a menuitems
| COPMinValue of string
306 | COPMultiLine of bool
| COPName of string
| COPNextControl of int
| COPNotBlank of bool
| COPOptionCaptionML of string
311 | COPOptionString of string
| COPOptionValue of string
| COPPageNamesML of string
| COPParentControl of int
| COPPasswordText of bool
316 | COPPermanentAssist of bool
| COPPushAction of 'a exp
| COPRowHeight of int
| COPRunFormLink of string
| COPRunFormLinkType of string
321 | COPRunFormView of string
| COPRunObject of string
| COPShapeStyle of string
| COPShowCaption of bool
| COPSourceExpr of 'a exp
326 | COPSubFormID of int
| COPSubFormLink of string
| COPSubFormView of string
| COPTableRelation of string
| COPTestTableRelation of bool

```

```

331 | COPToolTipML of string
    | COPTopLineOnly of bool
    | COPUpdateOnAction of bool
    | COPValidateTableRelation of bool
    | COPValuesAllowed of string
336 | COPVertAlign of string
    | COPVertGlue of string
    | COPVisible of bool
    and 'a cprop = 'a cprop' * 'a
    type 'a controlproperties = 'a cprop list
341
    type 'a controlheader = int * string * string * int * int * int
    type 'a control = 'a controlheader * 'a controlproperties
    type 'a controls = 'a control list

346 // Properties allowed on a form
    type 'a foprop' =
    | FOPTrig of string * 'a procedurebody
    | FOPPerm of 'a perm list
    | FOPActiveControlOnOpen of int
351 | FOPAutoPosition of string
    | FOPAutoSplitKey of bool
    | FOPBorderStyle of string
    | FOPCalcFields of string
    | FOPCaptionBar of string
356 | FOPCaptionML of string
    | FOPDataCaptionExpr of 'a exp
    | FOPDataCaptionFields of string
    | FOPDelayedInsert of bool
    | FOPDeleteAllowed of bool
361 | FOPEditable of bool
    | FOPHeight of int
    | FOPInsertAllowed of bool
    | FOPLogHeight of int
    | FOPLogWidth of int
366 | FOPLookupMode of bool
    | FOPMaximizable of bool
    | FOPMinimizable of bool
    | FOPModifyAllowed of bool
    | FOPMultipleNewLines of bool
371 | FOPPopulateAllFields of bool
    | FOPSaveControlInfo of bool
    | FOPSavePosAndSize of bool
    | FOPSaveTableView of bool
    | FOPSaveValues of bool
376 | FOPSizeable of bool
    | FOPSourceTable of int
    | FOPSourceTablePlacement of string
    | FOPSourceTableTemporary of bool
    | FOPSourceTableView of string
381 | FOPTableBoxID of int
    | FOPTimerInterval of int
    | FOPUpdateOnActivate of bool
    | FOPWidth of int
    and 'a foprop = 'a foprop' * 'a
386 type 'a formproperties = 'a foprop list

    type 'a formbody = 'a objproperties * 'a formproperties * 'a controls * 'a code
    // NAV form object
    type 'a form = string * string * 'a formbody
391

    // Properties allowed on a request form
    type 'a rfprop' =
396 | RFPTrig of string * 'a procedurebody
    | RFPPerm of 'a perm list
    | RFPCaptionML of string
    | RFPHeight of int
    | RFPLookupMode of bool
401 | RFPWidth of int
    | RFPSaveValues of bool
    | RFPSourceTable of int
    and 'a rfprop = 'a rfprop' * 'a
406 type 'a requestformproperties = 'a rfprop list

    type 'a requestform = 'a requestformproperties * 'a controls

    // Properties allowed on a section control
    type 'a scprop' =
411 | SCPAutoCalcField of bool
    | SCPAutoFormatExpr of 'a exp
    | SCPAutoFormatType of int
    | SCPBackColor of int

```

```

| SCPBlankNumbers of string
416 | SCPBlankZero of bool
| SCPBorder of bool
| SCPBorderColor of int
| SCPBorderStyle of string
| SCPBorderWidth of string
421 | SCPCaptionClass of 'a exp
| SCPCaptionML of string
| SCPDecimalPlaces of string
| SCPDivisor of int
| SCPFontBold of bool
426 | SCPFontItalic of bool
| SCPFontName of string
| SCPFontUnderline of bool
| SCPFontSize of int
| SCPFormat of string
431 | SCPHorzAlign of string
| SCPHorzGlue of string
| SCPLeaderDots of bool
| SCPMultiLine of bool
| SCPName of string
436 | SCPOptionCaptionML of string
| SCPOptionString of string
| SCPPadChar of string
| SCPParentControl of int
| SCPShapeStyle of string
441 | SCPSourceExpr of 'a exp
| SCPVertAlign of string
| SCPVisible of bool
and 'a scprop = 'a scprop' * 'a
type 'a sectioncontrolproperties = 'a scprop list
446
type 'a sectioncontrolheader = int * string * string * int * int * int
type 'a sectioncontrol = 'a sectioncontrolheader * 'a sectioncontrolproperties
type 'a sectioncontrols = 'a sectioncontrol list

451 // Properties allowed on a section
type 'a seprop' =
| SEPTrig of string * 'a procedurebody
| SEPKeepWithNext of bool
| SEPPlaceInBottom of bool
456 | SEPPrintOnEveryPage of bool
| SEPSectionHeight of int
| SEPSectionType of string
| SEPSectionWidth of int
and 'a seprop = 'a seprop' * 'a
461 type 'a sectionproperties = 'a seprop list

type 'a section = 'a sectionproperties * 'a sectioncontrols
type 'a sections = 'a section list

466 // Properties allowed on a dataitem
type 'a diprop' =
| DIPTrig of string * 'a procedurebody
| DIPCalcFields of string
| DIPDataItemIndent of int
471 | DIPDataItemLink of string
| DIPDataItemLinkReference of string
| DIPDataItemTable of int
| DIPDataItemTableView of string
| DIPDataItemVarName of string
476 | DIPGroupTotalFields of string
| DIPMaxIteration of int
| DIPNewPagePerGroup of bool
| DIPNewPagePerRecord of bool
| DIPPrintOnlyIfDetail of bool
481 | DIPReqFilterFields of string
| DIPReqFilterHeadingML of string
| DIPTotalFields of string
and 'a diprop = 'a diprop' * 'a
type 'a dataitemproperties = 'a diprop list
486
type 'a dataitem = 'a dataitemproperties * 'a sections
type 'a dataitems = 'a dataitem list

// Properties allowed on a report
491 type 'a reprow' =
| REPTrig of string * 'a procedurebody
| REPPerm of 'a perm list
| REPCaptionML of string
| REPHorzGrid of int
496 | REPLeftMargin of int
| REPOrientation of string
| REPPaperSize of string

```

```

| REPProcessingOnly of bool
| REPShowPrintStatus of bool
501 | REPUseReqForm of bool
| REPUseSystemPrinter of bool
and 'a reprop = 'a reprop * 'a
type 'a reportproperties = 'a reprop list

506 type 'a reportbody = 'a objproperties * 'a reportproperties *
'a dataitems * 'a requestform * 'a code
// NAV report object
type 'a report = string * string * 'a reportbody

511 type 'a NAVObject =
| Codeunit of 'a codeunit
| Table of 'a table
| Form of 'a form
516 | Report of 'a report

let getPos (pos : Lexing.Position) =
" at line " + pos.pos_lnum.ToString() +
521 " col " + pos.pos_cnum.ToString()

(*let dummy_pos = { new Lexing.Position
with pos_fname = "<dummy>";
and pos_lnum = 0;
526 and pos_bol = 0;
and pos_cnum = 0;
}
*)

```

## B.2 System Calls

```

1 #light "off"

// C/AL system call keywords //
6 module SysCallKeywords

open Microsoft.FSharp.Collections

11 (* C/AL system calls (reserved keywords) *)
let Words = [
"ABS"; "ACTIVATE"; "ACTIVE"; "ADDLINK"; "ADDTXT";
"APPLICATIONPATH"; "ARRAYLEN"; "ASCENDING"; "BEEP"; "BREAK";
"CALCDATE"; "CALCFIELD"; "CALCFIELDS"; "CALCSUM"; "CALCSUMS";
16 "CAPTION"; "CHANGECOMPANY"; "CHECKLICENSEFILE"; "CLASS"; "CLEAR";
"CLEARALL"; "CLEARLASTERROR"; "CLEARMARKS"; "CLOSE"; "CLOSINGDATE";
"CODECOVERAGELOG"; "COMMANDLINE"; "COMMIT"; "COMPANYNAME"; "COMPRESSARRAY";
"CONFIRM"; "CONSISTENT"; "CONST"; "CONTEXTURL"; "CONVERTSTR"; "COPY";
"COPYARRAY"; "COPYFILTER"; "COPYFILTERS"; "COPYLINKS"; "COPYSTR";
21 "COPYSTREAM"; "COUNT"; "COUNTAPPROX"; "CREATE"; "CREATEDATETIME";
"CREATEGUID"; "CREATEINSTREAM"; "CREATEOUTSTREAM"; "CREATETEMPFILE";
"CREATETOTALS"; "CURRENTDATETIME"; "CURRENTKEY"; "CURRENTKEYINDEX";
"CURRENTTRANSACTIONTYPE"; "DATE2DMY"; "DATE2DWY"; "DATI2VARIANT";
"DECIMALPLACESMAX"; "DECIMALPLACESMIN"; "DELCHR";
26 "DELETE"; "DELETEALL"; "DELETEDLINK"; "DELETEDLINKS"; "DELSTR";
"DMY2DATE"; "DOWNLOAD"; "DOWNLOADFROMSTREAM"; "DT2DATE"; "DT2TIME";
"DUPLICATE"; "DWY2DATE"; "EDITABLE"; "ENABLED"; "ENVIRON";
"EOS"; "ERASE"; "ERROR"; "EVALUATE"; "EXISTS";
"EXPORT"; "FIELD"; "FIELDACTIVE"; "FIELDCAPTION"; "FIELDcount";
31 "FIELDERROR"; "FIELDEXIST"; "FIELDINDEX"; "FIELDNAME"; "FIELDNO";
"FILENAME"; "FILTERGROUP"; "FIND"; "FINDFIRST"; "FINDLAST";
"FINDSET"; "FORM"; "FORMAT"; "GET"; "GETFILTER";
"GETFILTERS"; "GETLASTERRORTEXT"; "GETPOSITION"; "GETRANGEMAX";
"GETRANGEMIN"; "GETRECORD"; "GETRESPONSE"; "GETSTAMP"; "GETSUBTEXT";
36 "GETTABLE"; "GETVIEW"; "GLOBALLANGUAGE"; "GUIALLOWED"; "HASFILTER";
"HASLINKS"; "HASVALUE"; "HEIGHT"; "HYPERLINK"; "IMPORT"; "INCSTR";
"INIT"; "INLINEEDITING"; "INPUT"; "INSERT"; "INSSTR";
"ISACTION"; "ISAUTOMATION"; "ISBINARY"; "ISBOOLEAN"; "ISCHAR";
"ISCLEAR"; "ISCODE"; "ISCODEUNIT"; "ISDATE"; "ISDATEFORMULA";
41 "ISDECIMAL"; "ISEMPTY"; "ISFILE"; "ISINSTREAM"; "ISINTEGER";
"ISNULLGUID"; "ISOPTION"; "ISOUTSTREAM"; "ISRECORD"; "ISSERVICETIER";
"ISTEXT"; "ISTIME"; "ISTRANSACTIONTYPE"; "KEYCOUNT"; "KEYGROUPDISABLE";
"KEYGROUPENABLE"; "KEYGROUPENABLED"; "KEYINDEX"; "LANGUAGE"; "LEN";
"LENGTH"; "LOCKTABLE"; "LOCKTIMEOUT"; "LOGHEIGHT"; "LOGWIDTH";
46 "LOOKUPMODE"; "LOWERCASE"; "MARK"; "MARKEDONLY"; "MAXIMIZEDONOPEN";
"MAXSTRLEN"; "MESSAGE"; "MINIMIZEDONOPEN"; "MODIFY"; "MODIFYALL";
"NAME"; "NEWPAGE"; "NEWPAGEPERRECORD"; "NEXT"; "NORMALDATE";
"NUMBER"; "OBJECTID"; "OPEN"; "OPTIONCAPTION"; "OPTIONSTRING";

```

```

"OSVERSION"; "PADSTR"; "PAGENO"; "PAPERSOURCE"; "POS";
51 "POWER"; "PREVIEW"; "PRINONLYIFDETAIL"; "QUERYREPLACE"; "QUIT";
"RANDOM"; "RANDOMIZE"; "READ"; "READCONSISTENCY"; "READPERMISSION";
"READTEXT"; "RECORD"; "RECORDID"; "RECORDLEVELLOCKING"; "RELATION";
"RENAME"; "RESET"; "ROUND"; "ROUNDDATETIME"; "RUN";
"RUNMODAL"; "SAVEASHTML"; "SAVEASXML"; "SAVERECORD"; "SEEK";
56 "SELECTLATESTVERSION"; "SELECTSTR"; "SERIALNUMBER"; "SETCURRENTKEY";
"SETDESTINATION"; "SETFILTER"; "SETPERMISSIONFILTER"; "SETPOSITION";
"SETRANGE"; "SETRECFILTER"; "SETRECORD"; "SETRESPONSE";
"SETSELECTIONFILTER"; "SETSOURCE"; "SETSTAMP";
"SETTABLE"; "SETTABLEVIEW"; "SETVIEW"; "SHELL"; "SHOWOUTPUT";
61 "SKIP"; "SLEEP"; "STRCHECKSUM"; "STRLEN"; "STRMENU";
"STRPOS"; "STRSUBSTNO"; "TABLECAPTION"; "TABLENAME"; "TABLENO";
"TEMPORARYPATH"; "TESTFIELD"; "TEXTMODE"; "TEXTPOS"; "TODAY";
"TOTALSCAUSEDBY"; "TRANSFERFIELDS"; "TRUNC"; "TYPE"; "UNDEFINED";
"UPDATE"; "UPDATECONTROLS"; "UPDATEEDITABLE"; "UPDATEFONTBOLD";
66 "UPDATEFORECOLOR"; "UPDATEINDEXT"; "UPDATESELECTED"; "UPLOAD";
"UPLOADINTOSTREAM"; "UPPERCASE"; "URL"; "USERREQUESTFORM"; "USERID";
"VALIDATE"; "VALUE"; "VARIABLEACTIVE"; "VARIANT2DATE"; "VARIANT2TIME";
"VISIBLE"; "WIDTH"; "WINDOWSLANGUAGE"; "WORKDATE"; "WRITE"; "WRITEMODE";
71 "WRITEPERMISSION"; "WRITETEXT"; "XPOS"; "YIELD"; "YPOS"]

let GetWordsAsHashSet () =
    let set = new HashSet<string>() in
    let _ = List.map (function s -> set.Add(s)) Words in
    set

```

### B.3 Lexer Definition

```

////////////////////////////////////
// Lexer for NAV objects (C/AL) //
////////////////////////////////////

5 {
    #light "off"
    open System
    open CALpars
    open Lexing
10 open Microsoft.FSharp.Core
    open Microsoft.FSharp.Collections

    let getLoc lexbuf = Lexing.lexeme_start_p lexbuf

15 let sysKeyWords = SysCallKeywords.GetWordsAsHashSet ()
    let isSysCall s = sysKeyWords.Contains s

    // Keywords
    let keyword = function
20 | "AND" -> T_AND
    | "ARRAY" -> T_ARRAY
    | "BEGIN" -> T_BEGIN
    | "BY" -> T_BY
    | "CASE" -> T_CASE
25 | "CODE" -> T_CODE
    | "CONTROLS" -> T_CONTROLS
    | "DATA" -> T_DATA
    | "DATAITEMS" -> T_DATAITEMS
    | "DIV" -> T_DIV
30 | "DO" -> T_DO
    | "DOWNTO" -> T_DOWNTO
    | "ELSE" -> T_ELSE
    | "END" -> T_END
    | "EXIT" -> T_EXIT
35 | "EVENT" -> T_EVENT
    | "FALSE" -> T_FALSE
    | "FIELDS" -> T_FIELDS
    | "FOR" -> T_FOR
    | "IF" -> T_IF
40 | "IN" -> T_IN
    | "KEYS" -> T_KEYS
    | "LOCAL" -> T_LOCAL
    | "MENUITEMS" -> T_MENUITEMS
    | "MOD" -> T_MOD
45 | "NOT" -> T_NOT
    | "OF" -> T_OF
    | "OR" -> T_OR
    | "Permissions" -> T_PERMISSIONS
    | "PROCEDURE" -> T_PROCEDURE
50 | "PROPERTIES" -> T_PROPERTIES
    | "REPEAT" -> T_REPEAT
    | "REQUESTFORM" -> T_REQUESTFORM
    | "SECTIONS" -> T_SECTIONS
    | "TEMPORARY" -> T_TEMPORARY

```

```

55 | "THEN" -> T_THEN
| "TO" -> T_TO
| "TRUE" -> T_TRUE
| "UNTIL" -> T_UNTIL
| "VAR" -> T_VAR
60 | "WHILE" -> T_WHILE
| "WITH" -> T_WITH
| "WITHEVENTS" -> T_WITHEVENTS
| "XOR" -> T_XOR
(* System calls / predefined functions or identifiers *)
65 | str -> if isSysCall str then T_SYSCALL(str) else T_ID(str)

let rec getLineColumn str (l,c) =
let len = String.length str in
if len=0 then (l,c) else
70 | if String.sub str 0 1 = "\n" then getLineColumn (String.sub str 1 (len-1)) (l+1,0) else
getLineColumn (String.sub str 1 (len-1)) (l,c+1)

let record_newlines (lexbuf : lexbuf) =
let endpos = lexeme_end_p lexbuf in
let startpos = getLoc lexbuf in
75 | let (l,c) = getLineColumn (lexeme lexbuf) (startpos.pos_lnum, startpos.pos_cnum) in
lexbuf.EndPos <- {pos_bol = endpos.pos_bol;
pos_fname = endpos.pos_fname;
pos_cnum = c;
pos_lnum = l}

80
let extractIdentifier (lexbuf : lexbuf) =
let str = lexeme lexbuf in
let id = str.IndexOf ' ' in
let str = String.sub str (id + 1) (String.length str - id - 1) in
85 | let _ = record_newlines lexbuf in
str
}

let digit = ['0'-'9']
let whitespace = [' ' '\t']
90 | let newline = '\n' | "\r\n"
// C/AL uses '[]' to encode identifiers containing e.g. '{', '}', ';', '[' and ']'
let identifier = [^ '{' '}' ';' '\n' '\r']* | '[' ([^ ']' | "]")* ']'
let block = (identifier | newline)*

95
rule token = parse
| "OBJECT Codeunit " digit+ ' ' identifier { record_newlines lexbuf; T_CODEUNITHEADER(lexeme
lexbuf) }
| "OBJECT Table " digit+ ' ' identifier { record_newlines lexbuf; T_TABLEHEADER(lexeme lexbuf)
; }
| "OBJECT Form " digit+ ' ' identifier { record_newlines lexbuf; T_FORMHEADER(lexeme lexbuf) }
100 | "OBJECT Report " digit+ ' ' identifier { record_newlines lexbuf; T_REPORTHEADER(lexeme
lexbuf) }
| "{ " digit+ whitespace* ';' whitespace* ';' identifier whitespace* ';' identifier
whitespace* { record_newlines lexbuf; T_FIELDHEADER(lexeme lexbuf) }
| "{ ("No")? whitespace* ';' identifier { record_newlines lexbuf; T_KEYHEADER(lexeme lexbuf)
} }
| "AltSearchField=" identifier { T_ALTSEARCHFIELD(extractIdentifier lexbuf) }
| "BitmapList=" identifier { T_BITMAPLIST(extractIdentifier lexbuf) }
105 | "BorderWidth=" identifier { T_BORDERWIDTH(extractIdentifier lexbuf) }
| "CalcFields=" identifier { T_CALCFIELDS(extractIdentifier lexbuf) }
| "CalcFormula=" block { T_CALCFORMULA(extractIdentifier lexbuf) }
| "CaptionML=" identifier { T_CAPTIONML(extractIdentifier lexbuf) }
| "DataCaptionFields=" identifier { T_DATACAPTIONFIELDS(extractIdentifier lexbuf) }
110 | "DataItemLink=" block { T_DATAITEMLINK(extractIdentifier lexbuf) }
| "DataItemLinkReference=" identifier { T_DATAITEMLINKREFERENCE(extractIdentifier lexbuf) }
| "DataItemTableView=" block { T_DATAITEMTABLEVIEW(extractIdentifier lexbuf) }
| "DataItemVarName=" identifier { T_DATAITEMVARNAME(extractIdentifier lexbuf) }
| "DecimalPlaces=" identifier { T_DECIMALPLACES(extractIdentifier lexbuf) }
115 | "Description=" identifier { T_DESCRIPTION(extractIdentifier lexbuf) }
| "DrillDownFormID=" identifier { T_DRILLDOWNFORMID(extractIdentifier lexbuf) }
| "FontName=" identifier { T_FONTNAME(extractIdentifier lexbuf) }
| "Format=" identifier { T_FORMAT(extractIdentifier lexbuf) }
| "GroupTotalFields=" identifier { T_GROUPTOTALFIELDS(extractIdentifier lexbuf) }
120 | "InitValue=" identifier { T_INITVALUE(extractIdentifier lexbuf) }
| "KeyGroups=" block { T_KEYGROUPS(extractIdentifier lexbuf) }
| "LookupFormID=" identifier { T_LOOKUPFORMID(extractIdentifier lexbuf) }
| "MaxValue=" identifier { T_MAXVALUE(extractIdentifier lexbuf) }
| "MinValue=" identifier { T_MINVALUE(extractIdentifier lexbuf) }
125 | "Name=" identifier { T_NAME(extractIdentifier lexbuf) }
| "OBJECT-PROPERTIES" { T_OBJECTPROPERTIES }
| "OptionCaptionML=" identifier { T_OPTIONCAPTIONML(extractIdentifier lexbuf) }
| "OptionString=" identifier { T_OPTIONSTRING(extractIdentifier lexbuf) }
| "OptionValue=" identifier { T_OPTIONVALUE(extractIdentifier lexbuf) }
130 | "PadChar=" identifier { T_PADCHAR(extractIdentifier lexbuf) }
| "PageNamesML=" identifier { T_PAGENAMEML(extractIdentifier lexbuf) }
| "PaperSize=" identifier { T_PAPERSIZE(extractIdentifier lexbuf) }

```

```

135 | "ReqFilterFields=" identifier { T_REQFILTERFIELDS(extractIdentifier lexbuf) }
| "ReqFilterHeadingML=" identifier { T_REQFILTERHEADINGML(extractIdentifier lexbuf) }
| "RunFormLink=" block { T_RUNFORMLINK(extractIdentifier lexbuf) }
| "RunFormView=" block { T_RUNFORMVIEW(extractIdentifier lexbuf) }
| "RunObject=" ("Codeunit" | "Form" | "Report") ' ' digit+ { T_RUNOBJECT(extractIdentifier
lexbuf) }
| "ShortCutKey=" identifier { T_SHORTCUTKEY(extractIdentifier lexbuf) }
| "SIFTLevelsToMaintain=" identifier { T_SIFTLEVELSTOMAINAIN(extractIdentifier lexbuf) }
140 | "SourceTableView=" block { T_SOURCETABLEVIEW(extractIdentifier lexbuf) }
| "SQLIndex=" block { T_SQLINDEX(extractIdentifier lexbuf) }
| "SubFormLink=" block { T_SUBFORMLINK(extractIdentifier lexbuf) }
| "SubFormView=" block { T_SUBFORMVIEW(extractIdentifier lexbuf) }
| "SubType=" block { T_SUBTYPE(extractIdentifier lexbuf) }
145 | "SumIndexFields=" block { T_SUMINDEXFIELDS(extractIdentifier lexbuf) }
| "TableRelation=" block { T_TABLERELATION(extractIdentifier lexbuf) }
| "ToolTipML=" identifier { T_TOOLTIPML(extractIdentifier lexbuf) }
| "TotalFields=" identifier { T_TOTALFIELDS(extractIdentifier lexbuf) }
| "ValuesAllowed=" identifier { T_VALUESALLOWED(extractIdentifier lexbuf) }
150 | "Version List=" identifier { T_VERSIONLIST(extractIdentifier lexbuf) }
| whitespace { token lexbuf } (* white-space *)
| newline { record_newlines lexbuf; token lexbuf }
| digit+ { T_INT(Int32.Parse(lexeme lexbuf)) }
| digit+ ('.' digit+)? ([ 'e' 'E' ] '-?' digit+)? { T_REAL(lexeme lexbuf) }
155 | ([ 'a'-'z' ] | [ 'A'-'Z' ] | '_' ) ([ 'a'-'z' ] | [ 'A'-'Z' ] | '_' | digit)* { keyword (lexeme
lexbuf) } (* keyword or identifier *)
| '\'' ([^ '\r' '\n' ] | "\\")* '\'' { T_STRING(lexeme lexbuf) }
| '"' ([^ '\r' '\n' ] | "\\")* '"' { T_ID(lexeme lexbuf) }
| digit digit '-' digit digit '-' digit digit { T_DATE(lexeme lexbuf) }
| digit+ 'D' { T_DATE(lexeme lexbuf) }
160 | digit digit ':' digit digit ':' digit digit { T_TIME(lexeme lexbuf) }
| (digit+ | digit+ '.' digit+)'T' { T_TIME(lexeme lexbuf) }
| (digit+ | digit+ '.' digit+)'DT' { T_DATETIME(lexeme lexbuf) }
| "/" { SkipComment lexbuf }
| '+' { T_PLUS }
165 | '-' { T_MINUS }
| '*' { T_MULT }
| '/' { T_DIV }
| '=' { T_EQUAL }
| '<' { T_LT }
170 | '>' { T_GT }
| '<=' { T_LE }
| '>=' { T_GE }
| '<>' { T_NOTEQUAL }
| '.' { T_DOT }
175 | '..' { T_DOTDOT }
| ',' { T_COMMA }
| ';' { T_SEMICOLON }
| '(' { T_LPAR }
| ')' { T_RPAR }
180 | '[' { T_LBRACK }
| ']' { T_RBRACK }
| '{' { T_LCURLY }
| '}' { T_RCURLY }
| ':' { T_COLON }
185 | "::" { T_SCOPE }
| '@' { T_AT }
| ":" { T_ASSIGN }
| "+" { T_PLUSEQ }
| "-" { T_MINUSEQ }
190 | "*" { T_MULTEQ }
| "/" { T_DIVEQ }
| eof { T_EOF }
| _ { failwith("Lexical analysis failed")}

195 and SkipComment = parse
| newline { record_newlines lexbuf; token lexbuf }
| eof { T_EOF }
| _ {SkipComment lexbuf}

```

## B.4 Parser Definition

```

////////////////////////////////////
2 // Parser for NAV objects (C/AL) //
////////////////////////////////////

%{

7 #light "off"
  open CALAST
  open PropertiesParser

12 %}

```

```

%start NAVObjectList

// Headers
%token <string> T_CODEUNITHEADER
17 %token <string> T_TABLEHEADER
%token <string> T_FORMHEADER
%token <string> T_FIELDHEADER
%token <string> T_KEYHEADER
22 %token <string> T_REPORTHEADER

// Constants
%token <string> T_ID
%token <string> T_REAL
%token <int> T_INT
27 %token <string> T_STRING
%token <string> T_DATE
%token <string> T_TIME
%token <string> T_DATETIME
%token <string> T_SYSCALL
32

// General object properties
%token <string> T_CAPTIONML
%token <string> T_DATACAPTIONFIELDS
%token <string> T_DECIMALPLACES
37 %token <string> T_DRILLDOWNFORMID
%token <string> T_FONTNAME
%token <string> T_LOOKUPFORMID
%token <string> T_MAXVALUE
%token <string> T_MINVALUE
42 %token <string> T_OPTIONCAPTIONML
%token <string> T_OPTIONSTRING
%token <string> T_RUNFORMLINK
%token <string> T_RUNFORMVIEW
%token <string> T_RUNOBJECT
47 %token <string> T_TABLERELATION
%token <string> T_VALUESALLOWED
%token <string> T_VERSIONLIST

// Field properties
52 %token <string> T_ALTSEARCHFIELD
%token <string> T_CALCFORMULA
%token <string> T_DESCRIPTION
%token <string> T_INITVALUE
%token <string> T_SUBTYPE
57

// Key properties
%token <string> T_KEYGROUPS
%token <string> T_SIFTLEVELSTOMAINAIN
%token <string> T_SQLINDEX
62 %token <string> T_SUMINDEXFIELDS

// Form properties
%token <string> T_CALCFIELDS
%token <string> T_SOURCETABLEVIEW
67

// Control properties
%token <string> T_BITMAPLIST
%token <string> T_BORDERWIDTH
%token <string> T_NAME
72 %token <string> T_OPTIONVALUE
%token <string> T_PAGENAMEML
%token <string> T_SUBFORMLINK
%token <string> T_SUBFORMVIEW
%token <string> T_TOOLTIPL
77

// Menuitem properties
%token <string> T_SHORTCUTKEY

// Report properties
82 %token <string> T_PAPERSIZE

// DataItem properties
%token <string> T_DATAITEMLINK
%token <string> T_DATAITEMLINKREFERENCE
87 %token <string> T_DATAITEMTABLEVIEW
%token <string> T_DATAITEMVARNAME
%token <string> T_GROUPTOTALFIELDS
%token <string> T_REQFILTERFIELDS
%token <string> T_REQFILTERHEADINGML
92 %token <string> T_TOTALFIELDS

// Section control proeprties
%token <string> T_FORMAT
%token <string> T_PADCHAR

```

```

97 // Special tokens
   %token T_EOF

   // Keyword tokens
102 %token T_PERMISSIONS T_BEGIN T_END T_VAR
   %token T_PROCEDURE T_LOCAL T_EVENT T_IF T_THEN T_ELSE
   %token T_CASE T_OF T_WHILE T_DO T_REPEAT T_UNTIL
   %token T_FOR T_WITH T_EXIT T_TO T_DOWNT0
   %token T_TRUE T_FALSE T_NOT
107 %token T_MOD T_IN T_TEMPORARY T_WITHEVENTS T_ARRAY
   %token T_BY T_DATA T_DATAITEMS T_SECTIONS T_REQUESTFORM

   // Operator tokens
   %token T_ASSIGN T_NOTEQUAL T_LE T_GE T_DOT T_DOTDOT
112 %token T_PLUS T_MINUS T_MULT T_DIV T_AND T_COMMA
   %token T_SEMICOLON T_LPAR T_RPAR T_LBRACK T_RBRACK
   %token T_LCURLY T_RCURLY T_XOR T_EQUAL T_LT T_GT
   %token T_COLON T_SCOPE T_OR
117 %token T_AT T_PLUSEQ T_MINUSEQ T_MULTEQ T_DIVEQ

   // Object tokens
   %token T_OBJECTPROPERTIES T_PROPERTIES T_CODE T_KEYS
   %token T_FIELDS T_CONTROLS T_MENUITEMS

122 %type <Lexing.Position CALAST.NAVObject list> NAVObjectList
   %type <Lexing.Position CALAST.NAVObject> NAVObject
   %type <Lexing.Position CALAST.stmts> Statements
   %type <Lexing.Position CALAST.stmt> Statement
   %type <Lexing.Position CALAST.exp> Exp
127 %type <Lexing.Position CALAST.typedef> Type

   // Precedence and associativity
   %left T_DOTDOT
   %right T_SEMICOLON
132 %right T_COLON
   %left T_DO
   %left T_AND
   %left T_OR
   %left T_XOR
137 %right T_NOT
   %left T_LT
   %left T_GT
   %left T_EQUAL
   %left T_NOTEQUAL
142 %left T_LE
   %left T_GE
   %left T_IN
   %left T_SCOPE
   %left T_DOT
147 %left T_PLUS T_MINUS
   %left T_MULT
   %left T_DIV T_MOD
   %right T_IF
   %right T_THEN
152 %right T_ELSE
   %left T_LBRACK T_RBRACK

   %%

157 // Input should consist of a list of NAV objects
   NAVObjectList:
   { [] }
   | NAVObject NAVObjectList {$1 :: $2}

162 // The different kinds of NAV objects (supported for now)
   NAVObject:
   CodeUnit { Codeunit $1 }
   | Table { Table $1 }
   | Form { Form $1 }
167 | Report { Report $1 }

   // -- NAV Codeunit --
   CodeUnit:
172 T_CODEUNITHEADER T_LCURLY CodeUnitBody T_RCURLY { let header = ParseRegex "OBJECT Codeunit
   ([0-9]+) (.*)" $1 in
   (List.nth header 0, List.nth header 1, $3)
   }

   CodeUnitBody:
   ObjectProperties CodeUnitProperties Code { ($1,$2,$3) }
177 ObjectProperties:

```

```

    T_OBJECTPROPERTIES T_LCURLY ObjPropList T_RCURLY { $3 }

ObjPropList:
182   { [] }
    | ObjProp T_SEMICOLON ObjPropList { $1 :: $3 }

ObjProp:
    T_ID T_EQUAL Exp { (ExtractObjectProperty($1,$3), Parsing.symbol_end_pos()) }
187   | T_VERSIONLIST { (OBPVersionList $1, Parsing.symbol_end_pos()) }

CodeUnitProperties:
    T_PROPERTIES T_LCURLY CodeunitPropList T_RCURLY { $3 }

192 CodeunitPropList:
    { [] }
    | CodeunitProp T_SEMICOLON CodeunitPropList { $1 :: $3 }

CodeunitProp:
197   T_ID T_EQUAL Exp { (ExtractCodeunitProperty($1,$3), Parsing.symbol_end_pos()) }
    | Trigger { (CUPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
    | Permissions { (CUPPerm($1), Parsing.symbol_end_pos()) }

202 Trigger:
    T_ID T_EQUAL ProcedureBody { ($1,$3) }

Permissions:
    T_PERMISSIONS T_EQUAL PermListOpt { $3 }

207 PermListOpt:
    { [] }
    | PermList { $1 }

PermList:
212   Perm { [$1] }
    | Perm T_COMMA PermList { $1 :: $3 }

Perm:
217   T_ID T_INT T_EQUAL T_ID { ($1,$2,$4) }

Code:
    T_CODE T_LCURLY VarDeclListOpt ProcedureDeclList EventDeclList T_BEGIN Documentation T_END
    T_DOT T_RCURLY { ($3,$4,$5,$7) }

222 VarDeclListOpt:
    { [] }
    | T_VAR VarDeclList { $2 }

VarDeclList:
    VarDecl { [$1] }
227   | VarDecl VarDeclList { $1 :: $2 }

VarDecl:
    Identifier T_COLON Type T_SEMICOLON { ($1,$3) }

232 ProcedureDeclList:
    { [] }
    | ProcedureDecl ProcedureDeclList { $1 :: $2 }

ProcedureDecl:
237   ProcedureHeading T_SEMICOLON ProcedureBody T_SEMICOLON { ($1,$3) }

ProcedureHeading:
    LocalOpt T_PROCEDURE Identifier T_LPAR FpSectionListOpt T_RPAR ReturnTypeOpt { ($1,$3,$5,$7)
    }

242 LocalOpt:
    { "" }
    | T_LOCAL { "LOCAL" }

FpSectionListOpt:
247   { [] }
    | FpSectionList { $1 }

FpSectionList:
    FpSection { [$1] }
252   | FpSection T_SEMICOLON FpSectionList { $1 :: $3 }

FpSection:
    VarOpt Identifier T_COLON Type { ($1,$2,$4) }

257 VarOpt:
    { "" }
    | T_VAR { "VAR" }

```

```

ReturnTypOpt:
262   { NoReturnType }
   | T_COLON Type { UnNamedReturn($2) }
   | Identifier T_COLON Type { NamedReturn($1,$3) }

ProcedureBody:
267   VarDeclListOpt Block { ($1,$2) }

EventDeclList:
   { [] }
272 | EventDeclaration EventDeclList { $1 :: $2 }

EventDeclaration:
   EventHeader ProcedureBody T_SEMICOLON { ($1,$2) }

EventHeader:
277   T_EVENT Identifier T_SCOPE Identifier T_LPAR FpSectionListOpt T_RPAR T_SEMICOLON { ($2,$4,$6
   ) }

Documentation:
   { "" }

282 // -- Statement Parser --
Block:
   T_BEGIN Statements T_END { $2 }

287 // Sequence of statements (only allowed within BEGIN-END, REPEAT-UNTIL and ELSE-branch of
   CASE statements)
Statements:
   Statement { [$1] }
   | Statement T_SEMICOLON Statements { $1 :: $3 }

292 Statement:
   { (StmtEmpty, Parsing.symbol_end_pos()) }
   | Block { (StmtBlock($1), Parsing.symbol_end_pos()) }
   | Exp AssignOp Exp { (StmtAssign($1,$2,$3), Parsing.symbol_end_pos()) }
   | Exp { (StmtExp($1), Parsing.symbol_end_pos()) }
297 | T_IF Exp T_THEN Statement T_ELSE Statement { (StmtIfThenElse($2,$4,$6),
   Parsing.symbol_end_pos()) }
   | T_IF Exp T_THEN Statement { (StmtIfThen($2,$4),
   Parsing.symbol_end_pos()) }
   | T_CASE Exp T_OF CaseList CaseElse T_END { (StmtCase($2,$4,$5),
   Parsing.symbol_end_pos()) }
302 | T_WHILE Exp T_DO Statement { (StmtWhile($2,$4),
   Parsing.symbol_end_pos()) }
   | T_REPEAT Statements T_UNTIL Exp { (StmtRepeat($2,$4),
   Parsing.symbol_end_pos()) }
307 | T_FOR Exp T_ASSIGN Exp ForOp Exp T_DO Statement { (StmtFor($2,$4,$5,$6,$8),
   Parsing.symbol_end_pos()) }
   | T_WITH Exp T_DO Statement { (StmtWith($2,$4),
   Parsing.symbol_end_pos()) }
   | T_EXIT { (StmtReturnNone, Parsing.symbol_end_pos()) }
312 | T_EXIT T_LPAR Exp T_RPAR { (StmtReturn($3),
   Parsing.symbol_end_pos()) }

ForOp:
   T_TO { "TO" }
317 | T_DOWNTO { "DOWNTO" }

Case:
   ExpList T_COLON Statement { ($1,$3) }

322 CaseList:
   Case { [$1] }
   | Case T_SEMICOLON { [$1] }
   | Case T_SEMICOLON CaseList { $1 :: $3 }

327 CaseElse:
   { (CaseElseEmpty, Parsing.symbol_end_pos()) }
   | T_ELSE Statements { (CaseElse($2), Parsing.symbol_end_pos()) }

332 AssignOp:
   T_ASSIGN { "!=" }
   | T_PLUSEQ { "+=" }
   | T_MINUSEQ { "-=" }
   | T_MULTEQ { "*=" }
337 | T_DIVEQ { "/=" }

// -- Expression Parser --
Exp:
342   AtomExp { $1 }

```

```

| UnExp { $1 }
| BinExp { $1 }
| NaryExp { $1 }

347 AtomExp:
    Identifier { (Ident($1), Parsing.symbol_end_pos()) }
| T_STRING { (String($1), Parsing.symbol_end_pos()) }
| T_REAL { (Real($1), Parsing.symbol_end_pos()) }
| T_INT { (Int($1), Parsing.symbol_end_pos()) }
352 | T_DATE { (Date($1), Parsing.symbol_end_pos()) }
| T_TIME { (Time($1), Parsing.symbol_end_pos()) }
| T_DATETIME { (DateTime($1), Parsing.symbol_end_pos()) }
| T_TRUE { (True, Parsing.symbol_end_pos()) }
| T_FALSE { (False, Parsing.symbol_end_pos()) }

357 UnExp:
    T_LPAR Exp T_RPAR { $2 }
| T_NOT Exp { (Opr("NOT",[$2]), Parsing.symbol_end_pos()) }
| T_PLUS Exp { (Opr("+",[$2]), Parsing.symbol_end_pos()) }
362 | T_MINUS Exp { (Opr("-",[$2]), Parsing.symbol_end_pos()) }

BinExp:
    Exp T_PLUS Exp { (Opr("+", [$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_MINUS Exp { (Opr("-", [$1; $3]), Parsing.symbol_end_pos()) }
367 | Exp T_MULT Exp { (Opr("*", [$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_DIV Exp { (Opr("/", [$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_MOD Exp { (Opr("MOD", [$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_AND Exp { (Opr("AND", [$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_OR Exp { (Opr("OR",[$1; $3]), Parsing.symbol_end_pos()) }
372 | Exp T_XOR Exp { (Opr("XOR",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_LT Exp { (Opr("<",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_GT Exp { (Opr(">",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_EQUAL Exp { (Opr("=",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_LE Exp { (Opr("<=",[$1; $3]), Parsing.symbol_end_pos()) }
377 | Exp T_GE Exp { (Opr(">=",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_NOTEQUAL Exp { (Opr("<>",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_SCOPE Exp { (Opr("::",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_DOT Exp { (Opr(".",[$1; $3]), Parsing.symbol_end_pos()) }
| Exp T_DOTDOT Exp { (Opr("..", [$1; $3]), Parsing.symbol_end_pos()) }

382 NaryExp:
    Identifier T_LPAR ExpListOpt T_RPAR { (Call($1,$3),
                                           Parsing.symbol_end_pos()) }
| Exp T_LBRACK ExpList T_RBRACK { (Opr("arrref",$1 :: $3),
                                   Parsing.symbol_end_pos()) }
387 | Exp T_IN T_LBRACK ExpList T_RBRACK { (Opr("IN",$1 :: $4),
                                           Parsing.symbol_end_pos()) }

ExpListOpt:
392 { [] }
| ExpList { $1 }

ExpList:
397 | Exp { [$1] }
| Exp T_COMMA ExpList { $1 :: $3 }

// -- Identifier Parser --
Identifier:
402 T_ID { (Id($1), Parsing.symbol_end_pos()) }
| T_AT IdIndexNumber { (IdentOnlyIndex($2), Parsing.symbol_end_pos()) }
| Identifier T_AT IdIndexNumber { (IdentIndex($1,$3),
                                   Parsing.symbol_end_pos()) }
407 | T_SYSCALL { (SysCall($1), Parsing.symbol_end_pos()) }

IdentifierList:
{ [] }
| Identifier T_COMMA IdentifierList { $1 :: $3 }

412 IdIndexNumber:
    T_INT { (IndexPos ($1), Parsing.symbol_end_pos()) }
| T_MINUS T_INT { (IndexNeg ($2), Parsing.symbol_end_pos()) }

417 // -- Type Parser --
Type:
    SimpleType { $1 }
| SizedType { $1 }
| NumberedType { $1 }
422 | ParamType { $1 }
| ConstructedType { $1 }
| TextConstType { $1 }

SimpleType:

```

```

427 T_ID { ((match $1 with
      | "Action" -> TypeAction
      | "Date" -> TypeDate
      | "Time" -> TypeTime
      | "Integer" -> TypeInteger
432 | "Boolean" -> TypeBoolean
      | "DateTime" -> TypeDateTime
      | "Decimal" -> TypeDecimal
      | "RecordRef" -> TypeRecordRef
      | "GUID" -> TypeGUID
437 | "Dialog" -> TypeDialog
      | "Char" -> TypeChar
      | "DateFormula" -> TypeDateFormula
      | "Option" -> TypeOption
      | "Variant" -> TypeVariant
442 | "InStream" -> TypeInStream
      | "OutStream" -> TypeOutStream
      | "FieldRef" -> TypeFieldRef
      | "KeyRef" -> TypeKeyRef
      | "File" -> TypeFile
447 | "RecordID" -> TypeRecordID
      | "BigInteger" -> TypeBigInteger
      | "Duration" -> TypeDuration
      | _ -> failwith "Parser: SimpleType - This should not happen!"),
      Parsing.symbol_end_pos() }
452
SizedType:
  T_ID T_LBRACK T_INT T_RBRACK { ((match $1 with
      | "Code" -> TypeCode $3
      | "Text" -> TypeText $3
457 | _ -> failwith "Parser: SizedType - This should not happen
      !"),
      Parsing.symbol_end_pos() ) }

NumberedType:
  T_ID T_INT { ((match $1 with
462 | "Record" -> TypeRecord $2
      | "Codeunit" -> TypeCodeunit $2
      | "Form" -> TypeForm $2
      | "Report" -> TypeReport $2
      | "XMLport" -> TypeXMLport $2
467 | _ -> failwith "Parser: NumberedType - This should not happen!"),
      Parsing.symbol_end_pos() ) }
  | T_TEMPORARY T_ID T_INT { ((match $2 with
      | "Record" -> TypeTemporaryRecord $3
      | _ -> failwith "Parser: NumberedType2 - This should not happen!"
      ),
472 Parsing.symbol_end_pos() ) }

ParamType:
  T_ID Identifier T_WITHEVENTS { ((match $1 with
      | "Automation" -> TypeAutomationEvents $2
477 | _ -> failwith "Parser: ParamType - This should not happen
      !"),
      Parsing.symbol_end_pos() ) }
  | T_ID Identifier { ((match $1 with
      | "Automation" -> TypeAutomation $2
      | "OCX" -> TypeOCX $2
482 | _ -> failwith "Parser: ParamType2 - This should not happen!"),
      Parsing.symbol_end_pos() ) }

ConstructedType:
487 T_ARRAY T_LBRACK ArraySize T_RBRACK T_OF Type { (TypeArray ($3,$6),
      Parsing.symbol_end_pos() ) }
  | T_STRING { (TypeEnum $1, Parsing.symbol_end_pos() ) }

TextConstType:
492 T_ID T_STRING { ((match $1 with
      | "TextConst" -> TypeTextConst $2
      | _ -> failwith "Parser: TextConstType - This should not happen!"),
      Parsing.symbol_end_pos() ) }

497 ArraySize:
  T_INT { [$1] }
  | T_INT T_COMMA ArraySize { $1 :: $3 }

502
// -- NAV Table --
Table:
  T_TABLEHEADER T_LCURLY TableBody T_RCURLY { let header = ParseRegex "OBJECT Table ([0-9]+)
      (.*)" $1 in
      (List.nth header 0, List.nth header 1, $3) }

```

```

507 TableBody:
    ObjectProperties TableProperties Fields Keys Code { ($1,$2,$3,$4,$5) }

TableProperties:
512 T_PROPERTIES T_LCURLY TablePropList T_RCURLY { $3 }

TablePropList:
    { [] }
    | TableProp T_SEMICOLON TablePropList { $1 :: $3 }
517 TableProp:
    TableProperty { ($1, Parsing.symbol_end_pos()) }
    | Trigger { (TAPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
    | Permissions { (TAPPerm($1), Parsing.symbol_end_pos()) }
522 TableProperty:
    T_ID T_EQUAL Exp { ExtractTableProperty($1,$3) }
    | T_CAPTIONML { ParseTAPCaptionML $1 }
    | T_DATACAPTIONFIELDS { ParseTAPDataCaptionFields $1 }
527 | T_DRILLDOWNFORMID { ParseTAPDrillDownFormID $1 }
    | T_LOOKUPFORMID { ParseTAPLookupFormID $1 }

Fields:
532 T_FIELDS T_LCURLY FieldsList T_RCURLY { $3 }

FieldsList:
    Field { [$1] }
    | Field FieldsList { $1 :: $2 }
537 Field:
    T_FIELDHEADER T_RCURLY { ($1,[]) }
    | T_FIELDHEADER T_SEMICOLON FieldPropList T_RCURLY { ($1,$3) }

FieldPropList:
542 FieldProp { [$1] }
    | FieldProp T_SEMICOLON { [$1] }
    | FieldProp T_SEMICOLON FieldPropList { $1 :: $3 }

FieldProp:
547 FieldProperty { ($1, Parsing.symbol_end_pos()) }
    | Trigger { (FIPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }

FieldProperty:
552 T_ID T_EQUAL Exp { ExtractFieldProperty($1,$3) }
    | T_ALTSEARCHFIELD { ParseFIPAltSearchField $1 }
    | T_CALCFORMULA { ParseFIPCalcFormula $1 }
    | T_CAPTIONML { ParseFIPCaptionML $1 }
    | T_DECIMALPLACES { ParseFIPDecimalPlaces $1 }
    | T_DESCRIPTION { ParseFIPDescription $1 }
557 | T_INITVALUE { ParseFIPInitValue $1 }
    | T_MAXVALUE { ParseFIPMaxValue $1 }
    | T_MINVALUE { ParseFIPMinValue $1 }
    | T_OPTIONCAPTIONML { ParseFIPOptionCaptionML $1 }
    | T_OPTIONSTRING { ParseFIPOptionString $1 }
562 | T_SUBTYPE { ParseFIPSubType $1 }
    | T_TABLERELATION { ParseFIPTableRelation $1 }
    | T_VALUESALLOWED { ParseFIPValuesAllowed $1 }

Keys:
567 T_KEYS T_LCURLY KeysList T_RCURLY { $3 }

KeysList:
    { [] }
    | Key KeysList { $1 :: $2 }
572 Key:
    T_KEYHEADER T_RCURLY { ($1,[]) }
    | T_KEYHEADER T_SEMICOLON KeyPropList T_RCURLY { ($1,$3) }
577 KeyPropList:
    KeyProperty { [($1, Parsing.symbol_end_pos())] }
    | KeyProperty T_SEMICOLON KeyPropList { ($1, Parsing.symbol_end_pos()) :: $3 }

KeyProperty:
582 T_ID T_EQUAL Exp { ExtractKeyProperty($1,$3) }
    | T_KEYGROUPS { ParseKEPKeyGroups $1 }
    | T_SIFTLLEVELSTOMAINAIN { ParseKEPSIFTLevelsToMaintain $1 }
    | T_SQLINDEX { ParseKEPSQLIndex $1 }
    | T_SUMINDEXFIELDS { ParseKEPSumIndexFields $1 }
587

// -- NAV Form --
Form:

```

```

T_FORMHEADER T_LCURLY FormBody T_RCURLY { let header = ParseRegex "OBJECT Form ([0-9]+)
592      (.*)" $1 in
      (List.nth header 0, List.nth header 1, $3) }

FormBody:
  ObjectProperties FormProperties Controls Code { ($1,$2,$3,$4) }

597 FormProperties:
  T_PROPERTIES T_LCURLY FormPropList T_RCURLY { $3 }

FormPropList:
  { [] }
602 | FormProp T_SEMICOLON FormPropList { $1 :: $3 }

FormProp:
  FormProperty { ($1, Parsing.symbol_end_pos()) }
  | Trigger { (FOPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
607 | Permissions { (FOPPerm($1), Parsing.symbol_end_pos()) }

FormProperty:
  T_ID T_EQUAL Exp { ExtractFormProperty($1,$3) }
  | T_CALCFIELDS { ParseFOPCalcFields $1 }
612 | T_CAPTIONML { ParseFOPCaptionML $1 }
  | T_DATACAPTIONFIELDS { ParseFOPDataCaptionFields $1 }
  | T_SOURCETABLEVIEW { ParseFOPSourceTableView $1 }

Controls:
617 T_CONTROLS T_LCURLY ControlsList T_RCURLY { $3 }

ControlsList:
  { [] }
  | Control ControlsList { $1 :: $2 }
622

Control:
  ControlHeader T_RCURLY { ($1,[]) }
  | ControlHeader T_SEMICOLON ControlPropList T_RCURLY { ($1,$3) }

627 ControlHeader:
  T_LCURLY T_INT T_SEMICOLON T_ID T_SEMICOLON XPos T_SEMICOLON T_INT T_SEMICOLON T_INT
  T_SEMICOLON T_INT { ($2,$4,$6,$8,$10,$12) }

XPos:
  T_INT { $1.ToString() }
632 | T_ID { $1 } // "infinite"

ControlPropList:
  ControlProp { [$1] }
  | ControlProp T_SEMICOLON { [$1] }
637 | ControlProp T_SEMICOLON ControlPropList { $1 :: $3 }

ControlProp:
  ControlProperty { ($1, Parsing.symbol_end_pos()) }
  | Trigger { (COPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
642 | MenuItems { ($1, Parsing.symbol_end_pos()) }

ControlProperty:
  T_ID T_EQUAL Exp { ExtractControlProperty($1,$3) }
  | T_BITMAPLIST { ParseCOPBitmapList $1 }
647 | T_BORDERWIDTH { ParseCOPBorderWidth $1 }
  | T_CAPTIONML { ParseCOPCaptionML $1 }
  | T_DECIMALPLACES { ParseCOPDecimalPlaces $1 }
  | T_DRILLDOWNFORMID { ParseCOPDrillDownFormID $1 }
  | T_FONTNAME { ParseCOPFontName $1 }
652 | T_LOOKUPFORMID { ParseCOPLookupFormID $1 }
  | T_MAXVALUE { ParseCOPMaxValue $1 }
  | T_MINVALUE { ParseCOPMinValue $1 }
  | T_NAME { ParseCOPName $1 }
  | T_OPTIONCAPTIONML { ParseCOPOptionCaptionML $1 }
657 | T_OPTIONSTRING { ParseCOPOptionString $1 }
  | T_OPTIONVALUE { ParseCOPOptionValue $1 }
  | T_PAGENAMESML { ParseCOPPageNamesML $1 }
  | T_RUNFORMLINK { ParseCOPRunFormLink $1 }
  | T_RUNFORMVIEW { ParseCOPRunFormView $1 }
662 | T_TOOLTIPML { ParseCOPToolTipML $1 }
  | T_RUNOBJECT { ParseCOPRunObject $1 }
  | T_SUBFORMLINK { ParseCOPSubFormLink $1 }
  | T_SUBFORMVIEW { ParseCOPSubFormView $1 }
667 | T_TABLERELATION { ParseCOPTableRelation $1 }
  | T_VALUESALLOWED { ParseCOPValuesAllowed $1 }

MenuItems:
  T_ID T_EQUAL T_MENUITEMS T_LCURLY MenuItemsList T_RCURLY { COPMenuItems($1,$5) }

672 MenuItemsList:

```

```

MenuItem { [$1] }
| MenuItem MenuItemList { $1 :: $2 }

MenuItem:
677 T_LCURLY MenuItemPropList T_RCURLY { $2 }

MenuItemPropList:
MenuItemProp { [$1] }
| MenuItemProp T_SEMICOLON { [$1] }
682 | MenuItemProp T_SEMICOLON MenuItemPropList { $1 :: $3 }

MenuItemProp:
MenuItemProperty { ($1, Parsing.symbol_end_pos()) }
| Trigger { (MIPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
687

MenuItemProperty:
T_ID T_EQUAL Exp { ExtractMenuItemProperty($1,$3) }
| T_CAPTIONML { ParseMIPCaptionML $1 }
| T_RUNFORMLINK { ParseMIPRunFormLink $1 }
692 | T_RUNFORMVIEW { ParseMIPRunFormView $1 }
| T_RUNOBJECT { ParseMIPRunObject $1 }
| T_SHORTCUTKEY { ParseMIPShortCutKey $1 }

697 // -- NAV Report --
Report:
T_REPORTHEADER T_LCURLY ReportBody T_RCURLY { let header = ParseRegex "OBJECT Report
([0-9]+) (.*)" $1 in
(List.nth header 0, List.nth header 1, $3) }

702 ReportBody:
ObjectProperties ReportProperties DataItems RequestForm Code { ($1,$2,$3,$4,$5) }

ReportProperties:
T_PROPERTIES T_LCURLY ReportPropList T_RCURLY { $3 }
707

ReportPropList:
{ [] }
| ReportProp T_SEMICOLON ReportPropList { $1 :: $3 }

712 ReportProp:
ReportProperty { ($1, Parsing.symbol_end_pos()) }
| Trigger { (REPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
| Permissions { (REPPerm($1), Parsing.symbol_end_pos()) }

717 ReportProperty:
T_ID T_EQUAL Exp { ExtractReportProperty($1,$3) }
| T_CAPTIONML { ParseREPCaptionML $1 }
| T_PAPERSIZE { ParseREPPaperSize $1 }

722 DataItems:
T_DATAITEMS T_LCURLY DataItemList T_RCURLY { $3 }

DataItemList:
{ [] }
727 | DataItem DataItemList { $1 :: $2 }

DataItem:
T_LCURLY DataItemProperties Sections T_RCURLY { ($2,$3) }

732 DataItemProperties:
T_PROPERTIES T_LCURLY DataItemPropList T_RCURLY { $3 }

DataItemPropList:
{ [] }
737 | DataItemProp T_SEMICOLON DataItemPropList { $1 :: $3 }

DataItemProp:
DataItemProperty { ($1, Parsing.symbol_end_pos()) }
| Trigger { (DIPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
742

DataItemProperty:
T_ID T_EQUAL Exp { ExtractDataItemProperty($1,$3) }
| T_CALCFIELDS { ParseDIPCalcFields $1 }
| T_DATAITEMLINK { ParseDIPDataItemLink $1 }
747 | T_DATAITEMLINKREFERENCE { ParseDIPDataItemLinkReference $1 }
| T_DATAITEMTABLEVIEW { ParseDIPDataItemTableView $1 }
| T_DATAITEMVARNAME { ParseDIPDataItemVarName $1 }
| T_GROUPTOTALFIELDS { ParseDIPGroupTotalFields $1 }
| T_REQFILTERFIELDS { ParseDIPReqFilterFields $1 }
752 | T_REQFILTERHEADINGML { ParseDIPReqFilterHeadingML $1 }
| T_TOTALFIELDS { ParseDIPTotalFields $1 }

Sections:

```

```

T_SECTIONS T_LCURLY SectionList T_RCURLY { $3 }
757 SectionList:
    { [] }
    | Section SectionList { $1 :: $2 }

762 Section:
    T_LCURLY SectionProperties SectionControls T_RCURLY { ($2,$3) }

SectionProperties:
    T_PROPERTIES T_LCURLY SectionPropList T_RCURLY { $3 }
767 SectionPropList:
    { [] }
    | SectionProp T_SEMICOLON SectionPropList { $1 :: $3 }

772 SectionProp:
    SectionProperty { ($1, Parsing.symbol_end_pos()) }
    | Trigger { (SEPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }

SectionProperty:
777 T_ID T_EQUAL Exp { ExtractSectionProperty($1,$3) }

SectionControls:
    T_CONTROLS T_LCURLY SectionControlsList T_RCURLY { $3 }

782 SectionControlsList:
    { [] }
    | SectionControl SectionControlsList { $1 :: $2 }

SectionControl:
787 SectionControlHeader T_RCURLY { ($1,[]) }
    | SectionControlHeader T_SEMICOLON SectionControlPropList T_RCURLY { ($1,$3) }

SectionControlHeader:
    T_LCURLY T_INT T_SEMICOLON T_ID T_SEMICOLON XPos T_SEMICOLON T_INT T_SEMICOLON T_INT
    T_SEMICOLON T_INT { ($2,$4,$6,$8,$10,$12) }
792 SectionControlPropList:
    SectionControlProp { [$1] }
    | SectionControlProp T_SEMICOLON SectionControlPropList { $1 :: $3 }

797 SectionControlProp:
    SectionControlProperty { ($1, Parsing.symbol_end_pos()) }

SectionControlProperty:
    T_ID T_EQUAL Exp { ExtractSectionControlProperty($1,$3) }
802 | T_BORDERWIDTH { ParseSCPBorderWidth $1 }
    | T_CAPTIONML { ParseSCPCaptionML $1 }
    | T_DECIMALPLACES { ParseSCPDecimalPlaces $1 }
    | T_FONTNAME { ParseSCPFontName $1 }
    | T_FORMAT { ParseSCPFormat $1 }
807 | T_NAME { ParseSCPName $1 }
    | T_OPTIONCAPTIONML { ParseSCPOptionCaptionML $1 }
    | T_OPTIONSTRING { ParseSCPOptionString $1 }
    | T_PADCHAR { ParseSCPPadChar $1 }

812 RequestForm:
    T_REQUESTFORM T_LCURLY RequestFormProperties Controls T_RCURLY { ($3,$4) }

RequestFormProperties:
817 T_PROPERTIES T_LCURLY RequestFormPropList T_RCURLY { $3 }

RequestFormPropList:
    { [] }
    | RequestFormProp T_SEMICOLON RequestFormPropList { $1 :: $3 }
822 RequestFormProp:
    RequestFormProperty { ($1, Parsing.symbol_end_pos()) }
    | Trigger { (RFPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
    | Permissions { (RFPPerm($1), Parsing.symbol_end_pos()) }
827 RequestFormProperty:
    T_ID T_EQUAL Exp { ExtractRequestFormProperty($1,$3) }
    | T_CAPTIONML { ParseRFPCaptionML $1 }

```