

**PENNY : A Programming Language and Compiler for the
Context Interchange Project**

by

Fortunato Pena

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

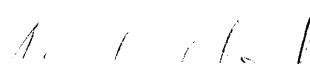
May 1997

© Fortunato Pena, MCMXCVII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part, and to grant
others the right to do so.



Author.....
Department of Electrical Engineering and Computer Science
May 23, 1997



Certified by.....
Michael D. Siegel
Principal Research Scientist, Sloan School of Management
Thesis Supervisor



Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

OCT 29 1997

ENG.

LIBRARIES

**PENNY : A Programming Language and Compiler for the Context
Interchange Project**

by
Fortunato Pena

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this Thesis, I designed and implemented a programming language and compiler for use under the *COntext INterchange* Architecture, MINT. The language, known as PENNY, was born from COINL, a deductive object-oriented language originally used to program under the MINT architecture. This new language boasts a new look and feel to provide a greater abstraction and minimize the complexity for the user. Furthermore, a compiler has been implemented which uses PENNY as its source language and DATALOG with negation as its destination language. Two existing applications have been re-coded in PENNY to demonstrate the language's ease of use and the compiler's capabilities.

Thesis Supervisor: Michael D. Siegel

Title: Principal Research Scientist, Sloan School of Management

Acknowledgments

As I say farewell to this phase of my life, I would like to acknowledge and thank the many people who have made these past five years a more enjoyable, if not bearable, time for me.

First of all, I would like to thank Professor Stuart Madnick and Dr. Michael Siegel for giving me the opportunity to work in the Context Interchange Project and for their invaluable advice and support. I would also like to thank the other members of the Context Interchange team who have provided me with a tremendous working environment over the past year and a half. In particular, I am indebted to Dr. Stéphane Bressane for his role as a mentor and his consistence pressure to keep me succeeding and Dr. Cheng Hian Goh who beared with me in my first couple months as I grappled with the challenges of a new learning environment. My gratitude also goes out to Tom Lee and Kofi Duodu Fynn for their wonderful insights and support they have given me.

A number of other people have read various drafts of this Thesis and provided me with valuable feedback. These include Professor Stuart Madnick, Dr. Stéphane Bressane, and Dr. Raphael Yahalom. In particular, I would like to thank Raphael for taking time out of his extremely busy schedule to read my Thesis and provide me with a number of wonderful suggestions.

To my friends, both here and back home, thank you all for putting up with me and encouraging me to stay focused. These past couple of years have been strenuous and I could not have made it without any of your support, in particular, Henry, Danika, Kofi, Maricruz, Francine, Doug, and Chuck. Thanks for the reading material, Chuck, I am sure I will find it invaluable as I continue with the next phase of my life.

Finally, I would like to express my appreciation and love to my siblings, Lisette and John and my parents Fortunato and Julia. Without your constant love and phone calls in the morning to wake me up for class, none of this would have been possible. It is to you that I dedicate this Thesis.

Contents

1	Introduction	7
1.1	Motivational Scenario	8
1.2	Organization of Thesis	9
2	Penny Programming	11
2.1	Background	11
2.1.1	COINL Domain Model	12
2.1.2	COINL Context Axioms	14
2.1.3	COINL Conversion Functions	14
2.1.4	COINL Elevation Axioms	15
2.2	PENNY	17
2.2.1	Motivations	17
2.2.2	Scenario	17
3	Penny Compiler	22
3.1	Related Works	22
3.2	Global Picture	23
3.3	Implementation	23
3.3.1	Lexical Analysis Module	25
3.3.2	Parser Module	25
3.3.3	Semantics Module	27
3.3.4	Code Generator Module	28
4	Conclusion and Future Work	29
4.1	Limitations	29
4.1.1	PENNY	29
4.1.2	PENNY Compiler	29
4.2	Conclusion	30
A	Penny BNF	31
A.1	Notational Convention	31
A.2	Reserved PENNY Keywords	31
A.3	BNF	32
B	TASC Penny Axioms	37
B.1	Domain Model	37
B.2	Context Axioms	38
B.2.1	DiscAF	38

B.2.2	WorldAF	38
B.2.3	DStreamAF	39
B.2.4	Olsen	39
B.3	Conversion Functions	39
B.4	Elevation Axioms	42
C	DLA Penny Axioms	46
C.1	Domain Model	46
C.2	Context Axioms	47
C.2.1	Distributor	47
C.2.2	Manufacturer	47
C.2.3	Olsen	47
C.3	Conversion Functions	47
C.4	Elevation Axioms	51

List of Figures

1-1	TASC Data sources	8
1-2	Results of Query	10
1-3	Assumptions of Sources	10
1-4	Correct Results of Query	10
2-1	TASC Domain Model	13
3-1	PENNY Compiler in the COIN Architecture	24

Chapter 1

Introduction

In recent years, advances in networking and telecommunications have led to an unprecedented growth in the number of information sources that are being physically connected together. This increased connectivity has given way to a proliferation of data readily accessible to users.

Unfortunately, with this growing abundance of data, the problem of understanding and interpreting it all is becoming a definite challenge [Mad96]. While the World Wide Web and Internet have provided an excellent infrastructure for the physical connectivity (the ability to exchange bits and bytes) amongst disparate data sources and receivers, they have failed to provide a reasonable amount of logical connectivity (the ability to exchange data meaningfully) among them. This logical connectivity can be essential when manipulating information from disparate data sources because the meaning of this information can be dependent on a particular context; a context which embodies a number of underlying assumptions. One example which illustrates this problem clearly is the handling of dates. The date “02-01-97” will mean February 1, 1997 if speaking to an American whereas it will mean January 2, 1997 if speaking to a European. This problem is generally referred to as the need for *semantic interoperability* among distributed data sources and as a result, any data integration effort must be capable of reconciling possible semantic conflicts among sources and receivers.

The *COntext INterchange* (COIN) project seeks to address the problem of *semantic interoperability* by providing a novel mediator-based architecture ¹ [BFP⁺97a, Goh96] for logical connectivity among disparate data sources. In essence, each information source in the architecture is tagged with a context; a context being a set of axioms which describes certain assumptions about the data. For instance, if \mathcal{A} is a database containing information on a company’s finances and the dates in which those finances were recorded, then the context associated with database \mathcal{A} would contain any underlying information needed to properly interpret those financial figures and dates, such as the scale-factor, the currency, and the format of the date. Using this context information the MINT architecture is able to resolve semantic conflicts through the use of its *Context Mediator* [BFP⁺97a, BLGea97]. The context axioms, along with general axioms (known as the COIN axioms) which will be discussed in Chapter 2, are coded by the users of the system and compiled into a format that the *Context Mediator* [BLGea97] can use. The *Mediator* ², based on an abduction

¹The Context Interchange architecture shall be referred to as MINT throughout this Thesis

²The word *Mediator* and *Context Mediator* are synonymous in this Thesis

WorldAF

Company_Name	Sales	Income	Date
Daimler-Benz AG	56,268,168	346,577	12/31/93

Olsen

FromCur	DEM
ToCur	USD
Rate	0.58
Date	31/12/93

DiscAF

Company_Name	Sales	Income	Date
Daimler-Benz AG	97,000,000,000	615,000,000	12/31/93

Figure 1-1: TASC Data sources

procedure described in [Goh96, BLGea97], automatically determines and resolves potential semantic conflicts among the different sets of data using the knowledge axioms.

The goal of this Thesis is thus two-fold. First, to provide a new programming language for the COIN axioms. Second, to construct a compiler for this new language as its source language and Datalog [CGT89] with negation as its destination language to implement the desired computational processes.

1.1 Motivational Scenario

The motivational scenario described herein shall be the foundation on which examples in this Thesis shall build upon. This scenario is an application that is currently running as a demonstration in our architecture. The code for the entire scenario is given in ppendix B. This scenario shall be referred to as the TASC demo throughout this Thesis.

The TASC demo consists of integrating information from three disparate tables: DiscAF, WorldAF, and DStreamAF (although for the sake of simplicity, our discussion shall focus only on the first two). These relations provide financial information on companies such as their profits, sales, number of employees, etc. In addition, there is an auxiliary table, Olsen, which is a web based source providing exchange rates between different currencies.

The DiscAF relation contains information for both foreign and domestic companies. WorldAF maintains information mainly on domestic companies, but it does have several foreign company listings as well. Both of these tables have information on the company Daimler Benz (although the names in both relations are not identical in reality, for didactical purposes, they have been made the same). Suppose that a user submits the following query over the relations shown in Figure 1-1 (no one would ever send a query like the following, but this is done just for illustrative purposes).

```
select w.Sales, d.Sales from WorldAF w, DiscAF d
where w.Comp = "DAIMLER-BENZ AG" and d.Comp = w.Comp;
```


Viewing the results returned in Figure 1-2, there appears to be a discrepancy in the data set. How is it possible that the sales figure for the same company differs?

The dilemma here is that the values returned are not represented using the same set of concepts. Each relation has its own set of assumptions which changes the interpretation of its data. DiscAF assumes that company financials are stored in the currency of the country of incorporation for that company and scaled by 1. Also, the format of the date field is kept in an American style using a '/' to separate the day, month, and year. WorldAF on the other hand, stores its company financial data in a US currency scaled by 1000. As DiscAF, it also assumes an American style format for its date field. In addition to DiscAF and WorldAF, Olsen and the user of the system also have their own set of assumptions. For example, Olsen assumes that dates are given in a European style (this is probably because the site is located in Zürich) with the day, month, and year separated by '/' and that currencies are three letter words denoting the currency type, so for US Dollars, it is 'USD' and for Deutsch Marks it is 'DEM'. Figure 1-3 layouts the underlying assumptions for each of the data sources as well as the user. Using this underlying information, the *Context Mediator* can automatically detect and reconcile these context differences, returning the correct answer back to the user as shown in Figure 1-4 (note, that because of arithmetic rounding errors, the figures are not exactly identical). In Chapter 2, it shall be demonstrated how these assumptions and relations are coded in MINT.

1.2 Organization of Thesis

This Thesis is organized into four chapters. In Chapter 2, we discuss the language PENNY and focus on how and why PENNY was developed from its predecessor COINL. Chapter 3 gives the implementation of the PENNY compiler and the reasons for its design decisions. Chapter 4 concludes this Thesis with a summary of contributions and a number of suggestions on how the current work may be extended both with respect to PENNY and its compiler.

w.Sales	d.Sales
56,268,168	97,000,000,000

Figure 1-2: Results of Query

	ScaleFactor	Currency	DateFormat
WorldAF	1000	USD	American "/"
DiscAF	1	Local	American "/"
Olsen	NA	NA	European "/"
User	1000	USD	American "/"

Figure 1-3: Assumptions of Sources

w.Sales	d.Sales
56,268,168	56,687,460

Figure 1-4: Correct Results of Query

Chapter 2

Penny Programming

PENNY was born from COINL [Goh96], a deductive object-oriented language ¹ originally used to program under MINT [BFP⁺97a]. Various MINT programming examples will be illustrated in this Chapter. These examples will first be written in COINL and then in PENNY to give us a basis for discussion.

The syntax which is used to illustrate the examples in this chapter deviates from the concrete syntax (the syntax used to actually type programs into the computer) in several characters, not available on a standard keyboard. We give the standard equivalent of our special characters. Thus `~>` should be used instead of \rightsquigarrow , `<-` should be used instead of \leftarrow , `->` should be used instead of \rightarrow , `=>` should be used instead of \Rightarrow , `~` should be used instead of \sim , and `^` should be used instead of \wedge . Furthermore, **bold facing** is used to signify keywords in the languages.

2.1 Background

There are six steps that need to be taken to properly program a new data source into MINT. The steps are:

1. Create and define a domain model
2. Define the context axioms for the source
3. Define the necessary conversion functions
4. Elevate each source into a semantic (“virtual”) relation
5. Define the integrity constraints on the relations being incorporated
6. Populate the registry with the proper export schemas

Currently, there is no support in PENNY or COINL for integrity constraints or registry information. These are expressed outside the language ² and compiled along with the other COIN axioms. Thus our efforts shall focus on items 1–4.

¹At this point, the reader of this Thesis is assumed to have a working knowledge of Prolog and the way Prolog systems work. Otherwise the reader is referred to two excellent books which cover the topic in more detail than necessary, [SS94, O’K90].

²For a more detailed discussion on these two topics in MINT, the reader is referred to [Goh96, BFP⁺97b]

2.1.1 Coinl Domain Model

A *domain model* specifies the semantics of the “types” of information units which constitutes a common vocabulary used in capturing the semantics of data in disparate sources, i.e. it defines the ontology which will be used. The various semantic types, the type hierarchy, and the type signatures (for *attributes* and *modifiers*) are all defined in the domain model. Types in the generalization hierarchy are rooted to system types, i.e. types native to the underlying system such as integers, strings, reals, etc.

Figure 2-1 below is the domain model which is used in the TASC scenario. Examining it closely, there are several things to notice.

First, there are five semantic types defined: **companyFinancials**, **currencyType**, **exchangeRate**, **date**, **companyName**, and **countryName**. Two of them, **companyFinancials** and **exchangeRate**, subclass the system type **number**, while the other three are derived from the system type **string**.

Second, the signatures for various *attributes* are defined. In COIN [Goh96], objects have two form of properties, those which are structural properties of the underlying data source and those that encapsulate the underlying assumptions about a particular piece of data. *Attributes* access structural properties of the semantic object in question. So for instance, the semantic type **companyFinancials** has two attributes, **company** and **fyEnding**. Intuitively, these attributes define a relationship between objects of the corresponding semantic types. Here, the relationship formed by the **company** attribute states that for any company financial in question, there must be a corresponding company to which it belongs (it doesn't make any sense to have financial data about a non-existent company). Similarly, the **fyEnding** attribute states that every company financial object has a date when it was recorded.

Finally, there are a set of signatures for the *modifiers* in the system. *Modifiers* also define a relationship between semantic objects of the corresponding semantic types. The difference, though, is that the values of the semantic objects defined by modifiers have varying interpretations depending on the context. So as an example, the semantic type **companyFinancials** defines two modifiers, **scaleFactor** and **currency** (you can distinguish *modifiers* from *attributes* by the fact that *modifiers* take in a context argument given by the **ctx** placeholder). The value of the object returned by the modifier **scaleFactor** depends on a given context. So if one is asking for the scale-factor in the DiscAF context, it is shown in Figure 1-3 to be 1, whereas if one asked for the scale-factor in the WorldAF context, the value of the object returned would be 1000. Similarly, the same can be said about the modifier **currency**. In the DiscAF context it returns an object whose value is the local currency of the country of incorporation as opposed to a US currency in the WorldAF context.

Figure 2-1 illustrates these concepts pictorially. Following the links on the graph, it is trivial to see how modifiers and attributes work; they just return objects of the given types.

1. **companyFinancials** :: **number**.
 companyFinancials[**company** ⇒ **companyName**].
 companyFinancials[**fyEnding** ⇒ **date**].
 companyFinancials[**scaleFactor**(**ctx**) ⇒ **number**].
 companyFinancials[**currency**(**ctx**) ⇒ **currencyType**].
2. **currencyType** :: **string**.
 currencyType[**curTypeSym**(**ctx**) ⇒ **string**].

3. `companyName :: string`.
`companyName[format(ctx) ⇒ string]`.
`companyName[countryIncorp ⇒ string]`.
4. `date :: string`.
`date[dateFmt(ctx) ⇒ string]`.
5. `exchangeRate :: number`.
`exchangeRate[fromCur ⇒ currencyType]`.
`exchangeRate[toCur ⇒ currencyType]`.
`exchangeRate[txnDate ⇒ date]`.
6. `countryName :: string`.
`countryName[officialCurrency ⇒ currencyType]`.

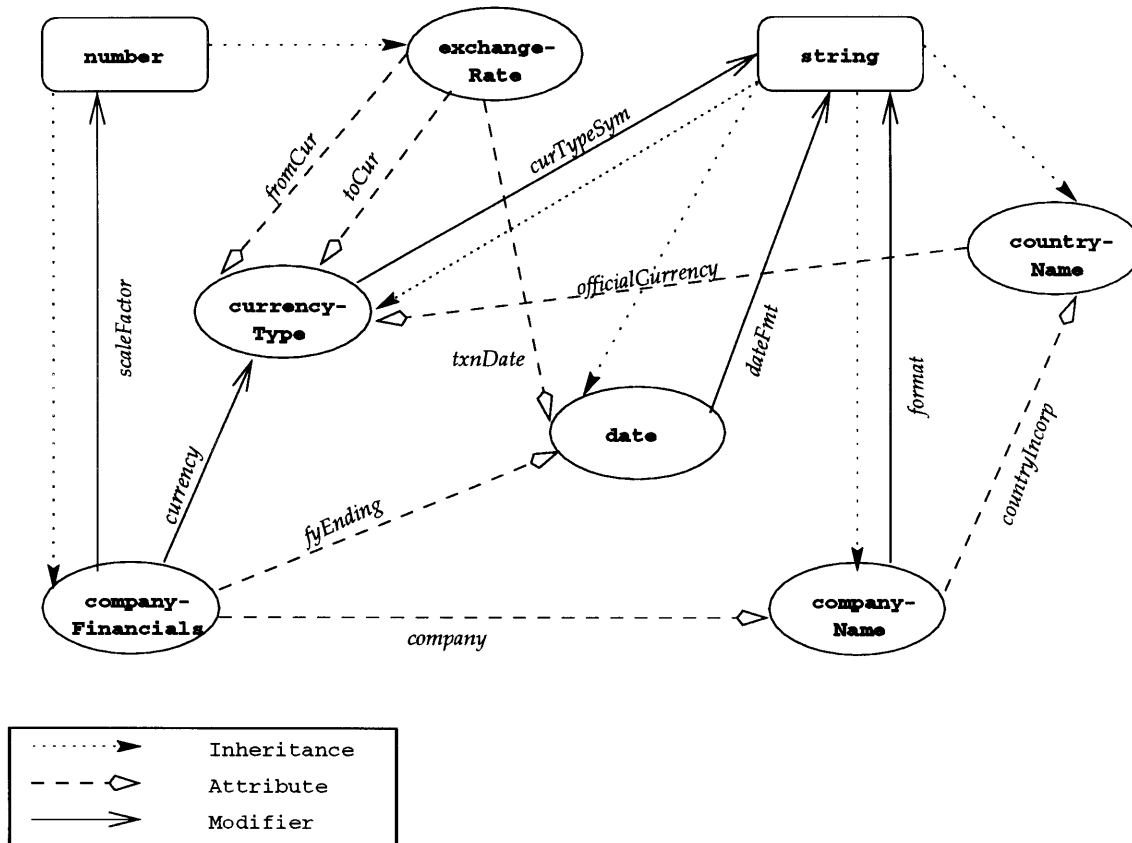


Figure 2-1: TASC Domain Model

Before moving on, there is one key concept left to explain. The term “semantic object” has been used several times already, but yet no definition was given. Essentially, a semantic object is a concept abstracting the data from the peculiarities of its representation in the source and from the assumptions underlying its interpretation [BLGea97]. For example, from Figure 1-3, the values 346,577 and 615,000,000 express the same money amount except using a different currency and scale-factor. These assumptions can be characterized by a

number of concepts (`scaleFactor`, `currency`, `format`, etc). Thus it is these concepts which define the basis for the modifiers.

2.1.2 Coinl Context Axioms

The next objective is to define the context axioms for the TASC demo. Rather than trying to define the axioms for all of the contexts used, this discussion shall focus only on the DiscAF context (`c_ds`).

Context axioms are a set of definitions for the modifiers of each semantic type given in the domain model. Remember that the value of the objects returned by a modifier depend on a given context. Thus the objects returned by the modifiers must be created and assigned a value; a value which will be dependent on the context in which they are created.

1. `:- begin_module(c_ds).`
2. `X : companyFinancials ~> scaleFactor(C, X)[value(C) -> V] <- V = 1.`
3. `X : companyName ~> format(C, X)[value(C) -> V] <- V = "ds_name".`
4. `X : date ~> dateFmt(C, X)[value(C) -> V] <- V = "American Style /".`
5. `X : companyFinancials ~> currency(C, X)[value(C) -> V] <-`
`X[company -> Comp],`
`Comp[countryIncorp -> Country],`
`Country[officialCurrency -> CurrencyType],`
`CurrencyType[value(C) -> V].`
6. `X : currencyType ~> curTypeSym(C, X)[value(C) -> V] <- V = "3char".`

From Figure 2-1, it is clear that the semantic type `date` has one modifier, `dateFmt`. This modifier returns an object of type `string` whose value is "American Style /" [Line 4] (if you are wondering where the value came from, refer back to Figure 1-3 and look at the format of the date in the DiscAF context). Similarly, the modifier `scaleFactor` needs to return an object of type `number` whose value is 1 [Line 2]. Modifiers can also be defined intensionally, as given in Line 5. Line 5 defines the value of the object returned by the `currency` modifier to be (reading the rule bottom up), the value for the currency of the object `X` is obtained by retrieving the official currency from the country of incorporation, which is obtained by retrieving the country from the company, which is obtained by retrieving the company from the object in question.

There are two last items to note. First, modifiers are tied, or bound, to a specific semantic type. The modifier `scaleFactor` is defined with respect to the type `companyFinancials`. Likewise, the modifier `dateFmt` is bound to the type `date`. This permits overriding of modifiers in a local context. Second, the encapsulation of these axioms is defined via the `:- begin_module(c_ds)` directive. This directive dictates the start of a context which extends to the end of the file.

2.1.3 Coinl Conversion Functions

The third item that needs to be discussed are conversion functions. Conversion functions define how the value of a given semantic object can be derived in the current context, given that its value is known with respect to a different context.

1. $X : \text{companyFinancials} \rightsquigarrow X[\text{cvt}(Tgt)@\text{scaleFactor}, Src, Val \rightarrow Res] \leftarrow$
 $\text{scaleFactor}(Src, X)[\text{value}(Tgt) \rightarrow Fsv],$
 $\text{scaleFactor}(Tgt, X)[\text{value}(Tgt) \rightarrow Fsv],$
 $Res = Val.$
2. $X : \text{companyFinancials} \rightsquigarrow X[\text{cvt}(Tgt)@\text{scaleFactor}, Src, Val \rightarrow Res] \leftarrow$
 $\text{scaleFactor}(Src, X)[\text{value}(Tgt) \rightarrow Fsv],$
 $\text{scaleFactor}(Tgt, X)[\text{value}(Tgt) \rightarrow Ftv],$
 $Fsv <> Ftv,$
 $Ratio \text{ is } Fsv / Ftv,$
 $Res \text{ is } Val * Ratio.$

The conversion functions given above, define the conversion for scale-factors among different company financials. The idea behind scale-factor conversions is to look for the scale-factor in the two different contexts and then multiply the value of the companyFianancial object by the ratio of the two scale-factors. This is defined via Rule 2. The first Rule just accounts for the cases when the scale-factors in both contexts are the same, in which case, there is no need to convert; the given value is passed back out (The first rule is an optimization technique. Although Rule 2 works even when the scale-factors are identical, Rule 1 will save us a database access).

2.1.4 Coinl Elevation Axioms

The mapping of data and data-relationships from the sources to the domain model is accomplished via the elevation axioms. There are three distinct operations which define the elevation axioms [Goh96]:

- Define a virtual semantic relation corresponding to each extensional relation
- Assign to each semantic object defined, its value in the context of the source
- Map the semantic objects in the semantic relation to semantic types defined in the domain model and make explicit any implicit links (attribute initialization) represented by the semantic relation

Examining the elevation axioms for DiscAF below, it shall be shown how each of the three criterion above is met.

1. $:- \text{dynamic 'DiscAF_p'}/7.$
2. $\text{'DiscAF_p'}(\text{f_ds_cname}(N, D), \text{f_ds_fyEnding}(N, D), \text{f_ds_shares}(N, D),$
 $\text{f_ds_income}(N, D), \text{f_ds_sales}(N, D), \text{f_ds_assets}(N, D), \text{f_ds_incorp}(N, D))$
 $\leftarrow \text{'DiscAF'}(N, D, -, -, -, -).$
3. $\text{f_ds_cname}(-, -) : \text{companyName}.$
 $\text{f_ds_cname}(N, D)[\text{countryIncorp} \rightarrow \text{f_ds_incorp}(N, D)].$
 $\text{f_ds_cname}(N, D)[\text{value}(c_ds) \rightarrow N] \leftarrow \text{'DiscAF'}(N, D, -, -, -, -).$
4. $\text{f_ds_fyEnding}(-, -) : \text{date}.$
 $\text{f_ds_fyEnding}(N, D)[\text{value}(c_ds) \rightarrow D] \leftarrow \text{'DiscAF'}(N, D, -, -, -, -).$
5. $\text{f_ds_shares}(-, -) : \text{number}.$
 $\text{f_ds_shares}(N, D)[\text{value}(c_ds) \rightarrow S] \leftarrow \text{'DiscAF'}(N, D, S, -, -, -, -).$

6. $f_ds_income(-,-) : companyFinancials$.
 $f_ds_income(N,D)[company \rightarrow f_ds_cname(N,D)]$.
 $f_ds_income(N,D)[fyEnding \rightarrow f_ds_fyEnding(N,D)]$.
 $f_ds_income(N,D)[value(c_ds) \rightarrow I] \leftarrow 'DiscAF'(N,D,-,I,-,-)$.
7. $f_ds_sales(-,-) : companyFinancials$.
 $f_ds_sales(N,D)[company \rightarrow f_ds_cname(N,D)]$.
 $f_ds_sales(N,D)[fyEnding \rightarrow f_ds_fyEnding(N,D)]$.
 $f_ds_sales(N,D)[value(c_ds) \rightarrow S] \leftarrow 'DiscAF'(N,D,-,S,-,-)$.
8. $f_ds_assets(-,-) : companyFinancials$.
 $f_ds_assets(N,D)[company \rightarrow f_ds_cname(N,D)]$.
 $f_ds_assets(N,D)[fyEnding \rightarrow f_ds_fyEnding(N,D)]$.
 $f_ds_assets(N,D)[value(c_ds) \rightarrow A] \leftarrow 'DiscAF'(N,D,-,A,-)$.
9. $f_ds_incorp(-,-) : countryName$.
 $f_ds_incorp(N,D)[officialCurrency \rightarrow f_ds_curType(N,D)]$.
 $f_ds_incorp(N,D)[value(c_ds) \rightarrow Y] \leftarrow 'DiscAF'(N,D,-,Y,-)$.
10. $f_ds_curType(-,-) : currencyType$.
 $f_ds_curType(N,D)[value(c_ds) \rightarrow V] \leftarrow$
 $'DiscAF'(N,D,-,V,-),$
 $Incorp[officialCurrency \rightarrow f_ds_curType(N,D)],$
 $Incorp[value(c_ds) \rightarrow Y],$
 $'Currencytypes'(Y, V)$.

The first thing that needs to be done is to define a semantic relation for DiscAF. A semantic relation is defined on the semantic objects in the corresponding cells. The data elements derived from the extensional relation are mapped to semantic objects. These semantic objects define a unique object-id for each data element. In the listing above, this is accomplished in Line 2.

Next, for each semantic object defined, it is assigned a value in the context of the source. These rules are shown in Lines 3–10. For example, the following rule,

$$f_ds_incorp(N,D)[value(c_ds) \rightarrow Y] \leftarrow 'DiscAF'(N,D,-,Y,-)$$

states that the value of the semantic object $f_ds_incorp(N,D)$ in the DiscAF context is derived from the external relation 'DiscAF'.

Finally, every semantic object is mapped to a semantic type defined in the domain model and any links present in the semantic relation are explicitly established. This is given by the rules:

$$f_ds_incorp(-,-) : countryName.$$

$$f_ds_incorp(N,D)[officialCurrency \rightarrow f_ds_curType(N,D)].$$

Here the semantic object $f_ds_incorp(-,-)$ is defined to be of type **countryName**. Furthermore, the link between the country name object and currency object, as given in the domain model (Figure 2-1), is initialized.

There are several things to note in these elevation axioms. First, not all semantic objects need be derived from a semantic relation. Take for instance, the object $f_ds_curType(-,-)$. It is not an elevated semantic object. Rather it is a user defined semantic object and has

no existence in the relation 'DiscAF'. These are called virtual semantic objects, virtual because they have no grounding in a semantic relation. In this case, the semantic object `f_ds.curType(-,-)` is created as part of the DiscAF source. As with any other semantic objects, the proper links need to be initialized and its value must be defined in the context of the source.

2.2 Penny

2.2.1 Motivations

After working in COINL for developing various applications in the group, it was quite obvious that the COINL syntax was not meeting the needs of the users. Although syntax is concerned only with the form of a program, it is inextricably tied to “semantics”, which is the meaning of the program. Since the basic goal of a programming language design is to define the means for describing computational processes, the syntax occurs principally to serve these semantic ends. Thus semantic goals are the original motivation for syntax design [Tuc86]. Syntax was but one reason to move towards the development of a new language. The second dealt with the complexity in programming in COINL such as dealing with all of the special cases, exceptions, and complex notation. Thus the efforts in creating PENNY also focused on removing this complexity from the language and pushing it into the compiler, an issue which is discussed in more detail in Section 4.1.1.

2.2.2 Scenario

The ideas and theory behind the integration of data sources into MINT were described in the previous section (Section 2.1). As a result, this section will only give details on how the same elements are implemented in PENNY.

Penny Domain Model

Once again, creating the domain model for the TASC demo, here is how the axioms are defined.

1. **semanticType** `companyFinancials::number` {
 attribute `companyName` `company`;
 attribute `date` `fyEnding`;
 modifier `number` `scaleFactor(ctx)`;
 modifier `currencyType` `currency(ctx)`;
};
2. **semanticType** `companyName::number` {
 modifier `string` `format(ctx)`;
 attribute `string` `countryIncorp`;
};
3. **semanticType** `exchangeRate::number` {
 attribute `currencyType` `fromCur`;
 attribute `currencyType` `toCur`;
 attribute `date` `txnDate`;
};

4. **semanticType** date::string {
 modifier string dateFmt(**ctx**);
 };
5. **semanticType** currencyType::string {
 modifier string curTypeSym(**ctx**);
 };
6. **semanticType** countryName::string {
 attribute currencyType officialCurrency;
 };

As before, there are five semantic types defined, each given by the keyword **semanticType**. As an example, **companyFinancials** is declared to contain two attributes and two modifiers. Its attribute **company** returns an object of type **companyName** (notice that the syntax of methods is similar to function declarations in C or C++). Its modifier **scaleFactor** returns an object of type **number**. All modifiers are defined via the keyword **modifier** (attributes are defined by **attribute**) and they take in a context given by the keyword placeholder, **ctx**. The other types in the generalization hierarchy are likewise defined.

Penny Context Axioms

Modifiers in PENNY are initialized as follows:

1. **use** ('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');
2. **context** c_ds;
3. **scaleFactor**<companyFinancials> = ~(1);
4. **currency**<companyFinancials> = ~(\$) ←
 Comp = **self**.company,
 Country = *Comp*.countryIncorp,
 CurrencyType = *Country*.officialCurrency,
 \$ = *CurrencyType*.value;
5. **format**<companyName> = ~("ds_name");
6. **dateFmt**<date> = ~("American Style /");
7. **curTypeSym**<currencyType> = ~("3char");
8. **end** c_ds;

In Line 3,

```
scaleFactor<companyFinancials> = ~(1);
```

the modifier **scaleFactor** returns an object whose value is 1. The first thing to notice is how the object is created. The **~** operator is used to create virtual semantic objects (these were covered in Section 2.1.4. The declaration **~(1)** creates a virtual object whose value is initialized to 1. This object is then assigned to the **scaleFactor** modifier. The virtual object created here as no name or explicit type. The compiler takes care of generating the

unique object-id for the virtual object and generating its correct type using the information in the domain model. But because this virtual semantic object has no name attached to it, it is impossible to reference it anywhere else in the rule.

In addition, it is possible to define modifiers intensionally, as shown in the definition for the `currency` modifier below.

```
currency<companyFinancials> = ~($) ←
    Comp = self.company,
    Country = Comp.countryIncorp,
    CurrencyType = Country.officialCurrency,
    $ = CurrencyType.value;
```

Methods (attributes and as one shall see later, modifiers), are referenced through the dot '.' operator, though only one level of dereferencing is currently allowed. The syntax is very similar to other object oriented languages such as OQL or C++. Second, the ← operator is the rule operator which separates the head from the body of a rule³. Third, you will notice the reference to the `self` identifier. This identifier corresponds to the `this` pointer in C++ or the `self` reference in SmallTalk, and is used to reference the object which was used to invoke this method. Finally, the \$ operator holds the value of the rule. Thus the value for the aforementioned rule is given by the line,

```
$ = CurrencyType.value;
```

There are several other new things which should be pointed out. First, you can now cross-reference files (with either a full or relative pathname), as shown in Line 1, through the use of the `use` keyword. It is similar to the `include` in C. The difference is, though, if you include a domain model, the statements are not compiled; they are only used as auxiliary information. Second, context definitions are given by `context c_ds`; and terminated by `end c_ds`. The context name must be the same in both places. Thus several contexts can exist mutually in the same file. Furthermore, the binding of the modifiers to a particular type is done through the type binding mechanism, `<type>`. So in the above example, `scaleFactor` is bound to the type `companyFinancials`.

Penny Conversion Functions

The conversion functions in PENNY are not very different than in COINL, with two exceptions. First notice how attributes and modifiers are referenced. As mentioned in the previous section, the dot (.) operator is used to dereference methods. Second, notice that the `value` methods no longer take in a context argument. This is because the compiler actually has enough information to automatically generate the missing context argument.

The type binding mechanism for conversion functions is identical to the type binding found in the context axioms.

Below is an excerpt of the conversion functions needed to do scale-factor conversions. Again there are two rules, the first takes into account the cases where the scale-factor is the same. The second, retrieves the scale-factor from both contexts and multiplies the value of the company financial by the ratio.

³Rules are similar to COINL or DATALOG rules

1. **cvt**(scaleFactor, Val)<companyFinancials> ←
SrcMod = **self.scaleFactor**(**source**),
TgtMod = **self.scaleFactor**(**target**),
SrcMod.value = *TgtMod.value*,
\$ = Val;
2. **cvt**(scaleFactor, Val)<companyFinancials> ←
FsvMod = **self.scaleFactor**(**source**),
FtvMod = **self.scaleFactor**(**target**),
FsvMod.value <> *FtvMod.value*,
Ratio = *FsvMod.value* / *FtvMod.value*,
\$ = Val * *Ratio*;

Like COINL conversion functions, PENNY conversion functions are allowed to take in arguments. But unlike in COINL, the source and target contexts are not passed in. Rather they are explicitly called through the keywords, **source** and **target** (Remember that the idea of conversion functions is to take the value of an object in a known context (source) and convert its value into a another context (target)). Because the value of objects returned by modifiers depends on a particular context, when calling modifiers, it is necessary that the desired context be passed in as an argument.

SrcMod = **self.scaleFactor**(**source**)

Penny Elevation Axioms

Finally, one must define and declare the elevation axioms for the sources. Here the the elevation axioms for the DiscAF relation shall be used as an example.

1. **elevate** 'DiscAF'(cname, fyEnding, shares, income, sales, assets, incorp)
2. **in** c_ds
3. **as** 'DiscAF_p'(\wedge cname : companyName, \wedge fyEnding : date, \wedge shares : void,
 \wedge income : companyFinancials, \wedge sales : companyFinancials,
 \wedge assets: companyFinancials, \wedge incorp : countryName)
{
 \wedge cname.countryIncorp = \wedge incorp;

 \wedge income.company = \wedge cname;
income.fyEnding = \wedge fyEnding;

 \wedge sales.company = \wedge cname;
 \wedge sales.fyEnding = \wedge fyEnding;

 \wedge assets.company = \wedge cname;
 \wedge assets.fyEnding = \wedge fyEnding;

 \wedge incorp.officialCurrency = \sim curType;

 \sim curType.value = \$ ←

```

    ~curType = Incorp.officialCurrency,
    Y = Incorp.value,
    'Currencytypes'(Y, $);
};

```

Most of the three operations are accomplished in the first line. The first line gives the external relation being elevated. The second signifies in which context the source values are to be defined. Finally, the third line gives the elevated relation name and the elevated semantic objects (these are the objects preceded by the \wedge). You will notice some very obvious differences between how these axioms are described in PENNY versus COINL.

First, no explicit value declarations need to be given. The compiler will automatically generate these given the above information. Of course, it is possible to override the compiler's default value method by simply supplying one of your own. So say you wanted the value of the country of incorporation for DiscAF to be "Germany" for every entry in the relation. Then you could override the compiler's default value constructor by doing the following,

```

     $\wedge$ incorp.value = "Germany";

```

Second, no explicit semantic objects need to be given. The names given in the relation are so that it is possible to reference them when defining the links. Semantic object names are given by $\wedge name$, where *name* can be an atom or a variable. Otherwise, the names are scoped within the '{ '}' braces. The compiler takes care of generating unique semantic objects-ids and the types that go along with them. Again, one can always override the type definition by providing your own such as,

```

    ~curType : number;

```

Third, it is still possible to create virtual semantic objects using the \sim operator. The difference between the virtual objects created here and the ones shown in the context axioms is that these are named virtual object instances. In the context axioms, the virtual objects created were unnamed. Creating unnamed virtual objects in the elevation axioms is not allowed for the simple fact that in many cases you need to reference that same object multiple times. Creating an unnamed object makes this impossible. As can be seen above, after creating a virtual semantic object, one can initialize it through its **value** method.

Fourth, you will notice the type **void**. This is a new data type introduced by PENNY. An object declared of type **void**, such as,

```

     $\wedge$ share : void;

```

will not generate any code when being compiled. This can help to speed up the reasoning process in the *Mediator*.

Finally, there is a restriction pertaining to the names of the elevated semantic objects. The name of the elevated semantic object in the semantic relation (the name preceded by \sim) must match a name in the external relation. This is how the mapping is defined for data elements from the external source to the elevated source.

Chapter 3

Penny Compiler

There are several issues in translating deductive object-oriented languages which are related to the object-oriented aspects of non-monotonic inheritance and method resolution. Non-monotonic inheritance means that the declaration or definition can be overridden in a subtype. Method resolution is the process of determining which implementation of a method to choose when non-monotonic inheritance is in effect. Another major hurdle is the issue of multiple inheritance. Multiple inheritance introduces a substantial amount of ambiguity because there needs to be a way to choose among multiple method definitions for overriding.

The PENNY compiler attempts to deal with some of these factors in a fashion suitable for the Context Mediator [Goh96, BLGea97]. Furthermore, the PENNY compiler serves as a prototype for the ideas and techniques presented in [ALUW93, DT94]¹.

3.1 Related Works

There have been some efforts in taking a deductive object-oriented language to a deductive framework. Although it may appear as though this approach requires more effort, there are several advantages to taking this path. First you can use a standard evaluator of Datalog with negation. Second, the theoretical background of deductive databases is far more consolidated than for object-oriented databases [ALUW93]. For instance, the various semantics for negation can be used directly [ALUW93].

Although not explicitly discussed, the following are additional works and efforts [DT95, DT94, Law93, Lef93] that had some influence over the design of the compiler.

The first is the Chimera project. This project focuses on only a subset of the Chimera language [BBC97]. In this subset, a variety of object-oriented aspects are supported. They support unique object identifiers and class hierarchies. They support both single and multiple inheritance and allow methods to be extensionally or intensionally defined. Methods can be either single or multi-valued. Although the system supports non-monotonic single inheritance, it imposes constraints on multiple inheritance. Namely, a class may not inherit a method with the same name from multiple super-classes. When such a problem does exist, the compiler chooses the method of the first superclass specified. Furthermore, the compiler supports late binding of objects and classes. They perform late binding only on the most specific class of the object. One thing that was not mentioned in [BBC97], but seems reasonable to assume is that they had enforced strict typing in their compiler.

¹As far as the author is aware, there do not exist any prototypes for these two research efforts

The second, [DG89], is a stand alone effort to show that with simple translations of Prolog or Datalog, one can incorporate object-oriented capabilities. Their system, while it does demonstrate solid, simple translations, it does have a problem with respect to our architecture. The translations are done into Prolog which has operational semantics. These operational semantics along with the cuts place an extra constraint on the generated code, one being the ordering of predicates.

The third work is [GMR96] where the goal of the research is to apply semantic query optimizations to object databases. The basic modeling primitive is an object which is identified through a unique object identifier. The behavior of objects is defined via methods that can be executed on the object type. Furthermore, there exist classes that can be organized into a hierarchy. Inheritance is supported in this model, although in the compiler, only monotonic inheritance is supported. In the translation, classes and methods are represented as a relation whereas facts about the hierarchy, object identity, and inheritance are expressed as integrity constraints.

The final work in [ALUW93] is a stand alone effort to show how classical Datalog semantics could be used very simply to provide semantics to an extension of Datalog with classes, methods, inheritance, overriding, and late binding. The authors consider several approaches, ranging from static resolution typical in C++ to dynamic resolution for late binding [ALUW93]. This research effort uses an extended version of Datalog with negation that supports classes, methods, and inheritance. There are two types of inheritance presented. One is static inheritance where the resolution is done at compile time and depends only on the base class and the method name. The second is dynamic inheritance which is method inheritance defined with respect to the applicability of methods and not only class membership and method name. Both monotonic and non-monotonic inheritance are supported. This approach as in [DG89] also offers the ability to use standard Datalog optimizations (e.g. magic sets) to further enhance the quality of the output.

3.2 Global Picture

The PENNY compiler is used to compile the domain model, context, conversion functions, and elevation information into a format that the *Context Mediator* can evaluate [BFP⁺97a, BFP⁺97b]. Figure 3-1 illustrates where this compiler fits in the global scheme of things. As described in Section 2.1, every source has a set of elevation axioms and context axioms, defining what and how the information in that source should be understood. The system itself has a domain model which describes the units of information the *Context Mediator* will be evaluating and a library of conversion functions to map values from one context to another. Furthermore, the user of the system has his own set of context axioms, defining how he wishes the returned data to be represented. All of this information is compiled by the PENNY compiler before going to the *Context Mediator*. Currently, the compilation occurs only once. When the *Mediator* starts up, it loads the compiled axioms into main memory.

3.3 Implementation

In this section our discussion focuses on the implementation and design decisions of the compiler. The PENNY compiler stems from my previous work and implementation of the COINL compiler. Both of the compilers, with the exception of the grammar and some

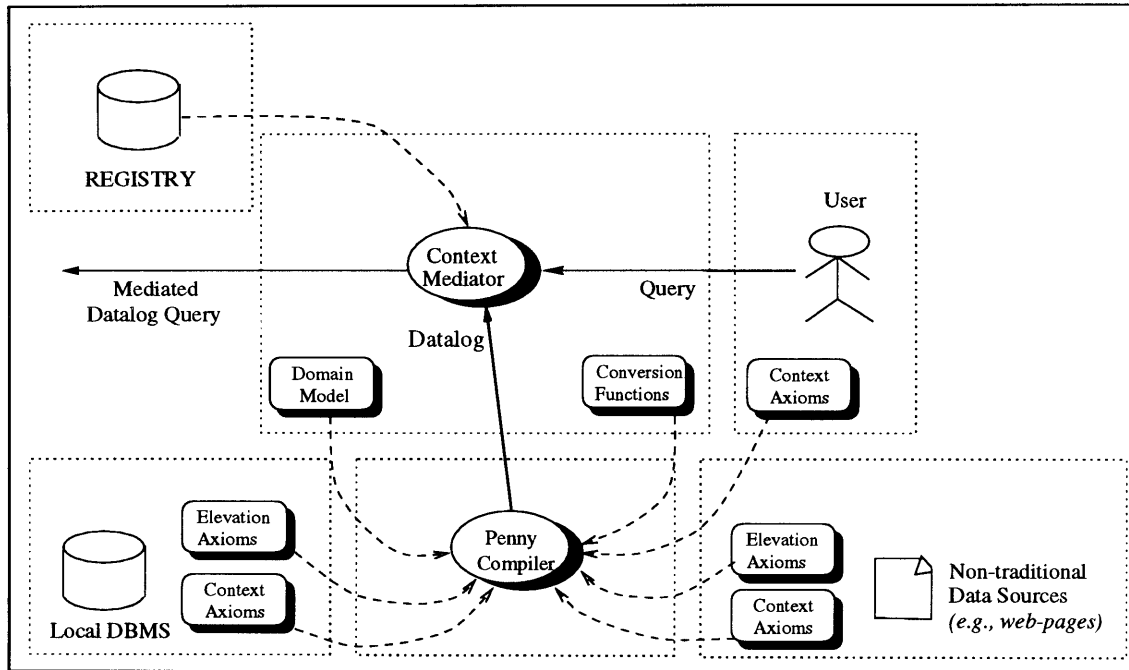


Figure 3-1: PENNY Compiler in the COIN Architecture

extensions to the intermediate language, are identical. As a matter of fact, the PENNY compiler was built from my existing COINL compiler code. For this reason, I have opted to only talk about the PENNY compiler.

The PENNY compiler is written completely in Prolog using the *ECLiPSe*² platform. There were several reasons why Prolog was chosen as the language of design. First, the ability to write static meta-programs to analyze, transform, and interpret other programs is particularly easy in Prolog because of the equivalence of programs and data: both are Prolog terms. Second because Prolog originated from attempts to use logic to express grammar rules and to formalize the process of parsing, it supports a number of facilities for parsing context-free grammars [SS94]. In particular, Prolog implements definite clause grammars (DCG), which are a generalization of context-free grammars that are executable, augmented by the language features of Prolog. They are equivalent to the Unix compiler-compiler tools, Yacc and Bison. One advantage to using the DCG's in Prolog, though, is that they come integrated into the environment. This is very convenient because it saves us the trouble of trying to talk across multiple, distinct platforms each having their own protocols. Third, the garbage collection in the Prolog system makes it convenient to work with large data structures and avoid the common programming errors in memory management that usually tend to occur in other languages such as C or C++. At this point, it is valid to evaluate both Scheme and Perl as both of these languages also employ garbage collection. The reason neither of these languages were chosen is because the author was not aware of any existing compiler-compiler tools available under those environments. Finally, because

²ECLiPSe: The ECRC Constraint Logic Parallel System. More information can be obtained at <http://www.ecrc.de/eclipse/>.

PENNY is a first order logic language, it made sense to evaluate it under the logic paradigm, thereby simplifying the tokenization and any symbolic manipulation needed in processing the transformations.

The PENNY compiler follows the methodology of compiler construction given in [ASU86]. It is composed of four modules: the lexer, the parser, the semantic checker, and the code generator. Currently there exists no optimizer module (although this shall be discussed more in depth in Section 4.1.2).

3.3.1 Lexical Analysis Module

The lexical analysis module is composed of 25 lines of *ECLiPSe* Prolog code. It uses the *ECLiPSe* builtin predicate `read_token/3` to extract tokens from the input stream. This does have the disadvantage that this routine will most likely have to be re-written when porting the compiler to a different Prolog environment. While on one hand this may be deemed as a problem, the `read_token` predicate did prevent the writing of a new lexer from scratch, a job which is both tedious and in my opinion, not very important with respect to the goal of this Thesis. Another aspect of the lexer has to do with how it reads tokens. The lexer will read the entire file in before passing the list of tokens back to the parser. Theoretically, this shouldn't cause a problem since we are in an environment with memory management and garbage collection, so if the token list gets too long, it should be swapped out to disk. The reason it reads the entire file is that the lexer is not flexible enough to recognize a chunk of tokens together. This is easily solved by rewriting the lexer as a finite state automaton where the different states would indicate what sort of information we were reading in. Then at certain states, once we knew that we had read enough information in to syntactically check it, we could return the chunk of tokens back to the parser. Once again, for convenience and time, it was deemed wiser to go with the builtin predicate and take the chances with the garbage collection.

There is an alternative to the approach taken here and that is to use the `op/3` predicate in Prolog to implement the lexer. We evaluate which tokens we want to treat as operators, give them a mapping to Prolog, and then use the normal Prolog `read/3` predicate to read in the Prolog terms. The problem with this method is that we have now tied our source language to our implementation language which is not only a serious design flaw, but can also cause enormous confusion.

3.3.2 Parser Module

The parser was one of the more complicated components of the compiler, for reasons dealing with file inclusion and error handling. The parser is implemented as a DCG grammar. It uses a list structure, known as the dictionary, to maintain the symbol table information and the intermediate code generated during the parsing phase. It was decided to combine both of the data structures into one since they both needed to be passed into the code generation module. The list used as the data structure has been abstracted away by a set manipulation routines. The intermediate code is embedded in the DCG itself. In retrospect, this was a poor design choice. A wiser implementation would have abstracted the generation of the intermediate language into a set of routines. This would have allowed us to change the representation of the intermediate language without touching the parser.

Rather than using a preprocessor to handle file inclusion, such as C or C++ compilers do, this functionality was employed inside the parser. Unlike a C or C++ include statement,

an include statement in PENNY (include statements are given by the keyword **use**) has a different meaning. When applied to regular axioms, such as elevation, context, or conversion axioms, then the semantics hold true. But when a user includes a file containing the domain model, the idea is for the domain model to be used as reference data, no actual code generation is performed. The reason for this is that the domain model would most likely have been compiled already so compiling it again would give duplicate code thus making the Mediator [BLGea97] slower. So upon registering a **use** statement, the parser will set a flag and call the tokenizer again with the new file to include and the current dictionary. During the parsing, if a domain model element is ever parsed and the include flag is set then no intermediate code is placed inside the dictionary. Even if a pre-processor had been used, the same scenario would have been encountered. It is conceivable that a file containing a domain model would included another file to augment its information. At this point, it would still be necessary to check whether the included file contained any domain model elements. This unfortunately implies that a domain model cannot include another file containing domain model information. An easy to fix to this problem, though, would be to introduce a new keyword into the language. Then whenever a user wants to include and compile another file they could use one keyword versus to when he wants just to use the file as auxiliary information.

The error handling in the parser is implemented via the phrase/3 predicate. The phrase/3 predicate takes in the list of tokens to parse and gives back any remaining tokens in the list. The phrase/3 predicate will backtrack to obtain all possible solutions, failing when it cannot parse any of the remaining tokens. The task here was to distinguish failure in the parser between failure due to incorrect tokens and failure due to the control nature of the DCG (remember that in logic programming, failure is an acceptable answer and furthermore is used as a control mechanism). To accomplish this task, the DCG rules needed to be re-written one level deeper. As an example, take the top level rule for parsing the domain model given by the following DCG rule (DCG rules resemble productions given in the PENNY BNF with the ::= replaced by \rightarrow).

Rule 1: $\langle \textit{Domain_Model} \rangle \rightarrow \langle \textit{Semantic_Decl} \rangle \langle \textit{Domain_Model} \rangle$

Rule 2: $\langle \textit{Domain_Model} \rangle \rightarrow \epsilon$

Now lets suppose that we have a list of tokens to parse and this is the first rule we encounter. There are three possibilities that result concerning the list of tokens:

- A. The list contains valid tokens of domain model elements
- B. The list contains some valid domain model tokens but somewhere in the token list lies an error and without any loss of generality, we can assume that the error lies somewhere in the middle of the token list
- C. The list does not contain any tokens corresponding to any domain model element, rather it is a list of valid tokens of another PENNY construct. Without any loss of generality, we can assume that the list contains valid tokens for context axioms

Let us start with case A. The Prolog system will hit upon Rule 1 first and evaluate it. It will recursively call Rule 1 (remember that in case A our entire list consists of valid domain model tokens) until the token list has been exhausted. On the next recursive call, Rule 1 will fail (the token list is empty at this point) and the Prolog system will choose Rule 2 to

evaluate next. Upon evaluating Rule 2, the rule evaluates to true, and the entire parsing comes to a halt, since we have already exhausted our token list and our predicate evaluated to true. At this point, the token list has been successfully parsed and the remaining set of tokens is the empty set.

Now consider case B. The Prolog system will once again encounter Rule 1 first and evaluate it. Only this time, because there lies an error in the token list, at some point during parsing the tokens, Rule 1 will fail. When this happens, the Prolog system will again choose Rule 2 to evaluate, in which case will evaluate to true, and again the parsing will halt. But remember that the error in the token list lay somewhere in the middle, and as such, when Rule 1 failed, there was a set of tokens remaining that could not be parsed. Thus when the parsing completes, what remains in the end is a set of tokens which could not be parsed, i.e. the tokens with the error in it. In this case, things worked out well, since the Prolog system parsed as much as possible and returned the remaining tokens (where the error lay) back to the user.

Finally, let us consider case C. The first time the Prolog system evaluates Rule 1, it will fail. The reason is that the tokens in the list correspond to context axioms, not the domain model. But nonetheless, the Prolog system evaluates the first rule it encounters and thus will fail. At this point, the Prolog system will evaluate Rule 2 and succeed, thus ending the parsing. Unfortunately, since the parsing has halted, the Prolog system did not get a chance to evaluate any other rules. As a result, what we are left with at the end is the full list of tokens. Obviously, this is a problem. Although the parser will tell us that we have remaining tokens, this is technically incorrect. We only have remaining tokens because at the time of failure, the parser could not distinguish between failing because an error in the domain model and failing because of control (here the control would be to fail at the domain model rule and have the Prolog system backtrack and choose another rule). So to get around this problem, the DCG was changed as follows:

Rule 1: $\langle \textit{Domain_Model} \rangle \longrightarrow \langle \textit{Semantic_Decl} \rangle \langle \textit{More_Domain_Model} \rangle$

Rule 2: $\langle \textit{More_Domain_Model} \rangle \longrightarrow \langle \textit{Domain_Model} \rangle$

Rule 3: $\langle \textit{More_Domain_Model} \rangle \longrightarrow \epsilon$

This time, instead of the recursion happening at the top level rule, it has been pushed down one level deeper. Now it should be clear that cases A and B work as before. This time, though, when we are in case C, what happens is as follows. When the Prolog system evaluates Rule 1, it will immediately fail, since once again the tokens in our list correspond to context axioms. This time though, there are no other rules with the same head that the Prolog system can evaluate, thus it will fail. At this point, the Prolog system will backtrack and proceed onto its next choice-point. Eventually, the Prolog system should reach the correct rules defining context axioms and successfully parse the entire token list, leaving us with the empty set as the remaining token set.

3.3.3 Semantics Module

The semantics module does very little at the moment. It does a few semantic checks, such as making sure **value** constructors take in valid context names. This is one module that definitely needs to be worked on more in the future. Because of time constraints both with respect to the language and compiler, this module has had to suffer in terms of implementation.

3.3.4 Code Generator Module

The compiler module takes in the list structure that the parser produces and iterates over the intermediate code. The main predicate is `generate_code/5` where the intermediate language is unified with the appropriate rule. The code generator takes care of doing the overriding and producing the **value** constructors for the semantic objects defined in the elevation axioms. This design was chosen for its simplicity both in the implementation and in its understanding.

There does exist an alternative, though. Since it is known priori what the generated code will look like, one can create a new DCG to parse this generated code. This DCG will take the generated output and map it to a dictionary data structure, a data structure identical to the one used in parsing PENNY. If the implementation of this new DCG is free of any side effects, then what will result is a parser from the desired output code to a data structure. The beauty in this is that using the logic capabilities inherent in Prolog, the new DCG should be symmetric. That is feed in the language and derive the data structure, or feed in the data structure and derive the language. Because of this duality, after parsing PENNY it is conceivable to pass the data structure to this new DCG thereby deriving the desired output language (remember that the idea was that both data structures were identical).

There are two issues which makes this alternative complicated, if not, infeasible. First, getting the language totally free of side effects and coding it to ensure the symmetry is not necessarily a straightforward task. Second, because the desired output has been augmented with extra information, such as overriding and the generation of **value** constructors, it was not clear how difficult this would have been to accomplish. Overall the extra work in making the Prolog system behave as described didn't seem to offer many advantages, except for the fact that the compiler would be exploiting the full capabilities of Prolog.

Chapter 4

Conclusion and Future Work

4.1 Limitations

Here we focus our discussion on the limitations and problems found in PENNY and its compiler and end this Thesis with a summary of contributions.

4.1.1 Penny

While it was hopefully clear from Section 2.2.1, that PENNY does indeed provide a better abstraction mechanism and cleaner design than that offered by COINL, it should be noted that the number of users who have programmed in PENNY is relatively small. Since the entire goal of the new language was aimed at removing extra complexity and burden normally placed on the user, it is difficult to ascertain whether this task has been fulfilled. There needs to be more rigorous use of the language to evaluate its constructs and design.

A ramification of the decrease in complexity in the language has been a decrease in the flexibility of the language. This is because PENNY enforces stricter semantics than COINL, thus there has been some expressiveness lost in the process. It is inconclusive at this point whether this eliminated expressiveness will affect the way users program in MINT. There remains a deal more of study to be done in this direction.

One feature that PENNY (as well as did COINL) does lack is support for registry population. Although not explicitly mentioned in this Thesis, the registry plays an important role in our system. It defines the export schemas of the relations used in the architecture. These export schemas are then elevated into “virtual” relations through the elevation axioms. The design of this part of the compilation system is crucial because this is where the connection to the external sources is made in the MINT architecture; any mismatch can cause serious problems in the run-time of the entire system. Now it seems perfectly reasonable that the compiler, given a little more information, can generate this registry information from the elevation axioms. This would be a definite advantage since it would eliminate the cross-referencing that needs to be done between the elevation axioms and registry information, therefore minimizing user errors.

4.1.2 Penny Compiler

As described in Section 3.3, there is a lot more work that can be done on the compiler.

First, better error handling routines can be coded into the compiler. Although currently, the parser will return the invalid tokens, it is up to the user to explicitly find the error. It

would be much nicer if, say, line number information and the exact error was generated by the parser. This would most likely lead to faster user debugging sessions. In addition to a more robust error handling capability, the parser could also be modified to eliminate the unnecessary backtracking it performs. Currently it works by reading a list of tokens and trying to parse them. It could be modified so that instead of using a list, it would use an input stream and extract tokens from the stream. Because the language is deterministic, extracting tokens from an input stream would reduce the number of times the system would try to backtrack. Furthermore this could in turn lead to a better error handling management since at any one point when the system would fail, it would be easier to determine whether it was due to an error in the input stream or it was part of the control mechanism. The disadvantage is that the DCG in the parser would need to be modified carefully to take these two cases into account. As for the rest of the system, no other module unit returns any of type of error information which is also a problem. It would be ideal for the semantics checker and the code generator to return useful information messages back to the user while compiling. Furthermore, while speaking of the semantics checker, a more robust and complete implementation awaits to be constructed.

Second, there is a lack of performance analysis and optimizations that can be performed on the generated code. Ideally, this could be accomplished by first coding the desired code by hand into the *Mediator* and then running performance analysis on the results. This will give a good indication where and what optimizations should be placed in the compiler.

Last, support for both context inheritance and multiple inheritance needs to be investigated. The research efforts in [DT95] is but one work which focuses on multiple inheritance. In terms of context inheritance, problems dealing with semantics need to be ironed out.

4.2 Conclusion

In closing, PENNY and the PENNY compiler are two invaluable tools for programming in MINT. The PENNY compiler has built on previous work in the area of deductive object-oriented translations whereas PENNY has its basis in COINL. Albeit needing more work, both PENNY and the PENNY compiler provide a basis upon which further research and extensions can be made to the COIN model.

Appendix A

Penny BNF

This section describes the full BNF for PENNY and the list of reserved keywords in the language. But before that, the notational convention used here to represent the PENNY BNF is given.

A.1 Notational Convention

1. Non-terminals are enclosed in angle brackets, e.g. *< Example >*
2. The following are terminals:
 - Character strings regardless of case
 - Punctuation symbols such as parentheses, commas, brackets, etc
 - Operator symbols such as +, -, *, /, =, >, etc
 - The digits 0...9
3. The symbol ϵ is used to denote the empty string
4. Productions are given by: $\mathcal{X} ::= \alpha$, where \mathcal{X} is a non-terminal and α denotes grammar symbols, either non-terminals or terminals
5. The left side of the first production is the start symbol
6. The vertical bar (|) is used to separate mutually exclusive options.
7. The percent symbol (%) symbol is used for commenting in the BNF
8. Variables names must begin with an uppercase letter
9. Boldfacing signifies PENNY keywords

A.2 Reserved Penny Keywords

- **as**
- **attribute**
- **context**

- ctx
- cvt
- elevate
- end
- in
- modifier
- not
- over
- self
- semanticType
- source
- target
- use
- value
- view
- void

A.3 BNF

```
%%
%% This is the START production
%%
< Penny_Rule > ::= < Top_Level_Rules >
```

```
%%
%% These are the rules which can exist at the top level of a program
%%
< Top_Level_Rules > ::= < Include_Directive > < Top_Level_Rules >
                        | < Domain_Model >
                        | < Context_Module_Def >
                        | < Elevation_View_Axioms >
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% DOMAIN MODEL
%%
```



```

< Domain_Model > ::= < Semantic_Decl > < Domain_Model > | ε

< Semantic_Decl > ::= semanticType < Subtype_Decl > { < Att_Mod_Decls > };
< Subtype_Decl > ::= < ident > :: < ident >

< Att_Mod_Decls > ::= [ < Att_Decl > | < Mod_Decl > ] < Att_Mod_Decls > | ε

< Att_Decl > ::= attribute < type > < ident > ;
< Mod_Decl > ::= modifier < type > < ident > (ctx) ;

%%
%% END DOMAIN MODEL
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% MODULES
%%

< Context_Module_Def > ::= < Context_Begin >
                        < Context_Conversion_Axioms_Def >
                        < Context_End >

< Context_Begin > ::= context < ident > ;
< Context_End > ::= end < ident >

< Context_Conversion_Axioms_Def > ::= < Context_Axioms >
                                     | < Conversion_Axioms >
                                     | < Include_Directive >

%%
%% END MODULES
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% CONTEXT AXIOMS
%%

< Context_Axioms > ::= < Context_Axioms_Decl >
                    < Context_Conversion_Axioms_Def > | ε

< Context_Axioms_Decl > ::= < ident > < Type_Binding > = < Virtual_Constructor > ;
                        | < ident > < Type_Binding > = < Virtual_Constructor >
                        < Rule_Definition > ;

< Virtual_Constructor > ::= ~([ < constant > | $ ])

```

```

%%
%% END CONTEXT AXIOMS
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% CONVERSION AXIOMS
%%

< Conversion_Axioms > ::= < Conversion_Axioms_Decl >
                        < Context_Conversion_Axioms_Def > | ε

< Conversion_Axioms_Decl > ::= < Cvt > < Type_Binding >
                              < Rule_Definition > ;

< Cvt > ::= cvt < Argument_Term >

%%
%% END CONVERSION AXIOMS
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% ELEVATION AXIOMS
%%

< Elevation_View_Axioms > ::= [ < View_Axioms > | < Elevation_Axioms > ]
                              < Elevation_View_Axioms >

< View_Axioms > ::= view < P_Term > over < View_Over > ;

< View_Over > ::= < P_Term > < View_Over2 >
< View_Over2 > ::= , < View_Over > | ε

< Elevation_Axioms > ::= elevate < Source_Relation >
                        in < ident >
                        as < Elevated_Relation >
                        { < Elevated_Attributes > } ;

< Source_Relation > ::= < P_Term >

< Elevated_Relation > ::= < ident > ( < Elevated_Arguments > )

< Elevated_Arguments > ::= < Earg > < Elevated_Arguments2 >
< Elevated_Arguments2 > ::= , < Elevated_Arguments > | ε

< Earg > ::= < elevated_ident > : < ident > | < elevated_ident > : void

```

```

< Elevated_Attributes > ::= < Term > = < Rule_Def_Value > ;
                                < Elevated_Attributes > | ε

%%
%% END ELEVATION AXIOMS
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% PENNY EXPRESSIONS
%%

< Rule_Def_Value > ::= < Term > | $ < Rule_Definition >

< Rule_Definition > ::= < Rule_Definition_Op > < Rule_Expressions >

< Rule_Expressions > ::= < E_Term > < Rule_Expressions2 >
< Rule_Expressions2 > ::= , < Rule_Expressions > | ε

< E_Term > ::= < Term > < E_Term1 >
              | < P_Term >
              | not < E_Term >
              | < ident > : < ident >

< E_Term1 > ::= < booleanOp > < Term > | < booleanOp > < Math_Expression >

< Math_Expression > ::= < Term > < binaryOp > < Term >

< Term > ::= < Virtual_Term >
           | < Elevated_Term >
           | < Attribute_Term >
           | < Modifier_Term >
           | < Singleton_Term >

< Virtual_Term > ::= < virtual_ident >.< Method >
< Elevated_Term > ::= < elevated_ident >.< Method >
< Attribute_Term > ::= < Object >.< AttModCvt >
< Modifier_Term > ::= < Object >.< AttModCvt >(< Context >)

< Singleton_Term > ::= < elevated_ident >
                    | < virtual_ident >
                    | < sourcerel_ident >
                    | < constant >
                    | < ident >

< Context > ::= source | target | < ident >
< Object > ::= self | < ident >
< Method > ::= value | < ident >
< AttModCvt > ::= < Method > | < Cvt >

```

```

%%
%% END PENNY EXPRESSIONS
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% AUXILLARY RULES
%%

< Include_Directive > ::= use([< atom > | < string >]) ;

< Type_Binding > ::= '< ident >''

< Rule_Definition_Op > ::= <-

< virtual_ident > ::= ~< ident >
< elevated_ident > ::= ^< ident >
< sourcerel_ident > ::= @< ident >

< Argument_Term > ::= (< Id_Term_List >)

< Id_Term_List > ::= [< Id_Term_List >
                    | < Id_Term > < Id_Term_List2 >
                    | ε

< Id_Term_List2 > ::= , < Id_Term_List > | ε

< P_Term > ::= < Id_Term >

< Id_Term > ::= < ident > < Argument_Term > | ε

< booleanOp > ::= < | > | < > | >= | =< | =
< binaryOp > ::= < booleanOp > | + | - | * | /

< ident > ::= < atom > | < string > | < variable >
< type > ::= < ident >

< atom > ::= [a - z]+[_A - Z a - z0 - 9]* | '[A - Z a - z0 - 9]+'
< string > ::= "[_A - Z a - z0 - 9]+"
< variable > ::= [A - Z_]+[_A - Z a - z0 - 9]*

< constant > ::= integer | real | < atom > | < string >

%%
%% END AUXILLARY RULES
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Appendix B

TASC Penny Axioms

The following code is the implementation of the TASC application which is currently running in our system. As described in Section 1.1, the TASC application focuses on integrating various information sources containing financial figures for a variety of companies, foreign and domestic. The DStreamAF context, has also been included although no reference was made to it earlier. There are some additional relations which are used for auxiliary information and name mappings.

The relation `Name_map_Ds_Ws` contains name mappings between company names in DiscAF and WorldAF. `Name_map_Dt_Ds` holds name mappings between company names in DStreamAF and DiscAF. Finally, to add closure to the name mapping routines, the table `Name_map_Dt_Ws` has the name mappings between company names in DStreamAF and WorldAF.

The relation `Currencytypes` converts between a given country and its official currency while `Currency_map` maps mnemonic currency values, e.g. “USD” to “US” and vice versa.

B.1 Domain Model

```
semanticType companyFinancials::number {
  attribute companyName company;
  attribute date fyEnding;

  modifier number scaleFactor(ctx);
  modifier currencyType currency(ctx);
};

semanticType companyName::number {
  modifier string format(ctx);
  attribute string countryIncorp;
};

semanticType exchangeRate::number {
  attribute currencyType fromCur;
  attribute currencyType toCur;
  attribute date txnDate;
};
```

```

semanticType date::string {
  modifier string dateFmt(ctx);
};

semanticType currencyType::string {
  modifier string curTypeSym(ctx);
};

semanticType countryName::string {
  attribute currencyType officialCurrency;
};

```

B.2 Context Axioms

B.2.1 DiscAF

```

use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

context c_ds;

scaleFactor<companyFinancials> = ~(1);

currency<companyFinancials> = ~($) <-
  Comp = self.company,
  Country = Comp.countryIncorp,
  CurrencyType = Country.officialCurrency,
  $ = CurrencyType.value;

format<companyName> = ~("ds_name");

dateFmt<date> = ~("American Style /");

curTypeSym<currencyType> = ~("3char");

end c_ds;

```

B.2.2 WorldAF

```

use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

context c_ws;

scaleFactor<companyFinancials> = ~(1000);

currency<companyFinancials> = ~('USD');

```

```

format<companyName> = ~("ws_name");

dateFmt<date> = ~("American Style /");

curTypeSym<currencyType> = ~("3char");

end c_ws;

```

B.2.3 DStreamAF

```

use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

context c_dt;

scaleFactor<companyFinancials> = ~(1000);

currency<companyFinancials> = ~($) <-
    Comp = self.company,
    Country = Comp.countryIncorp,
    CurrencyType = Country.officialCurrency,
    $ = CurrencyType.value;

format<companyName> = ~("dt_name");

dateFmt<date> = ~("European Style -");

curTypeSym<currencyType> = ~("2char");

end c_dt;

```

B.2.4 Olsen

```

use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

context c_ol;

dateFmt<date> = ~("European Style /");

end c_ol;

```

B.3 Conversion Functions

```

use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

context c0;

%%-----
%% conversion functions for companyFinancials

```

```

%%-----
cvt()<companyFinancials> <-
  U = self.value,
  W = self.cvt(scaleFactor, U),
  $ = self.cvt(currency, W);

%%-----
%% conversion functions for companyFinancials w.r.t. scaleFactors
%%-----
cvt(scaleFactor, Val)<companyFinancials> <-
  SrcMod = self.scaleFactor(source),
  TgtMod = self.scaleFactor(target),
  SrcMod.value = TgtMod.value,
  $ = Val;

cvt(scaleFactor, Val)<companyFinancials> <-
  FsvMod = self.scaleFactor(source),
  FtvMod = self.scaleFactor(target),
  Fsv = FsvMod.value,
  Ftv = FtvMod.value,
  Fsv <> Ftv,
  Ratio = Fsv / Ftv,
  $ = Val * Ratio;

cvt(currency, Val)<companyFinancials> <-
  SrcMod = self.currency(source),
  TgtMod = self.currency(target),
  SrcMod.value = TgtMod.value,
  $= Val;

cvt(currency, Val)<companyFinancials> <-
  SrcMod = self.currency(source),
  TgtMod = self.currency(target),
  SrcMod.value <> TgtMod.value,
  FyDate = self.fyEnding,
  olsen_p(FromCur, ToCur, Rate, TxnDate),
  SrcMod.value = FromCur.value,
  TgtMod.value = ToCur.value,
  FyDate.value = TxnDate.value,
  $ = Val * Rate.value;

%%-----
%% conversion functions for companyName
%%-----
cvt()<companyName> <-
  SrcMod = self.format(source),
  TgtMod = self.format(target),
  SrcMod.value = TgtMod.value,

```



```

    $ = self.value;

cvt(<companyName> <-
  SrcMod = self.format(source),
  TgtMod = self.format(target),
  SrcMod.value = "ds_name",
  TgtMod.value = "ws_name",
  'Name_map_Ds_Ws_p'(DsName, WsName),
  self.value = DsName.value,
  $ = WsName.value;

cvt(<companyName> <-
  SrcMod = self.format(source),
  TgtMod = self.format(target),
  SrcMod.value = "ws_name",
  TgtMod.value = "ds_name",
  'Name_map_Ds_Ws_p'(DsName, WsName),
  self.value = WsName.value,
  $ = DsName.value;

%%-----
%% conversion functions for dateFmts
%%-----

cvt(<date> <-
  SrcMod = self.dateFmt(source),
  TgtMod = self.dateFmt(target),
  SrcMod.value = TgtMod.value,
  $ = self.value;

cvt(<date> <-
  SrcMod = self.dateFmt(source),
  TgtMod = self.dateFmt(target),
  SrcFormat = SrcMod.value,
  TgtFormat = TgtMod.value,
  SrcFormat <> TgtFormat,
  SrcVal = self.value,
  datexform($, SrcVal, SrcFormat, TgtFormat);

%%-----
%% conversion functions for currency symbols.
%%-----

cvt(<currencyType> <-
  SrcMod = self.curTypeSym(source),
  TgtMod = self.curTypeSym(target),
  SrcMod.value = TgtMod.value,
  $ = self.value;

cvt(<currencyType> <-

```

```

    SrcMod = self.curTypeSym(source),
    TgtMod = self.curTypeSym(target),
    SrcMod.value = "3char",
    TgtMod.value = "2char",
    'Currency_map_p'(Char3, Char2),
    self.value = Char3.value,
    $ = Char2.value;

cvt()<currencyType> <-
    SrcMod = self.curTypeSym(source),
    TgtMod = self.curTypeSym(target),
    SrcMod.value = "2char",
    TgtMod.value = "3char",
    'Currency_map_p'(Char3, Char2),
    self.value = Char2.value,
    $ = Char3.value;

%%-----
%% conversion functions for number, a simple value
%% cvt function for number is overridden by the following:
%%   * companyFinancials
%%-----
cvt()<number> <-
    $ = self.value;

%%-----
%% conversion functions for strings, a simple value
%% cvt function for string is overridden by the following:
%%   * companyName
%%   * date
%%-----
cvt()<string> <-
    $ = self.value;

end c0;

```

B.4 Elevation Axioms

```

%%-----
%% Disclosure:DiscAF
%%-----
elevate 'DiscAF'(cname, fyEnding, shares, income, sales, assets, incorp)
in c_ds
as 'DiscAF_p'(^cname : companyName,    ^fyEnding : date, ^shares : void,
              ^income: companyFinancials, ^sales : companyFinancials,
              ^assets: companyFinancials, ^incorp: countryName)

```

```

{
  ^cname.countryIncorp = ^incorp;

  ^income.company = ^cname;
  ^income.fyEnding = ^fyEnding;

  ^sales.company = ^cname;
  ^sales.fyEnding = ^fyEnding;

  ^assets.company = ^cname;
  ^assets.fyEnding = ^fyEnding;

  ^incorp.officialCurrency = ~curType;

  ~curType.value = $ <-
    ~curType = Incorp.officialCurrency,
    Y = Incorp.value,
    'Currencytypes'(Y, $);
};

%%-----
%% Worldscope:WorldAF (Keys: CompanyName and Date)
%%-----
elevate 'WorldAF'(cname, fyEnding, shares, income, sales, assets, incorp)
in c_ws
as 'WorldAF_p'(^cname : companyName, ^fyEnding : date, ^shares : number,
  ^income: companyFinancials, ^sales : companyFinancials,
  ^assets: companyFinancials, ^incorp: countryName)
{
  ^cname.countryIncorp = ^incorp;

  ^income.company = ^cname;
  ^income.fyEnding = ^fyEnding;

  ^sales.company = ^cname;
  ^sales.fyEnding = ^fyEnding;

  ^assets.company = ^cname;
  ^assets.fyEnding = ^fyEnding;

  ^incorp.officialCurrency = ~curType;

  ~curType.value = $ <-
    ~curType = Incorp.officialCurrency,
    Y = Incorp.value,
    'Currencytypes'(Y, $);
};

```

```

%%-----
%% Olsen (Keys: Source Currency, Target Currency, and Date)
%%-----
elevate olsen(exchanged, expressed, rate, date)
in c_ol
as olsen_p(^exchanged : currencyType, ^expressed : currencyType,
          ^rate       : exchangeRate, ^date       : date)
{
    ^rate.fromCur = ^expressed;
    ^rate.toCur   = ^exchanged;
    ^rate.txnDate  = ^date;
};

%%-----
%% Datastream: DStreamAF (Keys: As_of_date and Name)
%%-----
elevate 'DStreamAF'(date, name, total_sales, total_items_pre_tax,
                  earned_for_ordinary, currency)
in c_dt
as 'DStreamAF_p'(^date           : date,
                ^name            : companyName,
                ^total_sales     : companyFinancials,
                ^total_items_pre_tax : companyFinancials,
                ^earned_for_ordinary : companyFinancials,
                ^currency        : currencyType)
{
    ^name.countryIncorp = ~incorp;

    ^total_sales.fyEnding = ^date;
    ^total_sales.company = ^name;

    ^total_items_pre_tax.fyEnding = ^date;
    ^total_items_pre_tax.company = ^name;

    ^earned_for_ordinary.fyEnding = ^date;
    ^earned_for_ordinary.company = ^name;

    ~incorp.officialCurrency = ^currency;
    ~incorp.value = $ <-
        Currency = ~incorp.officialCurrency,
        Two = Currency.value,
        'Currencytypes_p'(Country, Three),
        Two = Three.value,
        $ = Country.value;
};

%%-----

```

```
%% Auxillary Tables
```

```
%%-----
```

```
elevate 'Currencytypes'(country, currency)
in c_ds
as 'Currencytypes_p'(^country : string, ^currency : string) {};
```

```
elevate 'Currency_map'(cur3, cur2)
in c_ds
as 'Currency_map_p'(^cur3 : string, ^cur2 : string) {};
```

```
elevate 'Name_map_Ds_Ws'(dsNames, wsNames)
in c_ds
as 'Name_map_Ds_Ws_p'(^dsNames : string, ^wsNames : string) {};
```

```
elevate 'Name_map_Dt_Ds'(dtName, dsName)
in c_dt
as 'Name_map_Dt_Ds_p'(^dtName : string, ^dsName : string) {};
```

```
elevate 'Name_map_Dt_Ws'(dtName, wsName)
in c_dt
as 'Name_map_Dt_Ws_p'(^dtName : string, ^wsName : string) {};
```

Appendix C

DLA Penny Axioms

The following PENNY code is the implementation of the DLA application which is also running as a demonstration in our system. Although not mentioned in this Thesis, the code was provided to illustrate another example of programming in PENNY. This application focuses on integrating information on medical supplies from various manufacturers, distributors, and the defense logistic agency, hence the name DLA. There are three distributors in the system, Allied_Health_Corp, King_Medical_Supplies, and BMI. There also exists a relation containing information from the defense logistic agency (DLA). The information in the table DLA also houses the information on the various manufacturers.

In addition to the main relations, there exist a variety of auxiliary tables to aid in any needed conversions. The relations Dist_Man, Dist_Dla, and Man_Dla map between the distributor part number and manufacturer part number, the distributor part number and the national stock number, and the manufacturer part number and the national stock number respectively.

Finally the last two relations, Man_Unit_Map and DistDla_Unit_Map convert between the units of manufacture and the units of the DLA and the distributor units and the units of the DLA, respectively.

C.1 Domain Model

```
semanticType part_id::string {
    modifier string per_unit_type(ctx);
};
```

```
semanticType price::string {
    attribute part_id part;
    attribute string distributor;
    modifier string preferred_id(ctx);
};
```

C.2 Context Axioms

C.2.1 Distributor

```
use('/home/tpena/work/coinlc/Penny/examples/darpa/penny/dm0.pny');

context c_dist;

preferred_id<part_id> = ~("Distributor Part #");

per_unit_type<price> = ~("Sale Unit");

end c_dist;
```

C.2.2 Manufacturer

```
use('/home/tpena/work/coinlc/Penny/examples/darpa/penny/dm0.pny');

context c_man;

preferred_id<part_id> = ~("Manufacturer Part #");

per_unit_type<price> = ~("Manufacture Unit");

end c_man;
```

C.2.3 Olsen

```
use('/home/tpena/work/coinlc/Penny/examples/darpa/penny/dm0.pny');

context c_dla;

preferred_id<part_id> = ~("NSN");

per_unit_type<price> = ~("Basic Unit");

end c_dla;
```

C.3 Conversion Functions

```
use('/home/tpena/work/coinlc/Penny/examples/darpa/penny/dm0.pny');

context c0;
%%-----
%% conversion functions w.r.t. price
%%-----
cvt()<price> <-
    SrcMod = self.per_unit_type(source),
    TgtMod = self.per_unit_type(target),
```

```
SrcMod.value = TgtMod.value,  
$ = self.value;
```

```
cvt()<price> <-  
  SrcMod = self.per_unit_type(source),  
  TgtMod = self.per_unit_type(target),  
  SrcMod.value = "Sale Unit",  
  TgtMod.value = "Manufacture Unit",  
  SrcPrice = self.value,  
  Part = self.part,  
  PartValue = Part.value(c_dist),  
  Name1 = self.distributor,  
  'Dist_Unit_Map_p'(Dist, Name2, Ratio),  
  Name1.value = Name2.value,  
  PartValue = Dict.value,  
  $ = SrcPrice / Ratio.value;
```

```
cvt()<price> <-  
  SrcMod = self.per_unit_type(source),  
  TgtMod = self.per_unit_type(target),  
  SrcMod.value = "Manufacture Unit",  
  TgtMod.value = "Sale Unit",  
  SrcPrice = self.value,  
  Part = self.part,  
  Name1 = self.distributor,  
  PartValue = Part.value(c_man),  
  'Dist_Man_p'(Dist1, Man1),  
  'Dist_Unit_Map_p'(Dist, Name2, Ratio),  
  Name1.value = Name2.value,  
  Dist1.value = Dist.value,  
  PartValue = Man1.value,  
  $ = SrcPrice * Ratio.value;
```

```
cvt()<price> <-  
  SrcMod = self.per_unit_type(source),  
  TgtMod = self.per_unit_type(target),  
  SrcMod.value = "Manufacture Unit",  
  TgtMod.value = "Basic Unit",  
  SrcPrice = self.value,  
  Part = self.part,  
  PartValue = Part.value(c_man),  
  'Man_Unit_Map_p'(Man, Count),  
  PartValue = Man.value,  
  $ = SrcPrice / Count.value;
```

```
cvt()<price> <-  
  SrcMod = self.per_unit_type(source),  
  TgtMod = self.per_unit_type(target),
```



```

SrcMod.value = "Basic Unit",
TgtMod.value = "Manufacture Unit",
SrcPrice = self.value,
Part = self.part,
PartValue = Part.value(c_dla),
'Man_Dla_p'(Man1, Dla),
'Man_Unit_Map_p'(Man, Count),
PartValue = Dla.value,
Man1.value = Man.value,
$ = SrcPrice * Count.value;

```

```

cvt(<price> <-
  SrcMod = self.per_unit_type(source),
  TgtMod = self.per_unit_type(target),
  SrcMod.value = "Sale Unit",
  TgtMod.value = "Basic Unit",
  SrcPrice = self.value,
  Part = self.part,
  Name1 = self.distributor,
  PartValue = Part.value(c_dist),
  'DistDla_Unit_Map_p'(Dist, Name2, RBasic),
  PartValue = Dist.value,
  Name1.value = Name2.value,
  $ = SrcPrice / RBasic.value;

```

```

cvt(<price> <-
  SrcMod = self.per_unit_type(source),
  TgtMod = self.per_unit_type(target),
  SrcMod.value = "Basic Unit",
  TgtMod.value = "Sale Unit",
  SrcPrice = self.value,
  Part = self.part,
  Name1 = self.distributor,
  PartValue = Part.value(c_man),
  'DistDla_Unit_Map_p'(Dist, Name2, RBasic),
  'Dist_Dla_p'(Dist2, Dla),
  Name1.value = Name2.value,
  Dist.value = Dist2.value,
  PartValue = Dla.value,
  $ = SrcPrice * RBasic.value;

```

```

%%-----
%% conversion functions w.r.t. part_id
%%-----

```

```

cvt(<part_id> <-
  SrcMod = self.preferred_id(source),
  TgtMod = self.preferred_id(target),
  SrcMod.value = TgtMod.value,

```

```

    $ = self.value;

cvt(<part_id> <-
  SrcMod = self.preferred_id(source),
  TgtMod = self.preferred_id(target),
  SrcMod.value = "Distributor Part #",
  TgtMod.value = "Manufacturer Part #",
  SrcVal = self.value,
  'Dist_Man_p'(Dist, Man),
  SrcVal = Dist.value,
  $ = Man.value;

cvt(<part_id> <-
  SrcMod = self.preferred_id(source),
  TgtMod = self.preferred_id(target),
  SrcMod.value = "Manufacturer Part #",
  TgtMod.value = "Distributor Part #",
  SrcVal = self.value,
  'Dist_Man_p'(Dist, Man),
  SrcVal = Man.value,
  $ = Dist.value;

cvt(<part_id> <-
  SrcMod = self.preferred_id(source),
  TgtMod = self.preferred_id(target),
  SrcMod.value = "Manufacturer Part #",
  TgtMod.value = "NSN",
  SrcVal = self.value,
  'Man_Dla_p'(Man, Dla),
  SrcVal = Man.value,
  $ = Dla.value;

cvt(<part_id> <-
  SrcMod = self.preferred_id(source),
  TgtMod = self.preferred_id(target),
  SrcMod.value = "NSN",
  TgtMod.value = "Manufacturer Part #",
  SrcVal = self.value,
  'Man_Dla_p'(Man, Dla),
  SrcVal = Dla.value,
  $ = Man.value;

cvt(<part_id> <-
  SrcMod = self.preferred_id(source),
  TgtMod = self.preferred_id(target),
  SrcMod.value = "Distributor Part #",
  TgtMod.value = "NSN",
  SrcVal = self.value,

```

```

    'Dist_Man_p'(Dist, Man1),
    'Man_Dla_p'(Man2, Dla),
    SrcVal = Dist.value,
    Man2.value = Man1.value,
    $ = Dla.value;

cvt(<part_id> <-
    SrcMod = self.preferred_id(source),
    TgtMod = self.preferred_id(target),
    SrcMod.value = "Distributor Part #",
    TgtMod.value = "NSN",
    SrcVal = self.value,
    'Dist_Man_p'(Dist, Man1),
    'Man_Dla_p'(Man2, Dla),
    SrcVal = Dla.value,
    Man2.value = Man1.value,
    $ = Dist.value;

%%-----
%% conversion functions for strings, a simple value
%% cvt function for string is overridden by the following:
%% * part_id
%% * price
%%-----
cvt(<string> <-
    $ = self.value;

end c0;



## C.4 Elevation Axioms



use('/home/tpena/work/coinlc/Penny/examples/darpa/penny/dm0.pny');

%%-----
%% Dla_Inventory_v
%%-----
elevate 'Dla_Inventory_v'(nsn, stock, req, dist, distpno, usale, ratio,
    price, man, manpno, des, uman, count, type, size,
    style, other)

in c_dla
as 'Dla_Inventory_v_p'(^nsn : part_id, ^stock : string, ^req : string,
    ^dist : string, ^distpno: string, ^usale: string,
    ^ratio : string, ^price : price, ^man : string,
    ^manpno: string, ^des : string, ^uman : string,
    ^count : string, ^type : string, ^size : string,
    ^style : string, ^other : string)
{

```

```

    ^price.part = ^nsn;
    ^price.distributor = ^dist;
};

%%-----
%% King_Medical_Supplies_v
%%-----
view 'King_Medical_p'(A,B,C,D,E,F,G,H,I,J,K,L,M,N)
over 'King_Medical_Supplies_v_p'(A,B,C,D,E,F,G,H,I,J,K,L,M,N);

elevate 'King_Medical_Supplies_v'(nsn, distpno, usale, ratio, price, man,
                                manpno, des, uman, count, type, size,
                                style, other)

in c_dist
as 'King_Medical_Supplies_v_p'(^nsn   : string, ^distpno: part_id,
                              ^usale  : string, ^ratio   : string,
                              ^price  : price,  ^man     : string,
                              ^manpno : string, ^des     : string,
                              ^uman   : string, ^count   : string,
                              ^type   : string, ^size    : string,
                              ^style  : string, ^other   : string)
{
    ^price.part = ^distpno;
    ^price.distributor = ~dist;

    ~dist.value = "KING MEDICAL SUPPLIES";
};

%%-----
%% BMI
%%-----
view 'BMI_v'(A, B, C, D, E, F, G, H, I, J, K, L, M, N)
over 'Part_Description'(H,K,L,M,N), 'BMI'(A,B,C,D,E,F,G,H,I,J);

elevate 'BMI_v'(nsn, distpno, usale, ratio, price, man, manpno,
               des, uman, count, type, size, style, other)

in c_dist
as 'BMI_v_p'(^nsn   : string, ^distpno: part_id, ^usale: string,
            ^ratio  : string, ^price  : price,  ^man  : string,
            ^manpno: string, ^des    : string, ^uman  : string,
            ^count  : string, ^type   : string, ^size  : string,
            ^style  : string, ^other  : string)
{
    ^price.part = ^distpno;
    ^price.distributor = ~dist;

    ~dist.value = "BMI";
};

```

```

%%-----
%% Allied_Health_Corp_v
%%-----
view 'Allied_Health_p'(A,B,C,D,E,F,G,H,I,J,K,L,M,N)
over 'Allied_Health_Corp_v_p'(A,B,C,D,E,F,G,H,I,J,K,L,M,N);

elevate 'Allied_Health_Corp_v'(nsn, distpno, usale, ratio, price, man,
                               manpno, des, uman, count, type, size,
                               style, other)

in c_dist
as 'Allied_Health_Corp_v_p'(^nsn   : string, ^distpno: part_id,
                           ^usale  : string, ^ratio   : string,
                           ^price   : price,  ^man     : string,
                           ^manpno : string, ^des     : string,
                           ^uman   : string, ^count  : string,
                           ^type    : string, ^size   : string,
                           ^style   : string, ^other  : string)

{
    ^price.part = ^distpno;
    ^price.distributor = ~dist;

    ~dist.value = "ALLIED HEALTH CORP";
};

%%-----
%% Auxillary Tables
%%-----
elevate 'Dla_Inventory_v'(_,_,_,_,dpo,_,_,_,_,mpo,_,_,_,_,_,_)
in c_dist
as 'Dist_Man_p'(^dpo : string, ^mpo : string) {};

elevate 'Dla_Inventory_v'(nsn,_,_,_,dpo,_,_,_,_,_,_,_,_,_,_)
in c_dist
as 'Dist_Dla'(^dpo : string, ^nsn : string) {};

elevate 'Dla_Inventory_v'(nsn,_,_,_,_,_,_,_,_,mpo,_,_,_,_,_,_)
in c_dist
as 'Man_Dla_p'(^mpo : string, ^nsn : string) {};

elevate 'Dla_Inventory_v'(_,_,_,_,_,_,_,_,_,mpo,_,_,count,_,_,_,_)
in c_dist
as 'Man_Unit_Map_p'(^man : string, ^count : string) {};

elevate 'Dla_Inventory_v'(_,_,_,name,dist,_,ratio,_,_,_,_,_,count,_,_,_,_)
in c_dist
as 'DistDla_Unit_Map_p'(^dist : string, ^name : string, ^count : string)
{

```

```
^count.value = $ <-  
  $ = @ratio * @count;  
};
```

Bibliography

- [ALUW93] S. Abiteboul, G. Lausen, H. Uphoff, and E. Walker. Methods and rules. In *Proceedings of the ACM SIGMOD Conference*, pages 32–41, Washington DC, May 1993.
- [ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [BBC97] E. Bertino, S. Bressan, and B. Catania. Integrity constraint checking in chimera. In *Proceedings of Constraint Databases and Applications (ICDT97)*, 1997.
- [BFP⁺97a] S. Bressan, K. Fynn, T. Pena, C. Goh, and et al. Demonstration of the context interchange mediator prototype. In *Proceedings of ACM SIGMOD/PODS Conference on Management of Data*, Tucson, AZ, May 1997.
- [BFP⁺97b] S. Bressan, K. Fynn, T. Pena, C. Goh, and et al. Overview of a prolog implementation of the context interchange mediator. In *Proceedings of the Fifth International Conference and Exhibition on the Practical Applications of Prolog*, pages 83–93, London, England, April 1997.
- [BLG_{ea}97] S. Bressan, T. Lee, C. Goh, and et al. A procedure for context mediation of queries to disparate sources. 1997. Submitted.
- [BRU96] P. Buneman, L. Raschi, and J. Ullman. Mediator languages – a proposal for a standard, April 1996. Report of an I³/POB working group held at the University of Maryland.
- [CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), March 1989.
- [DG89] M. Dalal and D. Gangopadhyay. OOLP : A translation approach to object-oriented logic programming. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD-89)*, pages 555–568, Kyoto, Japan, December 1989.
- [DT94] G. Dobbie and R. Topor. Representing inheritance and overriding in datalog. *Computers and AI*, 13(2-3):133–158, 1994.
- [DT95] G. Dobbie and R. Topor. Resolving ambiguities caused by multiple inheritance. In *Proceedings of the Sixth International Conference on Deductive and Object-Oriented Databases*, Singapore, December 1995.

- [GMR96] J. Grant, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proceedings of the CP96 Workshop on Constraints and Databases*, Cambridge, MA, August 1996.
- [Goh96] Cheng Hian Goh. *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems*. PhD thesis, Sloan School of Management, Massachusetts Institute of Technology, 1996.
- [KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, (4):741–843, 1995.
- [Law93] M. Lawley. A prolog interpreter for f-logic. Technical report, Griffith University, Australia, 1993.
- [Lef93] A. Lefebvre. Implementing an object-oriented database system using a deductive database system. Technical report, Griffith University, Australia, 1993.
- [Mad96] S. E. Madnick. Are we moving toward an information superhighway or a tower of babel? the challenge of large-scale semantic heterogeneity. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 2–8, April 1996. Also reprinted in 21st (a Web 'Zine at <http://www.bxm.com/>).
- [O'K90] R.A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.
- [SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog : Advanced Programming Techniques*. The MIT Press, 1994.
- [Tuc86] A. B. Tucker. *Programming Languages*. McGraw-Hill, 1986.