Incorporating Time in the New World of Computing System

Hean Lee Poh

Computer Science Department
California Institute of Technology

5238:TR:87

Incorporating Time in the New World of Computing System

Hean Lee Poh

Master's Thesis

Computer Science Department

California Institute of Technology

5238:TR:87

(submitted July 30, 1986)

Acknowledgment

Special thanks to Bozena and Fred Thompson for their care, help and guidance, without which this thesis would not have been realized.

I am also grateful to Charley Kahler for his friendship and advice and thankful to those people in the New World System group and faculty and fellow students in the Caltech Computer Science Department who have helped me in one way or another. In particular, I would like to express my appreciation of help and friendship to John Ngai, Devendra Kalra, Jin Luo and Peggy Li.

Last but not least, I would like to thank the National University of Singapore for the continual support of my graduate studies.

Abstract

The New World of Computing System, referred to as the New World system, is a total system for the structuring, manipulation and communication of information. Time is a ubiquitous aspect of most databases. The aim of this thesis is to study the problems associated with the implementation of time in the New World system. Time information is not only stored in New World, they can be retrieved and processed to answer various types of user queries. This is an additional feature as compared to most models of time implementation in databases where the relationships between time intervals are not dealt with. To start with, ways of representing time in the form of floating point number are devised and discussed. Then the conversion of time information from its various user accustomed forms to New World system internal form and back are explored. Finally, the ambiguities and complexities involved in finding the intersection, subtraction, union and extension of two different sequences of time intervals associated with an object in a database are studied and algorithms for resolving these are presented . An explanation on how the crunchers work with the addition of time information is also given. This includes discussing about how quantifiers such as *at least 2*, *how many* etc. are handled in the New World system. Case studies are also conducted to test out these routines. As conclusion, the remaining problems associated with time implementation not covered in this thesis work are discussed.

Contents

Incorporating Time in the New World of Computing System

Hean Lee Poh
Master's Thesis
California Institute of Technology

1. Introduction

1.1 Time in database systems

Time is an important part of information about the constantly changing real world. Facts or data often refer to time. Time appears in financial accounts, movements of aircraft or ships, scheduling of jobs in factories, publication of books, just to name a few. Due to its importance in most information management applications, it deserves special attention.

This thesis discusses the methods for including time information in a particular database system, namely the New World of Computing System (refer to herein as the New World System). Section 2 gives a brief introduction to the New World System.It then investigates the various problems encountered in incorporating time information in New World. Finally, it discusses the remaining problems relating to time that are not covered in this thesis but deserve further research.

1.2 Existing work

Researchers in different fields have been concerned with the role that time plays in Information Processing. The designers of information systems are interested in keeping track of past data. Researchers in Artificial Intelligence have worked on a more realistic model of the real world representing not only snapshot knowledge but also knowledge about histories etc, and among them are logicians working on 'temporal logic' to capture the relationships of some statements with temporal reference, such as [1] [2]. The analysis of tensed statements is another domain of interest relating to time that is currently being pursued by the linguist [3] [4], and some computer scientists, in particular, Harper and Charniak who work on the semantic representation of such temporal information as tense, temporal adverbs and temporal connectives.[5] There is also an attempt [6] to relate works

in these two fields to one another, such as the exploration of the relationships between a computational temporal representation by James Allen [1] and the theory of tense by Norbert Hornstein [3]. A survey of work done in this area till 1982 is in [7]. However, most researchers working on time in knowledge representation adopt a theoretical approach, either working on problems related to temporal logic or proposing various ways of handling and representing time information, but I have not seen any work that actually implements time information processing capabilities in a system like New World, although a model for including time in a database query system has been proposed by Hafner [8], and a preliminary implementation to generate the examples of the model's capabilities has been done by her. However, the model does not indicate any capabilities with repect to the handling of the relationships between time intervals (e.g. time intersection etc.) and their significance in the handling of the duration of an object's membership in a class and an object being the attribute of another object.

1.3 The New World of Computing System*

The New World of Computing System, referred to as the New World System, is a total system for the structuring, manipulation and communication of information.[9] The user interface, or the language of communication between user and machine, is mainly a limited dialect of English. However, efforts have also been made to include French and Italian in the system, so that a user can work in a multi-language environment.[10] [11] In contrast to expert systems, in which experts build the knowledge base and users make use of this expert knowledge, New World guarantees user participation whereby users can create, test, modify, extend and make use of his own knowledge base. The main application of New World can be found in a research laboratory, business office or military department. It is a system for the professionals who command an overview of the organisation of the knowledge base and they can readily make changes by themselves. Besides, New World serves as a window to the outside world in that access to foreign databases can be accomplished via New World network system.[12]

1.4 Adding time to the New World system

Adding time to the New World System involves changing
(1) the New World English Syntax to include syntax rules that allow expressions of time to be understood;
(2) the New World English semantics in ways that will incorporate time in the database structures and carry out the processing of these data.

* The New World of Computing System was referred to previously in the literature as the ASK system, **A** Simple **K**nowledgeable System.

The second of these implies that time must have some internal representation within the New World database structures.

The result of incorporating time will allow the system to understand statement and question with time. The remainder of this thesis concerns how this was accomplished.

With the implementation of time, a more realistic picture can be drawn while providing answer to queries such as '*How many employees of each company are there?*'. In this case, for each company, the number of employees is enumerated with reference to the time interval during which this particular number is valid, for instance, '*2 employees in Company A from t1 to t2*' and '*3 employees in Company A from t2 to t3*', for t1, t2, t3 being some points of time.

The following example explains the way time is handled in a query in New World:

How many employees of each company are there?

Company	Employee	from (date)	to (date)
A	John	June 1, 1980	May 1, 1985
A	Mary	January 20, 1976	till now
B	Tom	February 15, 1983	March 1, 1986
C	Susan	April 15, 1984	till now
D	Mike	March 15, 1985	till now
D	Helen	April 1, 1983	April 30, 1986

The answer to the query based on the above data is given below:

```
A   0   ending January 20, 1976
    1   starting January 20, 1976 to ending June 1, 1980
        starting May 1, 1985
    2   starting June 1, 1980 to ending May 1, 1985

B   0   ending February 15, 1983
        starting March 1, 1986
    1   starting February 15, 1983 to ending March 1, 1986

C   0   ending April 15, 1984
    1   starting April 15, 1984

D   0   ending April 1, 1983
    1   starting April 1, 1983 to ending March 15, 1985
        starting April 30, 1986
    2   starting March 15, 1985 to ending April 30, 1986
```

As can see from the above example, to say that Company A has two employees is incomplete without the indication of time.

2. Background about New World

2.1 General organization of New World processing

The language processor of New World is a general rewrite rule-procedural semantics processor. It is a well known fact that the syntax of any well defined language can be specified by a general rewrite rule grammar. Thus a language in New World is implemented by declaring its syntax under any general rewrite rule grammar and defining its associated semantic procedure for each syntax rule. The New World system is 'sentence-driven' and its processing can be thought of as being organized around the following paradigm:

```
The system types a ">", indicating that it is ready for user input, and
waits for the user to respond.

The user enters a sentence---more generally any string ending in a return
key. (Such a string is often referred to as 'sentence' even when it does not
parse to a sentence in New World)

The system responds by processing the sentence, displaying the results
on the user's terminal, and cycles back to the first step.
```

So when a user enters a sentence, it is parsed. As a result of this parse, the relevant semantic procedures associated with the rules, and perhaps procedures associated with words in the sentence, are identified. Calls to these procedures and perhaps also the utilities are compiled and executed in the processing of the sentence.

2.2 Syntax rules and semantic procedures

In response to a query, the New World System goes through two phases, the syntax phase and the semantic phase, before an appropriate reply can be produced. During syntax processing phase, or known simply as syntax time, the sentence is parsed according to a set of grammar rules available in the system. This is done by matching the sentence structure with the appropriate grammar rule called syntax rule. Every syntax rule is associated with a semantic procedure which is basically a Pascal program. Therefore, after syntax time the semantic procedure which is linked to the matching syntax rule is executed. This phase is known as the semantic phase or semantic time.

The format for the syntax rule is as follows:
A rule is initiated by the word 'RULE' which is followed by a comment on the same line. This comment appears on the screen of the computer when the system assimilates the

whole syntax. The following lines are composed of a left hand side and a right hand side separated by a right arrow, as shown in an example below. Basically, the right hand side gets rewritten into the left hand side's format. Both sides can consist of two elements: literals which are enclosed between double quotes and parts of speech which are enclosed between a 'less than' and a 'greater than' sign. The part of speech may require certain *features* to be set or reset. This is specified in a list placed after the name of the part of speech and preceded by a colon. A plus or minus sign preceding the features indicates that that feature has to be turned on or off before the part of speech can be considered as fitting the rule. The last line starts with either a LEX, a SYN, a PRE or a POST, followed by a number which enumerates the various syntax rules in a file. These four markers specify at what time the rule should be applied. LEX means that the rule should be added in the lexicon. SYN means the rule should be applied at syntax processing time. PRE or POST means that it should be applied during semantic processing time before or after preprocessor is called. After a SYN, PRE or POST comes the name of the semantic procedure that is associated with the rule and is called as a result of the parsing of the part of speech/sentence.

An example of the syntax rule is given below:

```
RULE "June 3 of 1979" or "3rd of June of 1979"
<time:1-lit-td> => <time:+sdy-smo+td> " " <preposition:+of> " " <whole_number>
POST 307 con_date1
```

2.3 New World semantic utilities

The New World system provides many utilities that may be used by application programmers to extend New World to cover various applications domains. In this work on time, the utilities that were involved were of two kinds:

(1) modifying the data base handling utilities to include time consideration. These modifications will be discussed in section 7.4;

(2) the addition of utilities for handling time. The utility that has been added is called *'timeuty.pas'* which contains the *time_out* routine to be discussed in section 6 and the routines for handling time_intersection, time_subtraction, time_union and time_extension which will be explained in section 7.3.

3. The four sub-problems of implementing time in New World

3.1 General discussion

In the implementation of time information in New World, four problems are focused on. First, a way of storing the time information in the form of a floating number is devised so that calculations such as the duration of a certain event based on these information can be easily done. Therefore the New World internal representation of time is discussed here. Very often time information provided by the user is incomplete, or inexact. For instance, *after Tuesday* does not tell us when the event actually took place, Monday, Tuesday, Wednesday or even after that. Furthermore, the interpretation differs from person to person. [13] To capture this intrinsic ambiguity in time information , the concept of '*endflag*' which will be discussed in 4.1 is introduced. Then there is the rounding error problem associated with floating point representation. Using the depth of focus, an attempt to deal with the rounding errors to a certain extent is explained in 4.4.

The second and third problems deal with the conversion of user input form of time (from outside) into the New World internal form (to inside) and back. In section 5, the time syntax rules that are used to parse the various time input phrase and the associated semantic procedure that actually does the conversion are discussed. Further, a brief discussion on how the time output should look like and how this is done by the *time_out* routine can be found in section 6. Examples will also be given to demonstrate the function of this routine.

Finally, section 7 shows how time information can be stored in records and how these timed records can be processed. This includes discussing about the *intersection, union, subtraction and extension* of time information. An explanation on how the crunchers work with the addition of time information in New World is included here. The crunchers that will be discussed are *one_class_cruncher, member_cruncher and boolean_cruncher.*

3.2 The unit of time

Time has been incorporated into the basic structure of the New World record. It indicates the duration* of membership of an object in a class or the duration of an object being an attribute to another object. To facilitate the processing of time information, an internal representation of time has been devised. It calls for a basic unit of time which does not exist in natural language. Thus, time in New World is a real number measured

* By duration it means the validity period of membership to a class, and the duration needs not be finite, it could start from the beginning of mankind or it could start from a definite point in the past and continue up to today

in units of 1000 seconds from an origin which was set to be 0:00 hr January 1, 1980. The following example explains the definition of internal form of time in New World:

```
t = 64832.2 = 366*24*60*.06 (for 1980)
             +365*24*60*.06 (for 1981)\cr
             + 19*24*60*.06 (for the first 19 days of January 1982)
             +    12*60*.06 (for the first half of Jan 20, 1982)
```

The above is the internal form of noon of January 20, 1982.

3.3 User representation of time vs. internal representation of time

The user representation of time can appear in various forms such as 'January 20, 1982' or 1/20/82 which cannot easily be used for calculation of time interval unless it is converted to some standard representation format, and preferably floating point numbers. Hence, there is the internal representation of time as mentioned in 3.2. Since the internal representation of time comes in the form of floating point number, and hence rather unfriendly, it is converted back to the user representation form when the processing of time information is completed and is due for output to the terminal.

4. Internal Representation of time

4.1 Time interval and its end points

Time interval is a representation given to a period of time between two points of time which are called *endpoints*. These endpoints may or may not be bounded. Due to the uncertainties associated with time information, in particular with the starting point and ending point (here denoted as *left endpoint* and *right endpoint* respectively) of a time interval, there is a need to introduce a *flag* to each of the endpoints of an interval to specify whether the event actually takes effect from that particular point of time or it could have happened much earlier or much later. The same consideration is given to the ending of a time interval. This leads to three *endflags*:

For the left endpoints:

```
        t
end_in  >--- indicates that the associated event started on or after time t

        t
end_at  |--- indicates that the associated event started at time t

        t
end_out <--- indicates that the associated event started on or before time t
```

For right endpoints:

```
           t
 end_in ---< indicates that the associated event ended on or before time t
           t
 end_at ---| indicates that the associated event ended at time t
           t
 end_out---> indicates that the associated event ended on or after time t
```

Thus in the following example:

```
       t1                  t2
       |------------------->
```

indicates that the associated event started at time t1 and was still in progress at time t2 and

```
t1                  t2
<-------------------<
```

indicates that the associated event had definitely started by t1 and terminated by the time t2.

The information on time endpoints and endflags are kept in a time list in New World which will be discussed in section 7.2.

4.2 Data structure of the time interval

Because of other global assumption of New World, it is convenient to use one New World field of 64 bits to hold the information about a time interval. For each end of the interval, there are three possible endflags. Thus 2 bits are allocated for each of the two endflags. The endpoint of a time interval is a real number. The representation we have chosen is a floating point number whose mantissa is 23 binary digits plus sign. Its characteristic is a binary integer between -63 and +63.

Since the unit of time is 1000 seconds, the smallest increment of time is $2^{-63} * 1000$ seconds , or approximately 1/1000 pico second. The largest time that can be represented is roughly $2^{63} * 1000$ seconds, which is in the region of over 30 trillion years.

These calculations show that the limitation imposed by the characteristic does not lead to any practical difficulties. The same is not true for mantissa, whose limitation of 23 bits may lead to roundoff errors.

4.3 Rounding error

As mentioned earlier, the limitation imposed by the mantissa requires somehow careful analysis. Any positive time can be uniquely expressed as

$$t_1 = m_1 * 2^{c_1} * 1000 second$$

where c_1 is an integer and

$$1 \leq m_1 < 2$$

Suppose t1 is the time to be captured and it is precise to within 1 second, then t1 should be less than 94 days. Since

$$t_2 = 1 second = m_2 * 2^{c_2} * 1000 second$$

where

$$1 \leq m_2 < 2$$

Now since

$$c_2 = -10$$

in order that

$$c_1 - c_2 < 23$$

so that all the 23 significant binary bits of the mantissa will be preserved , c1 can only be as large as 12. Therefore, $2 * 2^{12} * 1000$ gives us 8,192,000 seconds which is about 94 days. Hence, the limitation imposed by the mantissa leads to roundoff error in the internal representation of time. The following table gives conservative estimates of the time intervals on either side of the origin of time in New World, i.e. New Year's eve, 1980, within which time the representation is precise to the given unit of time.

precise to	**within**
microsecond	7 seconds
second	94 days
minute	8 years
hour	532 years
day	16000 years
year	4 million years

Considering the nature of applications wherevarious degrees of precision in time is required, these seem quite adequate.

An alternative would have been to use fixed point rather than floating point for time. Suppose the data structure is an integer which is represented by 29 bits plus a sign bit. Hence, the smallest integer is $-2^{29} + 1$ and the largest is $2^{29} - 1$. The following table gives the span of time that can be represented given the different time units:

time unit	**span**
microsecond	12.4 days
minute	2042 years
day	2.94 million years
year	1073.6 million years

If we were willing to have say 4 realizations of New World, one for each unit in the above table, either fixed point of floating point would be quite satisfactory. It becomes a matter of taste which to use.

4.4 Using depth of focus to deal with rounding errors

One way of reducing the effects of rounding errors is the specification of depth of focus in time . From 4.3, it is clear that a time interval can be accurately reproduced at the output if the focus is on day, month, year such as June 3, 1979 and the time interval is of duration less than 16000 years. From the New World internal processing point of view, a time t is simply a floating point number without any recognition of the degree of significance (how precise) a certain date should be. Suppose the internal answer to the query '*When did John become an employee of ABC Inc?*' was 64832.2 (internal form of 1/20/82), the following three output forms are possible:

(1) January 20, 1982
(2) 1982
(3) 0:00:00.0 AM January 20, 1982

How accurate the answer in the output form has to be depends a great deal on the context that the answer is meant for. Thus, a measure of its significance called the significance unit, or sig_unit is attached to a time datum to specify its depth of focus. There are five sig_units, namely sig_second, sig_minute, sig_day, sig_month, sig_year which refer to accuracy up to second, minute, day, month, year respectively, for example:

output time	**with sig_unit**
1970	sig_year
June 1, 1982	sig_day
12:56am, 3/6/83	sig_minute
10:25:34	sig_second

Three examples of the use of sig_units in records handling is given below:

(1)

```
                                          low sig_unit   high sig_unit
>class: employee                              ---            ---
>John was an employee in 1982.               year           year
>Mary was an employee after June 3, 1982.    day            year

>Who were employees?
 John  starting January 1, 1982 and ending January 1, 1983.
 Mary  from on or after June 3, 1982.
```

(2)

```
                                                low sig_unit   high sig_unit
>class: animal                                      ---            ---
>Dinosaur was an animal which lived                 year           year
 from 225000000 BC to 65000000 BC.
>Fish was an animal which lived                     year           year
 from 400000000 BC.

>There were what animals?
 Dinosaur  on or before 225 * 10**6 years to on or after 65 * 10**6 years.
 Fish      from on or before 400 * 10**6 years.
```

(3)

```
                                                    low sig_unit   high sig_unit
>class: data                                           ---            ---
>Data A was collected starting 0.015231117 second     second         second
 and ending 0.017237856 second
>Data B was collected starting 0.000987466 second     second         second
 and ending 0.001254735 second

>What data were there?
 A  starting 15.23 milliseconds to ending 17.24 milliseconds
 B  starting 987.47 microseconds to ending 1.25 milliseconds
```

Due to the second input statement, the low sig_unit is modified to day and hence day and month are given in the output in response to the queries.

5. From outside to inside

5.1 Two examples of time syntax rules with semantic procedures

Due to the consideration of leap year * , depth of focus of time, and floating point evaluation, the task of conversion becomes quite complex. Therefore, algorithms are written for converting time points or time intervals that appear in various forms and require different treatment as such. It is impossible to enumerate and explain every syntax rule and its associated semantic procedures, so two of them are chosen to be presented here. They convert some of the commonest user representation forms of time, one of which deals with a time point and the other a time interval:

```
1. RULE "June 3, 1979" or "3rd of June, 1979"
   <time:1-lit-td> => <time:+sdy-smo+td> <pct:+comma> <whole_number>
   POST 307 con_date1
```

This syntax rule applies to time that is given in the above form, i.e. month, day followed by year. It should be noted that the application of this rule occurs after the application of another rule which deals with month and day only.

In semantic time, the semantic procedure con_date1 is called. Con_date1 picks up the year and the internal value of month and day which is the result from the semantic procedure associated with the abovemetioned syntax rule that deals with month and day only. It then converts year to the internal form using a procedure called *time_in* in a utility for handling time. Then it adds this internal value for year to the internal value for month and day. If it is a leap year and the month is March or later then it adds a day (86.4 units) to the total internal time. The time focus is set to sig_year for *high* and sig_day for *low* in this case.

---

* A leap year occurs every fourth year, but only those centesimal years divisible by 400. In other words 1900 is not a leap year but 2000 is a leap year. Thus to determine whether a year is a leap year, the function 'leapyear' can be applied which returns a *true* if it is a leapyear i.e.:

```
if ((year mod 4)=0) and (((year mod 100)<>0) or ((year mod 400)=0))
then leapyear is set to true
else leapyear is set to false;
```

```
2. RULE "after" <time:-td>
   <time:2+beg+prep> => <time:+lit+after> " " <time:-lit-td-prep>
   POST 335 after_time
```

This syntax rule deals with a time interval that begins from a certain point of time given in the various forms, one of which may be that presented in the rule above, and continues into an indefinite time in the future. Such a time interval is always given in the form of a time (date for instance) preceded by a preposition called '*after*'.

The associated semantic procedure is known as *after_time*. The time immediately following the preposition *after* is taken as the new left endpoint of an interval, and the right endpoint is set to plus infinity, which means that after a certain date is equivalent to having an interval which is not bounded at the right end. Also, the left endflag is *end_in* which means that the starting time of this interval is uncertain, it may start on or after that date. As for the right endflag, it is set to be *end_out* since infinity, be it plus or minus, is always associated with an endflag of *end_out*.

5.2 The coverage of New World time syntax

The following groups of syntax rules are written for parsing time expressions in New World:

(1) Lexical time rules

This is a set of lexical rules which adds the various time measuring units and time adverbs into the lexicon. Below is a summary of time expressions and adverbs to be stored in the lexicon:

— Time expression which measures or describes a certain period of time:
January-December, AM, PM, Monday-Sunday, BC ,AD, second/minute/day/week/ month/ year/century, Christmas/Thanksgiving and other festive seasons, Spring-Winter etc.

— Time adverb for pointing to a certain time point or interval:
this/last/next/later, beginning/starting/ending, before/on or before/after/ on or after,from,since,to, etc.

(2) Rules for time statements

This set of rules is associated with the respective semantic procedures to convert a period of time or just a point of time to the internal representation form. The following time expressions are captured in this set of rules:

date without year: *June 3, 3rd of June*

time: *now, 10:25 pm etc.*

date with year: *June 3, 1982, June of 1970 etc.*

day: *today, yesterday, tomorrow*

particular date/time: *this/next/last day/week/month/year/decade/century*

(3) Rules that allows the addition of a time or simply a preposition to a date and also the specification of a particular point of time with respect to the present time. Examples of these are *5:00 pm today, at 5 o'clock, in year 1976, 15 years from now*

Similarly, a semantic procedure is associated with each rule. In the case of the addition of a preposition, the preposition is simply ignored in the evaluation of the internal value.

(4) Rules that put prepositions on a point of time which makes it a time interval

The prepositions involved here are:
after or starting after time which makes it into an interval with a left endflag of *end_in*
beginning or starting in time which makes it into an interval with a left endflag of *end_at*
from or starting before time which makes it into an interval with a left endflag of *end_out*
before or ending before time which makes it into an interval with a right endflag of *end_in*
ending or ending in time which makes it into an interval with a right endflag of *end_at*
until or ending after time which makes it into an interval with a right endflag of *end_out*

(5) Rules that deal with time intervals such as *from time t1 to time t2, in n weeks, after 3pm before 4pm , between t1 and t2, last 2 days, the next 3 months, in the next 4 weeks etc.*

In this case, the left and right endpoints are taken from each of the endpoints mentioned during semantic time. The endflags are determined by the respective prepositions like *after, before, from etc* and if these are not present, end_at is used.

(6) Finally, there are rules for parsing a question about time information into a sentence and rules for parsing time at the beginning or end of a verb phrase into a verb phrase.

A list of the syntax rules is given in Appendix A.

5.3 The time conversion functions

It is worth taking a look at the various functions that actually do the conversions of year, month and day independently:

(1) year_in

The function *year_in* converts the quantity of year to multiples of thousand seconds with respect to the *time origin* at 0:00 1st January, 1980. Hence, adjustments for a year B.C. have to be made in the sense that the year is added to 1980; and in the case of AD, 1980 is simply subtracted from the year. In addition, leapyear has to be taken into account in the calculation of the internal value of year. The details about the adjustments due to leapyear is given in the following program:

```
function year_in(year:integer;BC:boolean;var leap:boolean):real;
var yrdiff,qq,numleap:integer;
    daytotal:real;
begin
   if BC then year:=(year + 1980);      (* year B.C. *)
   if not BC then yrdiff:=year - 1980 else yrdiff:=year;
(* if BC is true, it means that we have a year B.C. *)
   if (yrdiff>0) and not BC then qq:=(yrdiff+3) (*to include 1980 as a leapyear
                                                  in the next step*)
   else qq:=yrdiff;
(* to calculate the number of leapyears *)
   if qq<0 then
      numleap:=(qq div 4)-((qq-20) div 100)+((qq-20) div 400)
      else numleap:=(qq div 4)-((qq-24) div 100)+((qq-24) div 400);
(* correction for the number of leap years: *)
   Since 1900 and 2100 are the first centennial non-leapyear
   before and after 1980 respectively; So we need a correction of qq-20 to
   exclude 1900 in the counting of leap year. Similary, 2100 is not a
   leapyear and hence we need a correction of qq-24 to exclude 2100 in
   the counting of leapyear. For instance, year 2101 corresponds to a qq
   of (2101-1980 +3) = 124 => 124-24 = 100 => 100 div 100 =1 and hence
   the caluculated leapyear does not include 2100 in the counting.
 (* having determined the number of leapyear, the rest is just trivial *)
   daytotal:=((365.0 * yrdiff)+numleap) * 86.4;  {86.4 = 24*60*.06}
   if BC then daytotal:=-daytotal;
   year_in:=daytotal;
   leap:=leapyear(year);
end;
```

(2) month_in

The calculation of months is quite straight forward, except that care has to be taken while converting 3/1/84 and 2/29/84. Since time is converted in stages just as time phrase is being parsed. Therefore month_in is called to evaluate the month February which gives us a value equivalent to 31 days of the year concerned. (In other words 31 days have elapsed since the beginning of February) . Similary for the month March it gives us 31+28 days of the year. Hence the internal representation for February 29, 1984 is the same as that of March 1, 1984. Since March 1, 1984 has just passed a leapday, the time representation of March 1, 1984 has to be adjusted by adding a day to it, causing its correct conversion to be $(31 + 29) * 24 * 60 * .06$ time units.

```
function month_in(month:integer;leap:boolean):real;
var mo:real;
begin
  case month of
   0,1: {January}   mo:= 0.0;
     2: {February}  mo:= 2678.4;
     3: {March}     mo:= 5097.6;
     4: {April}     mo:= 7776.0;
     5: {May}       mo:=10368.0;
     6: {June}      mo:=13046.4;
     7: {July}      mo:=15638.4;
     8: {August}    mo:=18316.8;
     9: {September} mo:=20995.2;
    10: {October}   mo:=23587.2;
    11: {November}  mo:=26265.6;
    12: {December}  mo:=28857.6;
  end(* case *);
  if (leap and (month > 2)) then mo:=mo + 86.4;   (* leap year and after Feb. *)
  month_in:=mo;
end(* month_in *);
```

(3) day_in The function day_in simply returns the internal representation for the number of days in that particular month ,but excluding the day concerned. In other words, in the preceding example, when converting the number of days in February 29, only 28 days are used in the calculation.

```
function day_in(day:integer):real};

begin
   if (day<0) then WRITELN('ERROR: Input to function day_in < 0 ') else
     day_in:=(day-1)*86.4;
end(* day_in *);
```

(4) time_in

Time_in sums up the result obtained in the preceding three functions to get the final internal time. Also, it checks on the entry of illegal time information, like 61 sec. etc.

```
function time_in(year,month,day,hour,minute,second:integer;BC:boolean):real};

var leap:boolean;
    yearin,monthin,dayin,temp:real;
begin
   if ((year<0) or
       (not((month >= 1) and (month <= 12))) or
       (not((day  >= 1) and (day <= 31))) or
       (not((hour >= 0) and (hour <= 23))) or
       (not((minute >= 0) and (minute <= 60))) or
       (not((second >= 0) and (second <= 60)))) then temp:=0.000000
   else
     begin
        yearin:=year_in(year,BC,leap);
        monthin:=month_in(month,leap);
        dayin:=day_in(day);
        temp:=yearin+monthin+dayin+3.6*hour+0.06*minute+0.001*second;
     end;
   time_in:=temp;
end(* time_in *);
```

6. From inside to outside

6.1 What the time output should look like

Having discussed the methods of converting time expressions that come in various forms into its internal forms, the next step would be conversion of the internal time back to its output user form. The following table describes the output range designated for a given input range. In other words, if the internal value falls within a certain range, the output representation is precise to a certain time unit only.

```
tt is the internal time to be translated to the external string form.
        i n p u t   r a n g e              o u t p u t   r a n g e
  year 10**7 AD < tt                    : year: yy * 10**6 years AD
 year 100000 AD < tt < year 10**7 AD    : year: yy * 10000 years AD
  year 10000 AD < tt < year 100000 AD   : year: yy centuries AD
     year 5200 < tt < 10000 yr          : year: yyyy AD
     year 2000 < tt < year 5200         : day: mm dd, year: yyyy
   Jan. 1, 1981 < tt < year 2000        : hour: hh:mm, day: mm dd, year: yyyy
   Jan. 1, 1979 < tt < Jan. 1, 1981     : hour: hh:mm, day: mm dd, year: yyyy
                                          and seconds: ss.sss
     year 1960 < tt < Jan. 1, 1979      : hour: hh:mm, day: mm dd, year: yyyy
     year 1066 < tt < year 1960         : day: mm dd, year: yyyy
        year 0 < tt < year 1066         : day: mm dd, year: yyyy AD
   year 1200 BC < tt < year 0           : day: mm dd, year: yyyy BC
  year 10000 BC < tt < year 1200 BC     : year: yyyy BC
 year 100000 BC < tt < year 10000 BC    : year: yy * centuries BC
  year 10**7 BC < tt < year 100000 BC   : year: yy * 10000 years BC
                tt < year 10**7 BC      : year: yy * 10**6 years BC
```

6.2 The time_out procedure – the organization of a very messy routine

In New World, the plus and minus infinity are defined as $3.689338*10^{19}$ and $-3.689338*10^{19}$ time units respectively. Hence, a natural step in converting the internal time to its output form is to ensure that the internal time falls within this range. Next, the input range is divided into two groups:

(1) tt falls either before 1200 BC or after 5200 AD;

(2) tt falls between 1200 BC and 5200 AD.

where tt is the input internal value.

In group 1, a function *bigtt* is called to evaluate the corresponding output values, i.e. the number of years. This is done by dividing tt by the internal value for 400 years which is 12622780.8 units. This range is sub- divided into the following ranges:

(a) tt is greater than 10^7 AD;

(b) tt falls between 100000 AD and 10^7 AD;

(c) tt falls between 10000 AD and 100000 AD;

(d) tt falls between 5200 AD and 10000 AD.

(e) tt falls between 10000 BC and 1200 BC;

(f) tt falls between 100000 BC and 10000 BC;

(g) tt falls between 10^7 BC and 100000 BC;

(h) tt is less than 10^7 BC;

Based on the table given in Section 6.1, different formats for the output strings are adopted for each of the subranges in the function *bigtt*.

Besides the output of a string for the number of years, a string indicating the endflags is also concatenated to it. The following table gives the corresponding strings for describing the various endflags when they are used as a left or right endflag:

For a finite time interval:

	strings for	
	left endflag	right endflag
	---	---
end_at	starting	ending
end_in	on or after	on or before
end_out	on or before	on or after

For a interval that is bounded on the left only

left endflag	string
---	---
end_at	starting
end_in	from on or after
end_out	from on or before

```
For a interval that is bounded on the right only
right endflag          string
------------           ------
end_at                 ending
end_in                 to on or before
end_out                to on or after
```

In group 2, similar to group 1, the range for tt is further divided into the following subranges:

(a) tt falls between 1/1/79 and 1/1/81;

(b) tt falls between 1/1/81 and year 2000;

(c) tt falls between year 1960 and 1/1/79;

(d) tt falls between year 2000 and 5200 AD;

(e) tt falls between year 1200 BC and 1960;

The internal time is converted into the appropriate output form by a procedure called *time_point_out* which also takes care of the correction due to leap year. This involves some tricky calculations since every 4 year is a leap year, but every 100 year is not a leap year except when it is every 400 years. (refer to 5.1) The reason for having the different subranges is that the accuracy in the output forms depends on the range that tt is in.(refer to 6.1). For example in the first case where something close to 1980 is represented, it would be natural to expect the output to be in seconds. Since in the process of converting to the internal value and back, a truncation error is inevitable, therefore the output value has to be adjusted by referring to the low sig_unit (lsu) associated with the time data. This done by a procedure called *round_time* which adds one unit to the next higher unit if the lower unit exceeds a certain threshold. In other words, the output value is rounded up based on the lsu. For instance, if lsu is sig_month, then the value for month is increased by one if the number of days in the output exceeds 17 days, or even if there are only 16 days, but the number of hours exceeds 12 or the month of output has only 30 days, the value of month is also rounded up by one month. In case the month is February and it is a non-leap year, the number of month is incresed by one if the output for days exceeds 15.

Similarly, the description for the endflags is also given by a string that is concatenated to the final output time data. The same description for endflags is adopted as that given above for group 1.

6.3 Examples of time output

Suppose, today's date is September 3, 1985, and the time now is 5:00:00 am, and it is a Tuesday. The following study is run to test the time output given a certain input form. In other words, the input is converted to the internal form and by means of the time_out routine the internal value is reconverted to a standard output form.

Applying rules for time statements [5.2 (2)]

```
>1201 BC?
 starting 1201 BC to ending 1200 BC

>10000 AD?
 starting 100 centuries AD to ending 100.01 centuries AD

>August 2, 10000?
 100.01 centuries AD
{round up}

>this Monday?
 September 2, 1985

>last Sunday?
 September 1, 1985

>next Sunday?
 September 8, 1985
```

Applying rules for time statements with date and time and/or with preposition etc [5.2 (3)]

```
>at 7:00:56 am on June 3, 1985?
 7:01 AM June 3, 1985

>at 5:00:00 tomorrow?
 5:00 AM September 3, 1985

>at 5:00:23 am yesterday?
 5:00 AM September 2, 1985
```

```
>12:00:12 pm today?
 12:00 noon September 3, 1985

>12:00:12 am on June 3, 1985?
 12:00 AM June 3, 1985

>5 o'clock today?
 5:00 AM September 3, 1985

>15 years from now?
 September 3, 2000

>3214 years from now?
 September 3, 5199

>3215 years from now?
 5200 AD

>4 years from 29/2/1896?  {European representation}
 March 1, 1900

>4 years from 3/1/1896?   {American representation}
 March 1, 1900

>4 years from 1/March/1896?
 March 1, 1900

>4 years from 29-February-1896?
 March 1, 1900

>115 years from next month?
 October 2100
```

Applying rules that deal with time intervals [5.2 (5)]
The left and right endflags are indicated to the right of the query within the braces.

```
>the next 2 days?                                    {end_at, end_at}
 starting September 4 and ending September 5, 1985

>from May 1, 1985 to October 1, 1988?               {end_out, end_out}
 on or before May 1, 1985 to on or after October 1, 1988

>after May 1, 1985 before October 1, 1988?          {end_in, end_in}
 on or after May 1, 1985 to on or before October 1, 1988

>between January 1, 1985 and February 1, 1985? {end_in, end_in}
 on or after January 1, 1985 and on or before February 1, 1985
```

More examples about the output of the time interval can be obtained in Appendix B.

7. Internal processing of timed records

7.1 How time is stored in timed data records

The New World data record consists of one or more pages in a linked structure. The first page is fixed and the page pointer to which is the name of the record. Each page in the record has a header and a body. Each header contains such information as the page pointers, record type, header size, field offset to last field used and n-tuple size for entries, timed_record_flag etc., the details of which is contained in [10], Ch.8.

The body of a record page consists of entries. Each entry on a leaf page consists of 1+n+t fields. The first field contains two integers, of which the first integer is the number 1+n+t. The following n fields contain the n-tuple of data. The last t fields are time information.

Thus time is incorporated as an extension to the data fields in the New World records. A typical entry with time information has the following length in New World record:

$$length = flag\_cum\_length\_of\_entry\_field(1) + fields\_per\_record(n) + time\_field(t)$$

7.2 Time lists and timed records

A time list is a list whose typical entry is:

```
t1[i]^: ( lrr,time_flags, t1[i+1], t1[i], t2[i] )
```

where the real numbers t1[i] and t2[i] are considered end points of a time interval $t1[i] \leq t2[i]$. Further, the intervals in a time list are also ordered; $t2[i] \leq t1[i+1]$. Thus time information is kept in terms of time intervals *. Information about the nature of the end points of a time interval is retained in the time_flags, which is an integer encoded by a simple function called *set_time_flag***

In records, the information identifying a time interval is packed into a single field***. t1 and t2, the start and end times of the interval form a floating point number with 23

---

* Even the time point that has been discussed extensively in the previous sections is stored as an interval; the length of the interval is, however, only one unit long

** set_time_flag simply combines the two endflags by the following :$set\_time\_flag = lef * 256 + ref * 64$ where lef is the left endflag and ref the right endflag. For this purpose, end_at is defined as 0, end_in 1, and end_out 2.

*** An New World *field* is always 8 bytes long but of variable type, e.g. real, pointer etc and it includes the type time_field

binary digit mantissa and 7 binary digit characteristic; the end point flags are each two bits. Packing and unpacking is done by two little assembly language procedures, thus it is implementation dependent, since the bit structure of the representation of the real number is implementation dependent.

As an illustration, suppose the salary of Mary was $2000 from June 3, 1981 to May 1, 1984, when it was raised to $3000, and she has been drawing this salary since then. Let

```
t1  = June 3, 1981
t2  = May, 1 1984
too = the positive time infinity (i.e. indefinite time in the future)
tf1 = left endflag associated with t1  (end_out)
tf2 = right endflag associated with t2 (end_at)
tf3 = left endflag associated with t2  (end_at)
tf4 = right endflag associated with too (end_out)
      Note: end_out is always associated with infinity
[ti:tfj] : the real number combining the time ti with the time flag tfj.
Mary_rec is the page address of the record for Mary
```

Then the time list:

```
tl[1]^:  ( lrr, tf1*4 + tf2, tl[2], t1, t2  )
tl[2]^:  ( lrr, tf3*4 + tf4, nil  , t2, too )
```

corresponds to the entry for Mary in the salary attribute record:

```
+-----------------------------------------------------------+
| header - salary attribute                                 |
| .....  .........  ....  ...                               |
| 5:...  Mary_rec   2000  [t1:tf1:t2:tf2]                   |
| 4:...  Mary_rec   3000  [t2:tf3:too:tf4]                  |
| .....  .........  ....  ...                               |
+-----------------------------------------------------------+
```

Putting time into records involves not only the transformation of time in the various external forms into the internal form but also the addition of these internal values into the New World record. The function which performs this is known as *add_to_record.* If the same entry already exists in the record, then the time already associated with this entry has to be changed to include the new time. The following study illustrates the incorporation of time in New World records:

```
Suppose today's date is July 17, 1986.

>class:employee
 The new class employee has been added.

>individual:Jane Smith
 The individual Jane Smith has been added.

>Jane Smith was an employee starting June 3, 1983.
 Jane Smith has been added to the class employee.
```

First of all, June 3, 1983 and today's date are converted to the internal form, namely 107913.6000 and 206377.3010 with the flag 128, since 'was' corresponds to a right endpoint of today's date and endflag of end_out. The system then searches in the record for II as an entry. It adds II as a new entry to the class CC together with the time interval if the entry was not in the class or it gets a *time_union* of the present time list and that of the same entry which is already residing in the record.

```
>was Jane Smith an employee?
 yes starting June 3, 1983 to ending July 17, 1986
 no  ending June 3, 1983
     starting July 17, 1986
```

The processing of this sentence is done by *is_proc* and a function called *eval_np_np* within *is_proc* which is presented in 7.3.

Further examples of the addition of timed data to a record and subsequent question about what data are available is demonstrated in the study of Appendix B. In this study, how time information are returned as output from the internal values is also demonstrated.

7.3 Time list intersection, union, subtraction and extension

When two intervals have to be operated like two sets, such as *intersection, union or subtraction,etc.*, the problem seems trivial. However, if the various time flags involved are considered the problem becomes more complex. In fact no definite answer can be provided for such set operations on time intervals in some cases; and this is attributed to the fact that there are inherently some ambiguities involved in the time flags such as *end_out* and *end_in*. This is due to the inadequacy of the underlying data representation of time in terms of intervals and endpoint characteristics. More will be said about this in section 8.3. An example for this is the intersection of the following two time intervals: Interval A exists

until on or before t_1 and interval B exists on or after t_2 where $t_2 < t_1$ There is no *exact* interval which is the intersection of these two intevals. However, such ambiguities do not arise frequently, as time endflags existing in the database itself tend to be *end_at* in most database applications. This problem is dealt with in the later part of this sub-section.

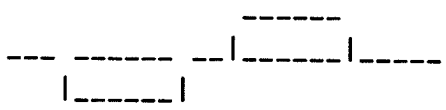
Another interesting problem is the recognition of the *relative position* of the endpoints of time intervals. This is essential since the program has to determine under which category such pair of intervals are to be operated. In other words, different results are obtained for operations on different categories of intervals.

Finally, there is the problem of operating with more than one intervals in each time list. In this case, for a start, the first two intervals are operated (intersected, subtracted, etc) , and the remainder of which is operated with the subsequent interval of one of the (appropriate) time list. This process continues till the last interval of one of the time lists. The problem becomes more complex when two intervals are generated as a result of the operation.

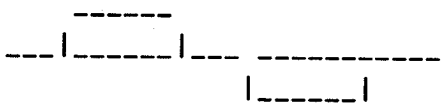
7.3.1 Recognition of Time Intervals

There are 13 possible cases in which two time intervals can be related to each other:

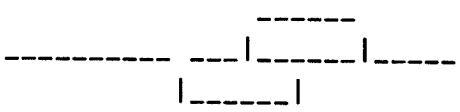
Case 1: *la,b,a3*



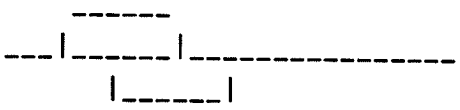
Case 2: *a,b,la3*



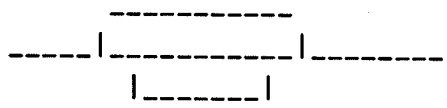
Case 3:*la,c2,a3*



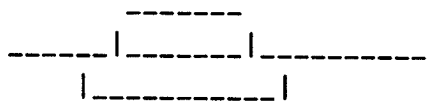
Case 4: *a,c2,la3*



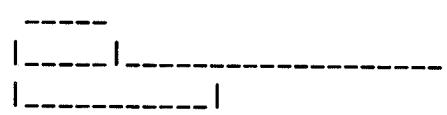
Case 5: *a,c2,a3*



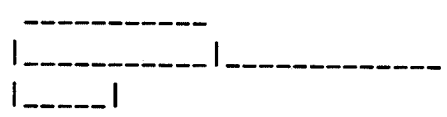
Case 6: *la,c2,la3*



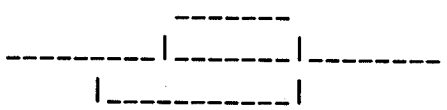
Case 7: *c,la3*



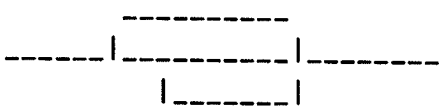
Case 8: *c,a3*



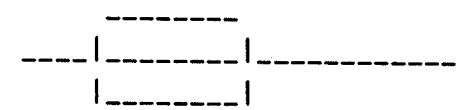
Case 9: *la,c2*



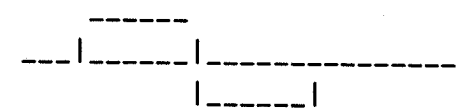
Case 10: *a,c2*



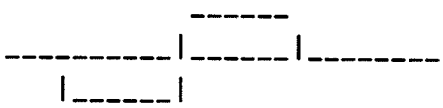
Case 11: *c*



Case 12: *a,la3*



Case 13: *la,a3*



Algorithm for recognition

Before any operations can be done on the intervals, the case in which the pair of intervals can be classified has to be identified. The following notations are used as features for identifying each of the cases:

Definition: An *interval section* is a section of an interval bounded by the end point of another time interval.

la : lower (first) interval section without intersection (i.e. it is not common to both intervals)

a : upper (first) interval section without intersection

la2: lower (second) interval section without intersection

a2 : upper (second) interval section without intersection

la3: lower (third) interval section without intersection

a3 : upper (third) interval section without intersection

b : no interval section and no intersection

c : (first) interval section with intersection

c2 : (second) interval section with intersection

```
(* t1, t2 are the two endpoints of the first time interval
   t3, t4 are the two endpoints of the second time interval *)

(* ta, tb, tc, td are the values of t1 to t4 sorted in increasing order
   i.e. from left to right on the time scale   *)

procedure recognition(var clsno:integer;ta,tb,tc,td,t1,t2,t3,t4:real);

   type idenfeat = (la,a,la2,a2,la3,a3,b,c,c2); {identification feature}
        class = set of idenfeat; {classification}
   var i:integer;
       cls:class;
```

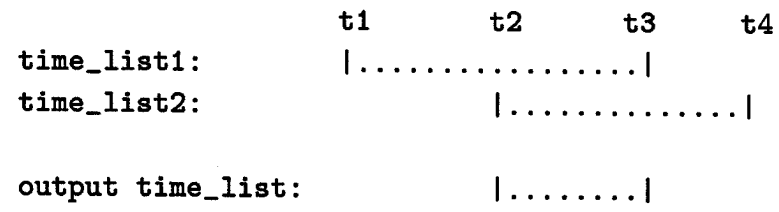
```
      t:array [1..4] of real;

   begin

cls:=[];
t[1]:=ta;
t[2]:=tb;
t[3]:=tc;
t[4]:=td;

(* i is a counter which keeps track of the numbering of interval section
   from left to right on the time scale *)

for i := 1 to 3 do begin
  if ((t[i]>=t1) and (t[i+1]<=t2)) and
     ((t[i]>=t3) and (t[i+1]<=t4)) then begin

(* feature c or c2 is set  *)

     if t[i]<>t[i+1] then begin
        if i=1 then
           cls:=cls+[c]
        else if (i=2) and not (c in cls) then
           cls:=cls+[c2];
        end
     else if i=1 then begin
        cls:=cls+[c];
        end;
     end
  else if ((t[i]>=t1) and (t[i+1]<=t2)) then begin

(* feature a or a2 or a3 is set  *)

    if i=1 then cls:=cls+[a]
    else if i=2 then cls:=cls+[a2]
    else cls:=cls+[a3];
```

```
    end
  else if ((t[i]>=t3) and (t[i+1]<=t4)) then begin

(* feature la or la2 or la3 is set  *)

    if i=1 then cls:=cls+[la]
    else if i=2 then cls:=cls+[la2]
    else cls:=cls+[la3];
    end
  else begin

(* feature b is set  *)

     cls:=cls+[b];
     end;
  end;{do-loop}

(* classification features completed *)

Then each case is defined with a combination of such features,
and 13 distinct combinations are obtained. The order of case number is
important as the algorithm searches through the cases one by
one and some cases which are 'subset' of other cases are placed
behind those cases in the numbering order.
Example:
    if (la in cls) and (b in cls) and (a3 in cls) then clsno:=1;

   end; {recognition}
```

7.3.2 Time intersection

Time_intersection returns the *intersection* of the given time lists, i.e., if E1 is the event associated with time_list1 and E2 with time_list2, then the returned time list expresses the time when both E1 and E2 were in progress. For example:

```
>When were John and Bill in Boston?
 John   starting t1 and ending t3
 Bill   starting t2 and ending t4
>When were both John and Bill in Boston?
 starting t2 and ending t3
```



This is how the function *time_intersection* is called

```
function  time_intersection{time_list1,time_list2:list_pointer;var:never:boolean
```

This function looks at all the intervals available in two time lists and intersect them one by one . Suppose time_list1 and time_list2 contain several time_intervals, then these intervals are retrieved one by one and from each list alternatively.

Hereby the procedure *time_intersec_int* is called to process two intervals at a time and output the *remainder time_list.* In other words, after every intersection, it is necessary to check if there is any remaining interval section to the right of the intersection interval, which can be intersected with a subsequent interval in the 'counterpart' timelist - this interval is called a *remainder timelist tl.* If there is none, then two fresh time intervals are retrieved for further intersection. If there is, then the next step would be to determine whether the remainder time list tl comes from timelist 1 or timelist 2. If tl comes from timelist 1 then it is intersected with the subsequent interval of timelist 2, and vice versa.

This is how *time_intersec_int* is called:

```
time_intersec_int(tl1,tl2,tl3,tl4,tl,tl5,bol);
```

Hereby only tl1,tl2 (the two intervals to be intersected) and tl3,tl4 (the following two intervals) contain relevant data when the following procedure is called. The other parameters are used for returning data only.

On return from the procedure, tl1 is the resulting intersection interval and probably tl2 too, i.e. if there is an additional interval preceding the 'main' interval; tl3 (timelist 1)

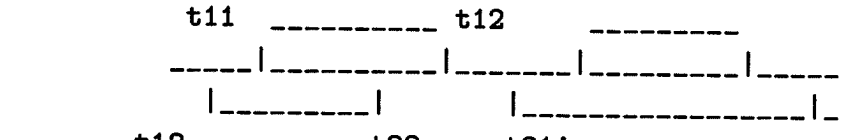
and tl4 (timelist 2) remain unchanged and tl5 contains the interval following the 'main' interval if there is such an additional interval; tl contains the remaining segment. Then the result is appended to the intersection list. If there is no more time interval in one of the timelists and the remainder timelist does not originate from this timelist, the intersection is done. Below is a brief description of time_intersec_int:

time_intersec_int

First, the four endpoints together with their respective flags are sorted in ascending order of time. The sorted sequence is needed for *recognition* procedure which returns the case number of these two intervals. Subsequently, *time_intersec_proc* is called.

```
time_intersec_proc(clsno,t1,t2,t3,t4,lef1,ref1,lef2,ref2,second1,second2);
```

It normally returns the intersection interval at t1,t2 and lef1,ref1. In cases where there are additional second intervals, say p-interval, it returns t3,lef2 and/or t4,ref2 as well, depending on whether these intervals occur before or after the 'main' interval. Hereby t1,t2,t3,t4 are real and NOT list pointers. On return from time_intersec_proc, the remaining timelist is computed. Here the next intervals have a role to play in determining tl and even to the extent that the intersection interval just has been found for the first two intervals needs to be adjusted. For example:



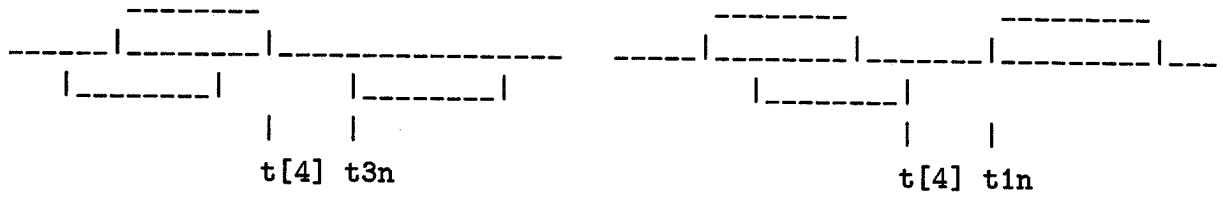

```
t12:end_in
t22:end_out
t21':end_out
all other end_point flags are assumed to be end_at
```

It is clear that the intersection of the first two intervals is the interval between t11:end_at and t12:end_in. But t21' being end_out can begin much earlier than t12 which means that the interval (t22,t12) is a uncertainty interval. So the intersection should be adjusted to t11:end_at,t22:p;whereas the remainder is also t22:p,t12:p, where p is an uncertainty endflag called *'possibly'*.

Now if there is no intersection, hence no output, remainder is the right interval i.e. the rightmost interval on the time scale.

tn is the left time endpoint of the next time interval to be considered. Depending on whether the last time interval section belongs to timelist 1 (upper) or timelist 2 (lower),

the respective (opposite) next time interval should be chosen: e.g.



Suppose the rightmost endpoint of the intersection interval is ti with endflag efi, ti is compared with tn to see if there is any overlap, if there is then the necessary adjustment is made to the intersection timelist to accomodate the remainder timelist. Below is a brief description of time_intersec_proc:

time_intersec_proc

It gets the respective algorithms for calculating intersection: the intervals for intersection will be:

The first interval will be tt1(or t1) with endflag le1 and tt2 (or t2) with endflag re1 and the second interval will be tt3 (or t3) with endflag le2 and tt4 (or t4) with endflag re2.

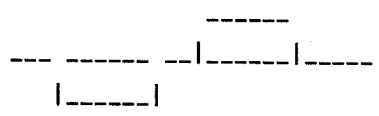
The procedure returns the 'main' intersection interval at tt1,le1 and tt2,re1

Sec1 returns a true only if there is a second interval preceding the 'main' intersection interval and it outputs the left endpoint of this second interval at tt3 and le2.

Sec2 returns a true only if there is a second interval following the 'main' intersection interval and it outputs the right endpoint of this second interval at tt4 and re2.

The example shown below demonstrates how two intervals are intersected in *time_intersec_proc*.

Example: Case 1



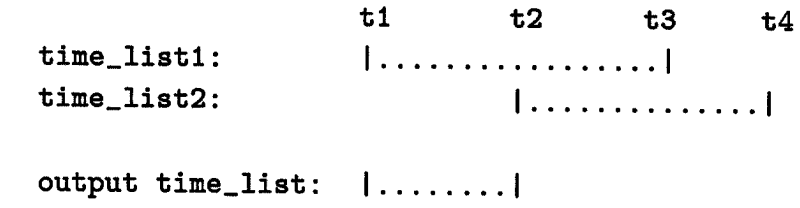
```
1: begin
   if (le1<>end_out) and (re2<>end_out) then begin
      tt1:=0.0;
      tt2:=0.0;
      end
   else begin
      {left enpoint of the intersection interval}
      if (le1=end_out) then begin
```

```
      if le2=end_out then begin
         tt1:=minus_infinity;
         le1:=p;
         end
      else begin {le2<>end_out}
         tt1:=tt3;
         le1:=p;
         end;
      end
   else begin {le1<>end_out and clearly re2=end_out}
      {tt1:=tt1}
      le1:=p;
      end;
   {right}
   if (re2=end_out) then begin
      if (re1=end_out) then begin
         tt2:=plus_infinity;
         re1:=p;
         end
      else if (re1<>end_out) then begin
        {tt2:=tt2}
         re1:=p;
         end
      end
   else begin {if re2<>end_out and clearly le1=end_out}
      tt2:=tt4;
      re1:=p;
      end;
   end;
   sec1:=false; {there is no need to set tt3,tt4 to 0 since they would
                 not be used back in time_intersec_int if sec is false}
   sec2:=false;
end;{case 1}
```

7.3.3 Time subtraction

Time subtraction returns the *relative compliment* of the given time lists, i.e., if E1 is the event associated with time_list1 and E2 with time_list2, then the returned time list expresses the time when E1 was in progress but E2 was not in progress. For example:

```
>When was John or Bill in Boston?
 John   starting t1 and ending t3
 Bill   starting t2 and ending t4
>When was John but not Bill in Boston?
 starting t1 and ending t2
```



The way two time lists are being processed are quite similar to that of time_intersection. Since time_list1 and time_list2 contain several time_intervals get these intervals out one by one and from each list alternatively.

Then the procedure time_subtract_int is called to process two intervals at a time and output the remaining segment. It returns the subtraction interval(s) in tl1 (and perhaps also tl2) and also the appropriate remainder interval for use in the next intersection.

Again ,similarly, time_subtract_int calls time_subtract_proc which produces the subtraction interval based on the case identified.

```
procedure time_subtract_proc{(clno:integer;var tt1,tt2,tt3,tt4:real;
                    var le1,re1,le2,re2:integer;var sec1,sec2:boolean)};
```

In this case, the intervals for subtraction are:

first interval: tt1(or t1) with endflag le1 and tt2 (or t2) with endflag re1

second interval: tt3 (or t3) with endflag le2 and tt4 (or t4) with endflag re2

Similarly, the procedure returns the 'main' intersection interval at tt1,le1 and tt2,re1

Sec1 returns a true if there is a second interval preceding the 'main' intersection interval and it outputs the left endpoint of this second interval at tt3 and le2

Sec2 returns a true if there is a second interval following the 'main' subtraction interval and it outputs the right endpoint of this second interval at tt4 and re2

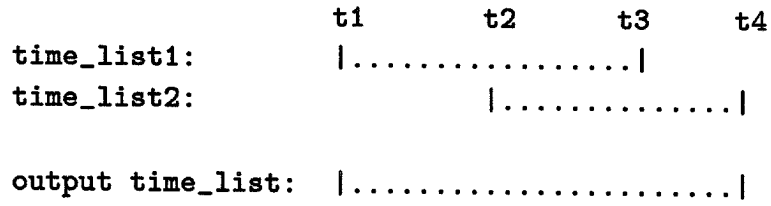
Again, using case 1, the following program illustrates how subtraction is done by *time_subtract_ proc*:

```
Example: Case 1

                              ------
              --- ------ --|------|-----
                  |------|
1: begin
     sec1:=false;
     sec2:=false;
     temp1:=tt1;
     temp2:=tt2;
     if (re2<>end_out) and (le1=end_out) then begin
        sec1:=true;
        tt1:=minus_infinity;
        tt2:=tt3;
        le1:=p;
        re1:=p;
        tt3:=tt4;
        tt4:=temp2;
        le2:=p;
        re2:=p;
        end
     else if (re2=end_out) then begin
        le1:=p;
        re1:=p;
        end;
   end;
```

7.3.4 Time union

Time union returns the *union* of the given time lists, i.e., if E1 is the event associated with time_list1 and E2 with time_list2, then the returned time list expresses the time when either E1 or E2 or both were in progress. For example:

```
>When was John or Bill in Boston?
 John  starting t1 and ending t3
 Bill  starting t2 and ending t4
>When was either John or Bill in Boston?
 starting t1 and ending t4
```



Time_list1 and time_list2 contain several time_intervals. These intervals are retrieved one by one in the order of time irrespective of which timelist it belongs to; this is different from the preceding two operations.

If the result of the time_union contains two intervals i.e. without continuity, then the old tl1 is stored away in the union time_list and the second (i.e. right) interval of the two intervals is used for further union with other intervals in the time_lists. If however, the result from time union is still a single continuous interval then this interval is used again for input into time_union_int for further union with other intervals in the time_lists.

It is interesting to note that even if one of the timelists is depleted, the time_union process should carry on till the time intervals in the other time list is depleted.

This is how the procedure time_union_int is called to process two intervals at a time.

```
time_union_int(tl1,tl2,tl3,tl4,bol);
```

Hereby tl1,tl2 (the two to be 'unioned') and tl3,tl4 (the following two intervals) contain relevant data when the following procedure is called.

On return from the procedure, tl1 is the resulting union interval and probably tl2 too ;tl3 (timelist 1) and tl4 (timelist 2) remain unchanged.

The result from time_union_int is appended to time_union_list as long as there is no continuity in it (i.e. tl2 contains something).

It is worth mentioning that there is always a union and there is no remainder in the case of time_union_int. As usual, Case 1 of time_union_proc is chosen to demonstrate how

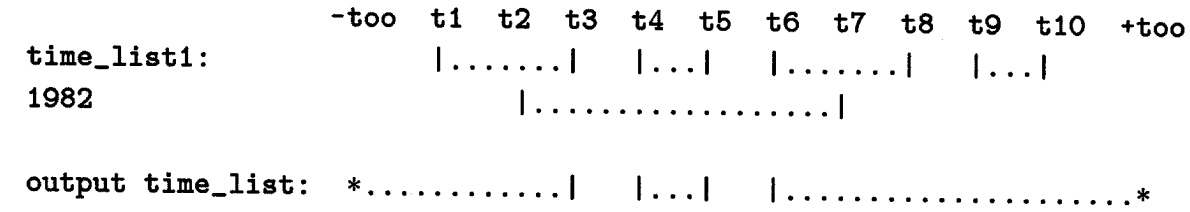
the union of two intervals are formed:

```
procedure time_union_proc(clno:integer;var tt1,tt2,tt3,tt4:real;
                    var le1,re1,le2,re2:integer;var bol:boolean);

1: begin
{similar to case 2}
{QQB
WRITELN('This is case 1 ');
{QQE}
bol:=false;
temp1:=tt1;
temp2:=tt2;
ltemp:=le1;
rtemp:=re1;
     if (re2<>end_out) and (le1<>end_out) then begin
        bol:=true; {no continuity}
        tt1:=tt3;
        tt2:=tt4;
        le1:=le2;
        re1:=re2;
        tt3:=temp1;
        tt4:=temp2;
        le2:=ltemp;
        re2:=rtemp;
        end
      else begin {(re2=end_out) or (le1=end_out)}
        tt1:=tt3;
        tt2:=tt2;
        le1:=p;
        re1:=p;
        end;
   end; {case 1}
```

7.3.5 Time extension

Time_list1 is the normal time list associated with an event. On the other hand, time_list2 is the list of the only relevant times. The desired output time list is the modification of time_list1 relative to time_list2 as *all time*. For example, if E1 is the event associated with time_list1, then the returned time list expresses the time when E1 was in progress relative to time_list2:

```
>When was John in Boston?
 starting t1 and ending t3
 starting t4 and ending t5
 starting t6 and ending t8
 starting t9 and ending t10
>When was John in Boston in 1982?
 ending t3
 starting t4 and ending t5
 starting t6
```



In time extension, the endpoint flags of the intervals from time_list2 are not considered.

First, the leftmost and rightmost endpoints of time_list2 are determined by getting the left endpoint of the first interval on time_list2 and the right endpoint of the last interval on time_list2, say t21 and t22. Then time extension on each of the intervals in time_list1 is performed with the constructed interval (t21,t22) and the result of which is stored in tl3. The following part of program demonstrates how time_extension is done with (t21,t22):

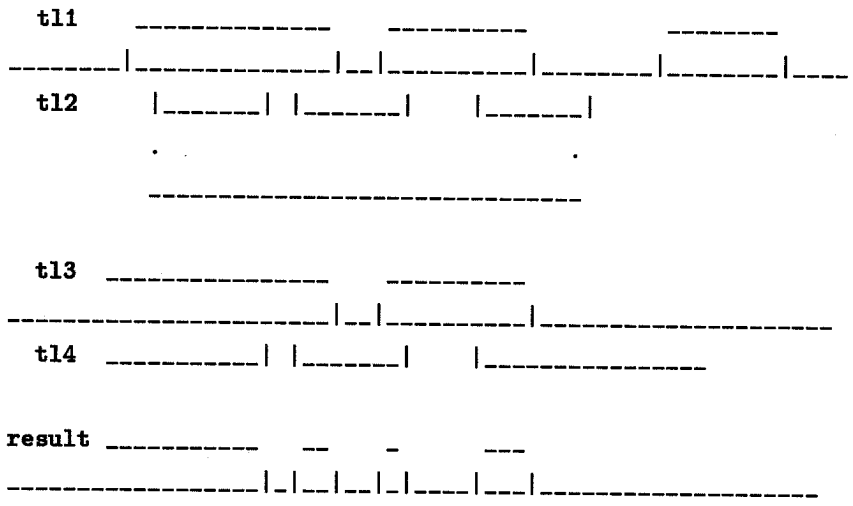
```
(* tinp1: timelist1 with endpoints [t11,t12]
   tinp2: timelist2                          *)
(* tt1 : left endpoint of extension interval
   tt2 : right endpoint of extension interval *)

tl3:=nil;       {set initial value of tl3 to nil}
```

```
while (tinp1<>nil) do begin
   t11:=tinp1^.f2.r;  {get the two endpoints from an interval on timelist1}
   t12:=tinp1^.f3.r;
   fetch_time_flag(tinp1^.flag,lef1,ref1); {decode the flag to obtain the
                                            endflags associated with t11, t12}
   if (t12<t21) then tinp1:=tinp1^.f1.1    {the interval not yet relevant}
   else if (t11>t22) then goto 10          {the interval no longer relevant}
   else begin
   {determining the left endpoint of the extension interval}
      if (t11<=t21) then begin
         tt1:=minus_infinity;
         lef:=end_out;
         end
      else if (t11>t21) then begin
         tt1:=t11;
         lef:=lef1;
         end;
   {determining the right endpoint of the extension interval}
      if (t12<t22) then begin
         tt2:=t12;
         ref:=ref1;
         end
      else if (t12>=t22) then begin
         tt2:=plus_infinity;
         ref:=end_out;
         end;
   {Then the result of the above is stored away one by one in tl3, according to
    the sequence of the intervals in timelist1}
   {Next we get the next interval of tl1 and repeat the whole process until the
    interval in timelist1 is no longer relevant.}
   end; { end of while loop}
```

Secondly, having completed the construction of tl3, the procedure starts constructing tl4 which is basically time_list2 except that the leftmost and rightmost endpoints are to be changed to minus and plus infinity respectively.

Finally, the result is obtained by intersecting tl3 and tl4.

Example:



7.4 The crunchers and the Is_Proc

After getting the time information into record and retrieving from it, the next interesting question would be the manipulation and coordination of these time information in conjunction with the processing of records. This occurs in the processing of noun phrases and verb phrases. In the case of noun phrases, a distinct feature of New World English is the handling of quantifiers which rely on the various procedures of *one_class_cruncher* for evaluating quantified expressions such as '*at least two ships*', '*how many boys*'. Hereby the time period during which the objects fall into the domain of the quantification is considered.

Another class of problems involves the verb phrases. *Is_proc* handles a simple question such as *Is John a student?.** The addition of time provides the mechanism of New World English with more dynamism, in the sense that in real world John would not be a student forever. Hence the various procedures in *is_proc* have been modified to include the processing of time information. Therefore, a statement such as *John is a student* is not always true; it is true starting from t_1 and ending at t_2, where t_1 and t_2 are two time points derived from the time information stored in the database.

* Initially, New World includes only the verbs '*to be*' and '*to have*'; further '*to have*' is transformationally reduced to '*to be*'. Other verbs are added by paraphrase.[9] Hence only *is_proc* needed to be considered.

7.4.1 New World English Data Structure

There is a difference between permanent and temporary data structure. By permanent data structure we mean those data that contain data on a relatively permanent basis, in that these data do not vanish after each application. On the other hand, temporary data structure exists only during sentence processing. Chapter 9.1 & 9.4 of [10] gives a detailed description of these two types of data structures. A brief summary on each of these data structures is included here:

The four principal data structures of an New World English data base are *classes,objects, attributes* and *relations (COAR structures).* Objects are the fundamental entities of the database which can be grouped into classes. Attributes and relations both associate objects with other objects. The difference between them is that attritutes are *single-valued* or is there is only one attribute to any object. For instance, in '*the age of a boy*' the attribute of the object *boy* is *age.* Relations are *multi-valued.* An object can have more than one relations. A good example for this is '*the destinations of the Alamo*', with destination being the relation and the Alamo the object.

There are five types of New World English data base records, namely object records, class records, attribute records, relation records and labeled classes. The labeled classes are data structures that arise during sentence processing. The principal function of labeled classes in New World English processing is to provide a uniform internal data structure. It makes possible the inclusion of '*labels*' and its use is closely linked to the way New World English handles quantifiers. By quantifiers we mean such words as '*all*' and '*some*'. New World English recognises 12 quantifiers among them *at least n, at most n, exactly n, how many* and *what, each etc.* (Ch.9.4.1 Quantifiers,[10]).

Labeled class records are a simple extension of class, attributes and relations. Each '*column*' in a labeled class is '*labeled*' with a type of quantifier. For example, the labeled class corresponding to the phrase '**Capital of some countries**' is:

	Capital of country
some	
USA	Washington D.C.
USSR	Moscow
.	.
.	.
Mexico	Mexico City
.	.
.	.

A labeled class record has essentially the same structure as a class or attribute record. If all New World record elements are considered as being a n-ary array, then for n=1 it reduces to a class record and n=2 a attribute or relation record. Further, n can be divided into two parts,

$$n = n_1 + n_2$$

For the application here, set

$$n_2 = 1$$

which refers to the last column or the members of the class. All members are of dimension one, in other words, there is no class of vectors here. Therefore, all n_1 elements are the labels of this member of the class. Time is included in the creation of labeled class since occasionally the membership of a certain element in a *labeled class* has validity period. Example: '*mayor of some city*'

	Mayor of City	
label:some		
Los Angeles	Bradley	
Chicago	Washington	(t_1/t_2) (t_3/t_4)
New York	Cory	(t_5/t_6)

where (t_i/t_j) is a time interval.

In the above example no time interval was given for Bradley being a mayor of Los Angeles. In this case, he is assumed to be *always* the mayor of Los Angeles. There are two time intervals given for the mayor of Chicago which is reasonable since a mayor can be elected for two non-consecutive terms.

7.4.2 The One Class Cruncher

There are three crunchers which are collectively referred to as the *one_class_cruncher*:

(1) the one class cruncher
(2) the member cruncher
(3) the boolean cruncher

One Class Cruncher

In the case of *one_class_cruncher*, the first labeled field of a labeled class is crunched out. The first labeled field corresponds to the field which is just before the last field. The field that is to be crunched out is supposed to be *unquantified* or *quantified* in the

following way: *how_many, at_least_n, at_most_n, exactly_n, all_but_n or coord.* A detailed explanation is given in chapter 9 of [10]. An example given for this is *'Teacher of at least two classes in each department'* with the following record:

each	at least 2	
Math	calculus	Jones
Math	geometry	Smith
Math	algebra	Smith
English	El1	Newton
English	El2	Whitney
English	Literature	Newton
Science	Physics	Jackson

which is crunched into

Math	Smith
English	Newton

In other words, the column with the various courses is crunched out. Basically, a procedure called *converse_record* switches the last two columns and then by comparing the first nn-1 fields (for nn see the explanation in the beginning of this section), it is possible to determine whether the current entry is still in the same *subclass.* By continually counting the number of such entries while still in the same subclass, it provides the answer to the various quantifiers mentioned. For instance, the first nn-1 fields are built into the output record if the number of such entries exceeds or equals to n for the case of *at_least_n.* Due to the inclusion of time, it is necessary to keep track of how many such entries are available in the same subclass during what time interval while comparing the entries. This is done by a procedure called *while_gen_timed* which will be elaborated below. Another procedure called *last_case_timed* answers the question connected to the quantifiers such as *at least n* by 'walking' down the *timed count list* tc produced by *while_gen_timed* and outputting the appropriate time lists at the end of each subclass.

while_gen_timed

The while_gen_timed procedure takes as parameter a new time list *tlnew* which is to be *'time_counted'* by intersecting *tlnew* with the existing accumulated *time count list tc.* tc is a time list which has in its flag the number of entries and its third field pointing to the a time list which is the time interval. In other words, during this time interval, there are this many entries in the record.

tc:	(lll,	flag,	L 1,	nil,	L 2)
L 1:	(lll,	flag-1,	L 3,	nil,	L 4)
.....					
L n:	(lll,	0,	nil,	nil,	L n+2)

L 2, L 4, ... L n+2 are the time intervals associated with the flag, which is reduced by one, each time we down the list.

tc is thus constantly being updated from the result of intersection with *tlnew*: while 'walking' down the list of tc, the intersection interval of the current time interval* and tlnew is pointed by the third field of a time list which is placed on top of tc and consequently tc gets built up. The remainder from tlnew is then used to intersect with the subsequent time list of tc with a lower count of entries(flag), so that this count would be incremented accordingly. The remainder from the current time interval remains in the same time_list. This is later combined with the newly created time list resulting from subsequent intersection between *tlnew* and tc. This is done by the editing part of the procedure. The code for the procedure is given in Appendix C.

last_case_timed

Last_case_timed produces output based on the newly created tc and the given quantifier. It 'walks' down the time list tc and creates *tltrue* and adds the time list to *tltrue* by means of *time_union*, if the flag satisfies the number given by the quantifier. Similarly it creates or updates *tlfalse* if it doesn't satisfy the quantifier. After going through the list tc, it then outputs the two time lists onto a record. In the case of *how_many*, it simply outputs the flag and the associated time list onto a record.

A case study for one_class_cruncher can be found in appendix D.

Member Cruncher

The member cruncher crunches off the *member field* (as opposed to *label field*) of the record which is quantified by *at_least_n*, *at_most_n*, *exactly_n*, *all_but_n*, *how_many* or *coord.* If the situation represented by the quantifier is satisfied by some entries of the record, then those entries that fail to satisfy the quantifier are crunched off if a boolean tf is true; on the other hand, if tf is false, then those entries that satisfy the quantifier are crunched off. If the last label field or the field before last field is quantified by *each* then the member field is replaced by a truth value. If it is quantified by *how_many* then it is replaced by a number. Otherwise the member field is crunched off entirely; only label fields of those entries which satisfy the quantifier are written onto the output record.

* The interval which corresponds to the current list pointed downward from tc

Using the same database as before, the function of member cruncher is illustrated here. In this case, the question is *Does every teacher have at least two classes* which is transformed into *'There are at least 2 classes of each teacher?'*.

label	**member**
each	*at least 2*
Jones	calculus
Smith	geometry
Smith	algebra
Newton	El1
Whitney	El2
Newton	Literature
Jackson	Physics

which is crunched into

Smith

Newton

The details of the *member cruncher* can be found in chapter 9 of [10]. The handling of time is similar to that of *one_class_cruncher* where *while_gen_timed* is used to build up the 'time count' list tc and the *last_case_timed* is used to check what time lists in tc satisfy the quantifiers. However, in the case of *member_cruncher*, the differentiation between *labeled class* and *unlabeled class* is made. Since there is no subclass in the case of *unlabeled class*, tc is built only once and *last_case_timed* is called at the end, whereas in the case of *labeled class* the labels are compared to determine the subclass and *while_ gen_timed* and *last_case_timed* are called for each subclass. Clearly, output follows each call of these two procedures/functions.

A case study for both member cruncher labeled class and ulabeled class can be found in Appendix D.

Boolean_Cruncher

Boolean cruncher is similar to member cruncher in the sense that the member field is crunched off entirely, only the label fields whose member field has a truth value are preserved. In other words, the member field in this case is assumed to be quantified by *truth_value*, and the last label field or the field before last should not be quantified by *each*. The *truth_value* can be of the type *true, false or some*. If the member field has a truth value of *true* then the associated time is stored in *tlt* by means of *time_union* with the old *tlt* and if the truth value is *false* then in tlf and if the truth value is *some* then in tls. After going through each entry in the record, *tltrue* is formed before output. This

time list should represent the time during which the statement is true for all entries, i.e. none of the entries has a member field that is false for this give period. Therefore, it is necessary to subtract the *time_union* of tlf and tls from tlt to obtain tltrue. The resulting *tltrue* is put away in a record if t_f is true. (The use of t_f is the same as that in member cruncher). The time interval *tlfalse* during which member field is false for all entries can be found using the same technique as *tltrue*, though the output of *tlfalse* is done only if t_f is false. Finally,*tlsome* is the intersection of *tlt* and *tlf* 'time_united' with tls and the output of which is independent of t_f.

In the case of *labeled class*, only *tlt* is obtained by forming time_union for each entry of the same subclass whose member field has a truth value of *true*. Nothing is done about *false* or *some*. At the end of each subclass, the resulting *tlt* is put away in a record together with the label fields, i.e. nn-1 fields which are typical of that subclass. Notice that *while_gen_time* and *last_case_timed* are not called in boolean cruncher.

All the above procedures/functions can be found in utyocc.pas, a semantic procedure for handling one_class_cruncher in New World.

7.4.3 The Is_Proc

Is_proc is one of thee central procedures of New World. It deals with verb phrases and is discussed in details in chapter 9.14 of [10]. Is_proc can be subdivided into six cases:

(1) Is there <noun_phrase>?
Example: *How many ships are there?*

(2) <noun_phrase> is <noun_phrase>?
Example: *What ships have length greater than 200 feet?*

(3) How <adjective> is <noun_phrase>?
Example: *How old is each employee?*

(4) <noun_phrase> is <adjective:+comp> than <noun_phrase>?
Example:*The salary of employees is greater than the salary of John?*

(5) Wh– is <noun_phrase>?
Example: *What is the destination of each ship?*

(6) (free variable) is <noun phrase>?
This occurs when a case noun relevant to a defined verb is missing in a particular use of that verb; i.e. the agent is missing.
Examples: *What ship carries each cargo type to each port?*
What cargo type is carried to each port?

Any of the above six forms can be negative. For instance, a question such as '*What*

cities are not capitals?' is allowed.

Is_proc starts by determining whether the clause is positive or negative. It then determines which of the six cases is applicable and in the process, identifies the agent and the object. Consider the simplest case: `<NP> is <NP>`. It is simple in the sense that neither the subject noun phrase nor the object noun phrase is labeled. Only the subject noun phrase can be quantified by *what, each, how many or coord.* The more complicated cases, where the subject and object noun phrases can be quantified, like *'What destination of each ship is home port of at least 2 ships?'*, are handled by the two_class_cruncher.

It should be noted that the application programmer can introduce new object type at this point, so that it will process the new object types under this case of `<NP> is <NP>`. Two such object types are the numbers and non-numbers. The subprocedure that handles the non-number objects is known as *eval_ np_np* and that of the number objects *eval_nu_nu.*

eval_np_np

Typical example: *'Are As Bs?','What As are Bs?','How many As are Bs?'*

If either or both the noun phrase records are empty, a diagnostic message is issued. So only cases where there are two non-empty records are considered. However, if both records are not quantified , the output is a boolean value. In the case where the quantifier of A is *what ,each or coord* then the output is a simple class (without label) and if the quantifier is *how many* the output is a real number.

The comparison of the two clases is explained in chapter 9.14 of [2]. and will not be elaborated here. Basically, it moves down the A record if the object(entry) in A is less than the object in B; and down the B record otherwise,, till all entries are finished. It is interesting to know how time is handle here. If either of the record contains time information, then the whole processing should include time. First, the case where both records are not quantified is considered. If two entries in subject and object record are equal then *tltrue* is formed by the intersection of the two time lists from each record. Also, forming the time subtraction of the object time list from the subject time list leads to a time interval *tlfalse.* This indicates the time interval during which there is an entry in the subject record but no entry in the object record. Besides, *tlfalse* is also formed by simply taking the time list from the subject record and forming 'time_union' with the old *tlfalse* when the two entries are not equal and the subject entry is less than the object entry. This is reasonable since all entries are sorted, and therefore this subject entry will not find a matching entry in the object record. Next the case where subject record is quantified by *what, each or coord* is considered. In this case, only when the two entries are matching that a time intersection of the two time list is performed and at the same time put away

in an output record together with the member field of the subject entry. Finally, there is a quantifier of *how_many* where *while_gen_timed* is called to 'time count' the intersection time list if the two entries are matching. At the end of the record comparison, *last_case_timed* is applied to count the number of entries in each time interval. For negative clause, the process is similar though *non-matching* entries are now of interest.

A case study for eval_np_np can be found in Appendix D.

eval_nu_nu

In this case, two number classes are considered. It could be that one of them is a real number but not both, since if both of them are real numbers and not classes, no time is processed. If both records are classes, then each entry in the subject record is compared with each entry in the object record. Clearly, if either of the record is not a class, then each entry of the record is compared with the real number only, which can be either a subject or an object.

Next, a case where both subject and object are classes is discussed. While 'walking' down the subject record, if it determines that the subject is an unknown number, then the associated subject time list forms *tlunkn*. If the corresponding object entry is an unknown number * then the associated object time list forms *tluk*. If none of the above cases appears, it gets the intersection of subject and object time list to form *tltrue* if the two numbers satify the relation between them. Similarly *tlfalse* is formed if the relation is not satisfied.

If either the subject or the object is a real number but not both, then it 'walks' down the record that is a class. Suppose the subject is a class, and the object is a real number, *tlunkn* is formed if the subject entry is unknown.If the subject entry number matches the object number, then *tltrue* is formed; If they do not match *tlfalse* is formed. In both cases, the only class can be a labeled class, and the result is sent to output at the end of each subclass. This is done by calling *put_away_time* which outputs a truth value together with the labels and the associated time list. If we subtract *tlfalse* from *tltrue* we get a time list that is to go with the truth value *true*; and if we subtract *tltrue* from *tlfalse*, we get a time list which is associated with *false*; and finally if we intersect *tltrue* and *tlfalse* we get a time list that is to go with *som* . These three cases presume that tlunkn is never set. However, if tlunkn is set we get a truth value of *unk*.

A study which covers eval_nu_nu, boolean_cruncher and one_class_cruncher can be found in Appendix D.

---

* In New World, a specific number, namely -10^{-37} is specified as the '*unknown number*'. All the arithmetic procedures know about it; thus $3 + unknown = -10^{-37}$ and $cos(unknown) = -10^{-37}$.

8. Remaining problems

8.1 Extensions that can be made within the New World environment

Each data record has two significance units (sig_units) associated with it. The sig_units are set on the basis of the input data and they are used for output of data. In New World, these sig_units are declared as global variables and can be modified as the application in an New World environment proceeds. Thus, it would be desirable to store the sig_units which are particular to the input data in the records. This can be done by storing the low and high sig_units in its record_header_extension. The addition of new time data to the record may modify the low sig_units downward but not up ,and the high sig_units upward but not down.

At the present time, no distinction is made between duration and single occurence. Thus from 'John was in London from June 1, 1968 to October 20, 1969' , one cannot tell, except from context, whether he was there throughout the interval or at some single time within the interval. The two end point flags occupy 4 bits but only 9 combinations out of the $2^4 = 16$ combinations are used. Thus a duration flag could also be included, distinguishing between the above two possiblities. The syntax for time could then be extended into words such as '*throughout*' etc., thus considerably increasing the breath of time information recorded in the data which are available for use in applications.

These two extensions, although entailing considerable changes in relevant semantic procedures, could be made without basic changes in data structures.

8.2 Extensions requiring more radical changes

In this version of the implementation of time in New World, no information about the frequency of a certain object belonging to a class or an entity being the attribute of an object can be provided by the system. Queries such as '*How frequently was John in London?*' and '*How often was coal the cargo of Maru?*' would not be processed by the present system. Such extension to the system requires the implementation of the flag for distinguishing between '*throughout the interval*' or '*certain time point in the past*' as mentioned in 8.1. Also, extra facilities must be provided for storing such information as '*Five times last year*', in other words, additional capability of storing the numerical frequency 5.

Another extension worth considering is the handling of connectives such as 'when' and 'while'. For example, '*John was in London when Mary arrived there*' does not provide any information about when exactly Mary was in London. Therefore the system has to be modified to include such 'relative' information.

Finally, it would be desirable to make some distinctions between the various tenses in English, and in particular, the handling of past tense and present perfect tense. For example, '*John was in London starting June 3, 1967*' and '*John has been in London since June 3, 1967*' should carry different meanings since for the latter, it would be safer to assume that John is still in London to this date. Besides, the present system does not deal with the time information carried by sentences like '*John has visited London*'. However, this entails further research into the English tenses and its associated semantics.

8.3 Some alternative considerations relevant to the implementation of time in data base systems

Some alternative considerations with respect to the implementation of time information in database can be found in [1] and [2]. In [1], James Allen introduced an interval-based temporal logic to tackle the problem of *imprecision of scale* and *uncertainty* in representing time intervals. In [2], Siegfried Günther introduced the notion of ordered 6-tuples representing the 6 temporal relations between the starting and ending points of two intervals. For example, the temporal relation between the starting points of these two intervals is captured as one of the elements of the 6-tuples, and it can assume the value of *'greater than'*, *'less than'*, *'greater than or equal to'*, *'less than or equal to'* or *'unknown'*. A time network could be thus established to handle the problems with respect to the manipulation of time lists discussed in 7.3. Further, due to the fact that temporal statements in natural languag are often vague and fuzzy, it would be interesting to explore the possiblities of using fuzzy set theory to handle the uncertainty encountered in representing time information [14] [15]. For example, it would be possible to handle such ambiguous information as *'Mary arrived in London in around the 1970's'* using fuzzy set theory.

References

[1] Allen,J.F., "Maintaining Knowledge about Temporal Intervals", CACM Vol 26, No.11, Nov 1983.

[2] Günther, S., "Zur Repräsentation Temporaler Beziehungen in SRL", KIT-Report 21, Technische Universität Berlin, Sep 1984.

[3] Hornstein, Norbert, "Towards a Theory of Tense", Linguistic Inquiry Vol 8 No 3 Summer 1977. p.521-557

[4] McCawley, J.D., "Tense and Time Reference in English", in C.J.Fillmore and D.T.Langendoen, eds., *Studies in Linguistic Semantics* Holt, Rinehart and Winston, New York.

[5] Harper M.P., Charniak E., "Time and Tense in English", Proceeding of the 24th meeting of ACL, 1986.

[6] Yip,K.M., "Tense, Aspect and the Cognitive Representation of Time",Proc. of IJCAI 85, 806-814.

[7] Bolour,A., Anderson,T.L., Dekeyser,L.J., Wong,H.K.T., " The Role of Time in Information Processing, A Survey", SIGMOD, April 1982.

[8] Hafner,C., "Semantics of Temporal Queries and Temporal Data", Proceeding of the 23rd meeting of ACL, 1985.
The Role of Time in Information Processing, A Survey", SIGMOD, April 1982.

[9] Thompson,B.H. and Thompson F.B., "Introducing ASK, A Simple Knowledgeable System", in Proceedings, Conf. on Applied Natural Language Processing, 1983, pp 17-24.

[10] Thompson,B.H. and Thompson F.B., "ASK, A Simple Knowledgeable System", a major documentation of the system, to be published by Springer Verlag.

[11] Sanouillet,R., "ASK French-A French Natural Language Syntax", MS Thesis, 1984, Caltech.

[12] Thompson,B.H. and Thompson F.B., "ASK as Window to the World", in Proceeding IEEE Int'l Conf. on Systems, Man & Cybernetics, 1984.

[13] Poh, H.L., "The Understanding of Time", Term Paper, Psycholinguistics,Caltech, June, 1986.

[14] Schmucker, K.J., "Fuzzy sets, Natural Language Computations, and Risk Analysis", published by Computer Science Press, 1984.

[15] Sheng R.L., "A Linguistic Approach to Temporal Information Analysis", Ph.D. Thesis, Computer Science Div., Dept. of EE & CS, University of California, Berkeley.

Appendix A

Time Syntax Rules

(1) Lexical Time Rules

```
RULE January
<time:+lit+byr+td+smo> => "January"
LEX 1

RULE February
<time:+lit+byr+td+smo> => "February"
LEX 2

RULE March
<time:+lit+byr+td+smo> => "March"
LEX 3

RULE April
<time:+lit+byr+td+smo> => "April"
LEX 4

RULE May
<time:+lit+byr+td+smo> => "May"
LEX 5

RULE June
<time:+lit+byr+td+smo> => "June"
LEX 6

RULE July
```

```
<time:+lit+byr+td+smo> => "July"
LEX 7

RULE August
<time:+lit+byr+td+smo> => "August"
LEX 8

RULE September
<time:+lit+byr+td+smo> => "September"
LEX 9

RULE October
<time:+lit+byr+td+smo> => "October"
LEX 10

RULE November
<time:+lit+byr+td+smo> => "November"
LEX 11

RULE December
<time:+lit+byr+td+smo> => "December"
LEX 12

RULE Jan
<time:+lit+byr+td+smo+abr> => "Jan"
LEX 1

RULE Feb
<time:+lit+byr+td+smo+abr> => "Feb"
LEX 2

RULE Mar
<time:+lit+byr+td+smo+abr> => "Mar"
LEX 3

RULE Apr
```

```
<time:+lit+byr+td+smo+abr> => "Apr"
LEX 4

RULE June
<time:+lit+byr+td+smo+abr> => "Jun"
LEX 6

RULE Jul
<time:+lit+byr+td+smo+abr> => "Jul"
LEX 7

RULE Aug
<time:+lit+byr+td+smo+abr> => "Aug"
LEX 8

RULE Sep
<time:+lit+byr+td+smo+abr> => "Sep"
LEX 9

RULE Sept
<time:+lit+byr+td+smo+abr> => "Sept"
LEX 9

RULE Oct
<time:+lit+byr+td+smo+abr> => "Oct"
LEX 10

RULE Nov
<time:+lit+byr+td+smo+abr> => "Nov"
LEX 11

RULE Dec
<time:+lit+byr+td+smo+abr> => "Dec"
LEX 12

RULE Jan.
```

```
<time:1-abr> => <time:+lit+byr+td+smo+abr> "."
PRE 1 no_op1

RULE AM
<time:+lit+ampm+td> => "AM"
LEX 0

RULE am
<time:+lit+ampm+td> => "am"
LEX 0

RULE PM
<time:+lit+ampm+td> => "PM"
LEX 1

RULE pm
<time:+lit+ampm+td> => "pm"
LEX 1

RULE Sunday
<time:+lit+day+td> => "Sunday"
LEX 0

RULE Monday
<time:+lit+day+td> => "Monday"
LEX 1

RULE Tuesday
<time:+lit+day+td> => "Tuesday"
LEX 2

RULE Wednesday
<time:+lit+day+td> => "Wednesday"
LEX 3

RULE Thursday
```

```
<time:+lit+day+td> => "Thursday"
LEX 4

RULE Friday
<time:+lit+day+td> => "Friday"
LEX 5

RULE Saturday
<time:+lit+day+td> => "Saturday"
LEX 6

RULE day
<time:+lit+ut+sdy+dy+td> => "day"
LEX 0

RULE week
<time:+lit+ut+sdy+td> => "week"
LEX 1

RULE month
<time:+lit+ut+smo+td> => "month"
LEX 2

RULE year
<time:+lit+ut+syr+td> => "year"
LEX 3

RULE "minute"
<time:+lit+ut+smn+td> => "minute"
LEX 4

RULE "hour"
<time:+lit+ut+smn+td> => "hour"
LEX 5

RULE "second"
```

```
<time:+lit+ut+sse+td> => "second"
LEX 6


RULE "centuries"
<time:+lit+syr+ut+td> => "centuries"
LEX 7

RULE "century"
<time:+lit+syr+ut+td> => "century"
LEX 7

RULE "decade"
<time:+lit+syr+ut+td> => "decade"
LEX 8

RULE o'clock
<time:+lit+oclock+td> => "o'clock"
LEX 0

RULE BC
<time:+lit+bcad+td> => "BC"
LEX 1

RULE AD
<time:+lit+bcad> => "AD"
LEX 0

RULE "today"
<time:+lit+sdy> => "today"
LEX 1

RULE "tomorrow"
<time:+lit+sdy> => "tomorrow"
LEX 2
```

```
RULE "yesterday"
<time:+lit+sdy> => "yesterday"
LEX 3

RULE "now"
<time:+sse+lit> => "now"
LEX 0

RULE "noon"
<time:+lit+td+minute> => "noon"
LEX 0

RULE "tonight"
<time:+lit+tongt> => "tonight"
LEX 0

RULE "morning"
<time:+lit+morn+td> => "morning"
LEX 4

RULE "night"
<time:+lit+ut+td> => "night"
LEX 0

RULE "afternoon"
<time:+lit+ut+afn+td> => "afternoon"
LEX 5

RULE "evening"
<time:+lit+ut+even+td> => "evening"
LEX 6

RULE "Eve"
<time:+lit+eve> => "Eve"
LEX 0
```

```
RULE "Christmas"
<time:+lit+sea> => "Christmas"
LEX 4

RULE "New year"
<time:+lit+sea> => "New Year"
LEX 5

RULE "Thanks Giving"
<time:+lit+sea> => "Thanksgiving"
LEX 6

RULE "Easter"
<time:+lit+sea> => "Easter"
LEX 7

RULE "Spring"
<time:+lit+sea> => "spring"
LEX 0

RULE "Summer"
<time:+lit+sea> => "summer"
LEX 1

RULE "fall"
<time:+lit+sea> => "fall"
LEX 2

RULE "Autumn"
<time:+lit+sea> => "autumn"
LEX 2

RULE "Winter"
<time:+lit+sea> => "winter"
LEX 3
```

```
RULE seasons
<time:+sdy> => <time:+lit+sea>
PRE 349 season_proc

RULE "this"
<time:+lit+period+this> => "this"
LEX 1

RULE "last"
<time:+lit+period> => "last"
LEX 2

RULE "next"
<time:+lit+period> => "next"
LEX 3

RULE "later"
<time:+lit+lat> => "later"
LEX 0

RULE "beginning"
<time:+lit+eb+beg> => "beginning"
LEX 1

RULE "starting"
<time:+lit+eb+beg> => "starting"
LEX 1

RULE "ending"
<time:+lit+eb+end> => "ending"
LEX 2

RULE "terminating"
<time:+lit+eb+end> => "terminating"
LEX 2
```

```
RULE beg/end at/in/on
<time:+lit+eb+end> => <time:+lit+eb> " " <time:+lit+prep+aio-interval>
PRE 1 no_op1

RULE "before"
<time:+lit+prep+before> => "before"
LEX 1

RULE on or before
<time:+lit+prep> => <time:+lit+prep+aio> " or " <time:+lit+before>
LEX 1

RULE "after"
<time:+lit+prep+after> => "after"
LEX 2

RULE on or after
<time:+lit+prep> => <time:+lit+prep+aio> " or " <time:+lit+after>
LEX 2

RULE "from"
<time:+lit+prep+from> => "from"
LEX 3

RULE "since"
<time:+lit+prep+from> => "since"
LEX 3

RULE "to"
<time:+lit+prep+to> => "to"
LEX 4

RULE "up to"
<time:+lit+prep+to> => "up to"
LEX 4
```

```
RULE "until"
<time:+lit+prep+until> => "until"
LEX 5

RULE "till"
<time:+lit+prep+till> => "till"
LEX 5

RULE "at"
<time:+lit+prep+aio> => "at"
LEX 6

RULE "in"
<time:+lit+prep+aio> => "in"
LEX 7

RULE "on"
<time:+lit+prep+aio> => "on"
LEX 8
```

(2) Rules for Time Statements

```
RULE "June 3"
<time:1-lit-smo+sdy-byr> => <time:+td+smo+lit+byr> " " <whole_number>
POST 304 con_mday1

RULE "3rd of June"
<time:2-lit-smo+sdy-byr> => <ordinal> " of " <time:+td+smo+lit+byr>
POST 305 con_mday2

RULE 10:25
<time:+smn+td> => <whole_number> ":" <whole_number>
POST 309 con_time1

RULE 10:25 PM
<time:+smn+td+ampm> => <time:+smn+td-prep-ampm> " " <time:+lit+ampm>
```

```
POST 310 con_time2

RULE 10:25:50
<time:+td+sse> => <whole_number> ":" <whole_number> ":" <whole_number>
POST 314 con_sec1

RULE 10:25:50 PM
<time:1+ampm> => <time:+td+sse-prep-ampm> " " <time:+lit+ampm>
POST 315 con_sec2

RULE five o'clock
<time:+td+smn> => <whole_number> " " <time:+lit+oclock>
POST 316 con_hour1

RULE "now"
<time:1-lit> => <time:+sse+lit>
PRE 341 con_now

RULE "today/yesterday/tommorrow"
<time:1-lit-td> => <time:+lit+sdy>
PRE 340 con_day0

RULE "June 1979" but not "June 3 1979" or "3rd of June 1979"
<time:1-lit-td> => <time:+smo+td+byr-sdy> " " <whole_number>
POST 339 con_month1

RULE "80 BC or 1900 AD"
<time:+byr+bcad-td> => <whole_number> " " <time:+lit+bcad>
POST 338 con_bcad

RULE "June of 1970"
<time:1-lit-td> => <time:+td+byr> " of " <whole_number>
POST 306 con_myear

RULE "June 3 of 1979" or "3rd of June of 1979"
<time:1-lit-td> => <time:+sdy-smo+td> " " <preposition:+of> " "
```

```
                    <whole_number>
POST 307 con_date1

RULE "June 3, 1979" or "3rd of June, 1979"
<time:1-lit-td> => <time:+sdy-smo+td> <pct:+comma> <whole_number>
POST 307 con_date1

RULE "3-June-1980"
<time:+sdy-td> => <whole_number> "-" <time:+td+smo+lit> "-" <whole_number>
POST 308 con_date3

RULE "3/June/1980"
<time:+sdy-td> => <whole_number> "/" <time:+td+smo+lit> "/" <whole_number>
POST 308 con_date3

RULE "3 June 1980"
<time:+sdy-td> => <whole_number> " " <time:+td+smo+lit> " " <whole_number>
POST 308 con_date3

RULE "mm/dd/yy" if mm>12 then mm=>dd and dd=>mm i.e. dd/mm/yy (European)
<time:+sdy> => <whole_number> "/" <whole_number> "/" <whole_number>
POST 311 con_date4

RULE "this/next/last <month_name>"
<time:-td+smo> => <time:+lit+period> " " <time:+lit+smo-ut-num>
POST 312 con_month2


RULE "this/next/last year/day/month/week/decade/century"
<time:2-lit-td-ut> => <time:+lit+period> " " <time:+lit+ut>
POST 313 con_time_unit1

RULE "this/next/last <weekday>"
<time:+sdy+byr-td-prep> => <time:+lit+period> " " <time:+lit+day>
POST 317 con_wkday1
```

```
RULE "<weekday> last/next/last week"
<time:+byr-td+sdy-prep> => <time:+lit+day> " " <time:+lit+period>
                                          " " <time:+lit+ut+sdy-dy>
POST 318 con_wkday2
```

(3) Rules for time statements with time/date and/or prepositions at a particular time point

```
RULE "in" 1985
<time:+syr+interval> => "in " <whole_number>
PRE 348 in_year

RULE "year" 1985
<time:+syr+num> =>  <time:+lit+syr-num> " " <whole_number>
PRE 2 no_op2

RULE "the " year 1985
<time:+syr> => "the " <time:+syr+num>
PRE 1 no_op1

RULE "in " year 1985
<time:+syr+interval> => "in " <time:+syr+num>
POST 348 in_year

RULE "in the " year 1985
<time:+syr+interval> => "in the " <time:+syr+num>
POST 348 in_year

RULE "on" <date>
<time:1-lit+interval> => "on " <time:-td+sdy-prep-int-num>
POST 319 on_date1

RULE "at" <time:+smn>
<time:1+at> => "at " <time:+td+smn-int-num-prep-lit>
POST 320 at_min1
```

```
RULE "at" <time:+sse>
<time:1+at> => "at " <time:+td+sse-int-num-prep-lit>
POST 321 at_sec1

RULE "at" <time:+smn or +sse> "on" <date>
<time:1-lit-td> => <time:+at> " on " <time:-td-prep+sdy>
POST 322 at_on_time_date

RULE <time: +sse> on <date>
<time:1-lit-td> => <time:+td+sse-prep> " on " <time:-td-prep+sdy>
POST 323 sec_on_date


RULE <time:+smn> on <date>
<time:1-lit-td> => <time:+td+smn-prep-int-num> " on " <time:-td-prep+sdy>
POST 324 min_on_date


RULE  "5:00 (PM) today/yesterday/tomorrow"
<time:1-lit-td+ytt> => <time:+smn+td-prep> " " <time:+lit+sdy>
POST 325 min_date1

RULE "5:00:23 (PM) today/yesterday/tomorrow"
<time:1-td-lit+ytt> => <time:+sse+td-prep> " " <time:+lit+sdy>
POST 326 sec_date1

RULE "at 5:00:00 today/tomorrow/yesterday"
<time:1-td+at-lit> => "at " <time:+sse-td-prep+ytt>
POST 1 no_op1

RULE "at 5:00 today/tomorrow/yesterday"
<time:1-td+at-lit> => "at " <time:+smn-td-prep+ytt>
POST 1 no_op1

RULE n days/weeks/... from a given time
<time:2-lit-td+int+num> => <whole_number> " " <time:+lit+ut> " from "
```

```
                              <time:-td-num-int-lit>
POST 331 n_unit_from_time
```

(4) Rules that put prepositions on a time point

```
*** >--- end_in

RULE "after" <time:-td>
<time:2+beg+prep> => <time:+lit+after> " " <time:-lit-td-prep>
POST 335 after_time

RULE starting after <time:-td>
<time:2-lit+beg+prep> => <time:+lit+eb+beg> " " <time:+lit+after> " "
                         <time:-lit-td-prep>
POST 335 after_time

*** |--- end_at

RULE beginning <time>
<time:2+beg+prep> => <time:+lit+eb+beg> " " <time:-td-eb-lit-beg-end-interval>
POST 350 endpoint_proc

RULE starting "in" 1985
<time:1-lit-eb+syr> => <time:+lit+eb+beg> " in " <whole_number>
PRE 353 ending_in_year

*** <--- end_out

RULE from <time>
<time:2+beg+prep> => <time:+lit+from> " " <time:-td-eb-lit>
POST 327 from_time

RULE starting before <time>
<time:2-lit+beg+prep> => <time:+lit+eb+beg> " " <time:+lit+before> " "
                       <time:-lit-td-prep>
```

```
POST 327 from_time

*** ---< end_in

RULE "before" <time>
<time:2+end+prep> => <time:+lit+before> " " <time:-lit-td-prep>
POST 334 before_time

RULE ending before <time:-td>
<time:2-lit+end+prep> => <time:+lit+eb+end> " " <time:+lit+before> " "
                         <time:-lit-td-prep>
POST 334 before_time

*** ---| end_at

RULE ending <time>
<time:2+end+prep> => <time:+lit+eb+end> " " <time:-td-eb-lit-beg-end-interval>
POST 350 endpoint_proc

RULE ending "in" 1985
<time:1-lit-eb+syr> => <time:+lit+eb+end> " in " <whole_number>
PRE 353 ending_in_year

*** ---> end_out

RULE until <time>
<time:2+end+prep+tu> => <time:+lit+until> " " <time:-td-eb-lit>
POST 328 until_time

RULE till <time>
<time:2+end+prep+tu> => <time:+lit+till> " " <time:-td-eb-lit>
POST 328 until_time

RULE to <time>
<time:2+end+prep+tu> => <time:+lit+to> " " <time:-td-eb-lit>
POST 328 until_time
```

```
RULE ending after <time>
<time:2-lit+end+prep> => <time:+lit+eb+end> " " <time:+lit+after> " "
                      <time:-lit-td-prep>
POST 328 until_time
```

(5) Rules that deal with time intervals

```
RULE "last/this/next" n "days/week/month/century/decade" e.g. next 2 months
<time:3-lit-td+num> => <time:+lit+period> " " <whole_number> " " <time:+lit+ut>
POST 329 fix_n_time_unit

RULE "the" last/next n days/weeks/months/etc e.g. the last 3 months
<time:3-lit+num-td> => "the " <time:+lit+period-this> " " <whole_number>
                                                    " " <time:+lit+ut>
POST 329 fix_n_time_unit

RULE in "the" last/next n days/weeks/etc.
<time:3-lit+num+prep+int> => "in the " <time:+lit+period-this> " "
                                       <whole_number> " " <time:+lit+ut>
POST 329 fix_n_time_unit

RULE <time:+beg> and <time:+end-tu>
<time:+interval> => <time:-td+beg> " and " <time:-td+end-tu>
POST 330 from_time_to_time

RULE <time:+beg> to/until <time:+end-tu>
<time:+interval> => <time:-td+beg> " " <time:-td+end+tu>
POST 330 from_time_to_time

RULE in n "day/week/.." (* same as bewteen now and n day/week/..+now *)
<time:2-lit+num+prep+int> => "in " <whole_number> " " <time:+lit+ut>
POST 332 in_n_unit

RULE "n days/weeks/years/..." later
<time:2-lit-td-ut+num> => <whole_number> " " <time:+lit+ut> " " <time:+lit+lat>
```

```
POST 333 n_unit_later


RULE "after <time> before <time>" eg. after five o'clock before 5:30 pm
<time:+int+prep> => <time:+after> " " <time:+before>
POST 336 after_time_before_time
```

(6) Rules for processing sentence

```
RULE sentence => time?
<left_delimiter> <sentence> =>
         <left_delimiter> <time:-td-lit> <pct:+question_mark>
POST 301 time_answer_proc

RULE put time at end of verb phrase
<verb_phrase:1+fti> => <verb_phrase:-fti> " " <time:-td-lit>
* e.g., "[The Tokyo Maru carried coal] in May 1982"
SYN 302 time_case_proc

RULE put time at beginning of verb phrase
<verb_phrase:1+fti> => <time:-td-lit+prep> <pct:+comma> <verb_phrase:-fti>
* e.g., "beginning in June 1985, [the Tokyo Maru carried coal] "
*       "Before May 1978, did the Tokyo Maru carry coal?"
SYN 302 time_case_proc

RULE time modification of noun phrase
<noun_phrase:2+time> => <time:-td-prep-lit> " " <noun_phrase:-pof>
POST 303 time_noun_proc
```

(7) Miscellaneous rules

```
RULE Singular nouns to plural
<time:1> => <time:+lit+ut> "s"
SYN 337  sin_to_plu

RULE in "June 1979"
<time:1+prep> => "in " <time:-td-prep+smo-lit-num>
POST 1 no_op1
```

```
RULE beginning and ending
<time:1-lit-td+prep> => "between " <time:-td-eb-prep> " and " <time:-td-eb-prep>
POST 354 between_time
```

Appendix B

Addition of timed data to record and output from record

```
>class:student
 The new class student has been added.
>individual:Bailey,Bain,Baird,Baker,Bakus,Baldwin,Balen
 The following new individuals have been added:
   Bailey  Bain  Baird  Baker  Bakus  Baldwin  Balen
>individual:Ball,Bancroft,Banister,Barbosa,Barber,Banks
 The following new individuals have been added:
   Ball  Bancroft  Banister  Barbosa  Barber  Banks
>Bailey is a student.
 Bailey has been added to the class student.
>Baker is a student before May 1, 1985.
 Baker has been added to the class student.
>Baird is a student ending May 1, 1985.
 Baird has been added to the class student.
>Bain is a student until May 1, 1985.
 Bain has been added to the class student.
>Balen is a student after May 1, 1985.
 Balen has been added to the class student.
>Baldwin is a student beginning May 1, 1985.
 Baldwin has been added to the class student.
>Bakus is a student from May 1, 1985.
 Bakus has been added to the class student.
>Banister is a student before October 1, 1988.
 Banister has been added to the class student.
>Bancroft is a student ending October 1, 1988.
 Bancroft has been added to the class student.
>Ball is a student until October 1, 1988.
 Ball has been added to the class student.
>Banks is a student after October 1, 1988.
```

```
 Banks has been added to the class student.
>Barber is a student beginning October 1, 1988.
 Barber has been added to the class student.
>Barbosa is a student from October 1, 1988.
 Barbosa has been added to the class student.
>
>individual:Beckwith,Beeson,Behn,Behrens,Beisel,Bell,Belino,Beman,Bender
 The following new individuals have been added:
   Beckwith Beeson Behn Behrens Beisel Bell Belino Beman Bender
>Beckwith is a student from May 1, 1985 until October 1, 1988.
 Beckwith has been added to the class student.
>Beeson is a student from May 1, 1985 to before October 1, 1988.
 Beeson has been added to the class student.
>Behn is a student from May 1, 1985 and ending October 1, 1988.
 Behn has been added to the class student.
>Behrens is a student after May 1, 1985 until October 1, 1988.
 Behrens has been added to the class student.
>Beisel is a student after May 1, 1985 to before October 1, 1988.
 Beisel has been added to the class student.
>Bell is a student after May 1, 1985 and ending October 1, 1988.
 Bell has been added to the class student.
>Belino is a student starting May 1, 1985 until October 1, 1988.
 Belino has been added to the class student.
>Beman is a student starting May 1, 1985 to before October 1, 1988.
 Beman has been added to the class student.
>Bender is a student starting May 1, 1985 and ending October 1, 1988.
 Bender has been added to the class student.
>what is a student?
 There are 22 lines in your answer.
>Which lines do you want? End of file time.stu2.
>all
 Balen      from on or after May 1, 1985
 Baldwin    starting May 1, 1985
 Bakus      from on or before May 1, 1985
 Baker      to on or before May 1, 1985
 Baird      ending May 1, 1985
```

Bain	to on or after May 1, 1985
Bailey	
Banks	from on or after October 1, 1988
Barber	starting October 1, 1988
Barbosa	from on or before October 1, 1988
Banister	to on or before October 1, 1988
Bancroft	ending October 1, 1988
Ball	to on or after October 1, 1988
Bender	starting May 1, 1985 to ending October 1, 1988
Beman	starting May 1, 1985 to on or before October 1, 1988
Belino	starting May 1, 1985 to on or after October 1, 1988
Bell	on or after May 1, 1985 to ending October 1, 1988
Beisel	on or after May 1, 1985 to on or before October 1, 1988
Behrens	on or after May 1, 1985 to on or after October 1, 1988
Behn	on or before May 1, 1985 to ending October 1, 1988
Beeson	on or before May 1, 1985 to on or before October 1, 1988
Beckwith	on or before May 1, 1985 to on or after October 1, 1988

Note:

end_at: starting, ending
end_in: on or after, to on or before
end_out: on or before, to on or after

Appendix C

Procedures for 'Time Counting'

```
   procedure while_gen_timed{(tlnew:list_pointer;
              var tc:list_pointer)};
      var tcc,tccc,tlold,tl,temp:list_pointer;
          still,never:boolean;
      begin
{QQB
WRITELN('   at top of generate record loop, tc:');
LIST_LIST(tc);
{QQE}
         tcc:=tc;
         still:=true;
         while (tcc <> nil) and still do begin
            tlold:=tcc^.f3.1;
{QQB
WRITELN('WHILE_GEN_TIMED: top of while tcc <> nil loop, tcc:');
{QQE}
            tl:=time_intersection(tlold,tlnew,never);
            if not never then begin
               temp:=new_list(111,(tcc^.flag + 1));
               temp^.f1.1:=tc;
               temp^.f2.1:=nil;
               temp^.f3.1:=tl;
               tc:=temp;
               end;
            tl:=time_subtraction(tlold,tlnew,never);  {tlold - tlnew}
```

```
            if not never then tcc^.f3.l:=tl
            else tcc^.flag:=-1;
            tlnew:=time_subtraction(tlnew,tlold,never);
                                              {tlnew - tlold}
            if not never then tcc:=tcc^.f1.l
            else still:=false;
            end;  {while (tcc <> nil) and still}
         {add what remains}
         if tlnew <> nil then begin
            temp:=new_list(lll,1);
            temp^.f1.l:=tc;
            temp^.f2.l:=nil;
            temp^.f3.l:=tlnew;
            tc:=temp;
            end;
         {edit the list}
{QQB
WRITELN('WHILE_GEN_TIMED: tl before editing');
{QQE}
         while tc^.flag = -1 do tc:=tc^.f1.l;
         tcc:=tc;
         while tcc <> nil do begin
            tccc:=tcc;
            while (tccc^.f1.l <> nil) do begin
               if tccc^.f1.l^.flag = -1 then
                  tccc^.f1.l:=tccc^.f1.l^.f1.l
               else if tccc^.f1.l^.flag = tcc^.flag then begin
                  tcc^.f3.l:=time_union(tcc^.f3.l,tccc^.f1.l^.f3.l);
                  tccc^.f1.l:=tccc^.f1.l^.f1.l;
                  end
               else tccc:=tccc^.f1.l;
               end;
            tcc:=tcc^.f1.l;
            end;  {tcc <> nil}
{QQB
WRITELN('WHILE_GEN_TIMED: end');
```

```
WRITELN('tc:');
LIST_LIST(tc);
{QQE}
         end;  {while_gen_timed}

   function last_case_timed{(tc:list_pointer;qq:quant_type;qn,nn:integer;
      entry:n_tuple; last:boolean; var geno:list_pointer):page_pointer};
      var tcc,tltrue,tlfalse,genn:list_pointer;
          never_true,never_false:boolean;
      begin
         never_true:=true;
         never_false:=true;
         tcc:=tc;
         while tc <> nil do begin
            case qq of
               at_least_n : if tc^.flag >= qn then begin
                               if never_true then begin
                                  tltrue:=tc^.f3.1;
                                  never_true:=false;
                                  end
                               else tltrue:=time_union(tltrue,tc^.f3.1);
                               end
                            else if never_false then begin
                               tlfalse:=tc^.f3.1;
                               never_false:=false;
                               end
                            else tlfalse:=time_union(tlfalse,tc^.f3.1);
               exactly_n  : if tc^.flag = qn then begin
                               if never_true then begin
                                  tltrue:=tc^.f3.1;
                                  never_true:=false;
                                  end
                               else tltrue:=time_union(tltrue,tc^.f3.1);
                               end
                            else if never_false then begin
                               tlfalse:=tc^.f3.1;
```

```
                              never_false:=false;
                              end
                           else tlfalse:=time_union(tlfalse,tc^.f3.1);
                  at_most_n  : if tc^.flag <= qn then begin
                              if never_true then begin
                                 tltrue:=tc^.f3.1;
                                 never_true:=false;
                                 end
                              else tltrue:=time_union(tltrue,tc^.f3.1);
                              end
                           else if never_false then begin
                              tlfalse:=tc^.f3.1;
                              never_false:=false;
                              end
                           else tlfalse:=time_union(tlfalse,tc^.f3.1);
                     how_many   :  begin end;
                     end;  {case}
               tc:=tc^.f1.1;
               end; {while loop}
            if qq = how_many then begin
{QQB
WRITELN('qq=how_many');
{QQE}
               tc:=tcc;
               while tc <> nil do begin
                  entry^[nn].r:=tc^.flag;
                  build_sorted_record(geno,0,entry,tc^.f3.1);
                  tc:=tc^.f1.1;
                  end;
               end
            else begin
               if not never_true then begin
                  entry^[nn].b:=[tru];
                  build_sorted_record(geno,0,entry,tltrue);
                  end;
               if not never_false then begin
```

```
      entry^[nn].b:=[];
      build_sorted_record(geno,0,entry,tlfalse);
      end;
   end;
if last then last_case_timed:=finish_new_record(geno)
else last_case_timed:=nil_page_pointer;
end;  {last_case_timed}
```

Appendix D

Case study for one_class_cruncher

```
>class:CC
 The new class CC has been added.

>individual:II,I1,I2,JJ,J1,J2,J3,KK,K1,LL,L1,L2
 The following new individuals have been added:
   II  I1  I2  JJ  J1  J2  J3  KK  K1  LL  L1  L2

>relation:AA
 The new individual relation AA has been added.
>II, JJ, KK and LL are CCs.
 II  JJ  KK  LL  have been added to the class CCs.
>AA of II is I1 starting October 1, 1988 and ending October 1, 1990.
 I1 was added as AA of II.
>AA of II is I2 starting October 1, 1987.
 I2 was added as AA of II.
>AA of JJ is I1 starting October 1, 1986 and ending October 1, 1989.
 I1 was added as AA of JJ.
>AA of JJ is I2 starting October 1, 1985.
 I2 was added as AA of JJ.
>AA of JJ is J3 starting October 1, 1988 and ending October 1, 1990.
 J3 was added as AA of JJ.
>AA of KK is K1 starting October 1, 1989.
 K1 was added as AA of KK.
>AA of LL is I1 starting October 1, 1985 and ending October 1, 1987.
 I1 was added as AA of LL.
>AA of LL is L2 starting October 1, 1986 and ending October 1, 1991.
 L2 was added as AA of LL.
```

```
   >Who is AA of each CC?
CC        AA
LL        L2        starting October 1, 1986 to ending October 1, 1991
          I1        starting October 1, 1985 to ending October 1, 1987
KK        K1        starting October 1, 1989
JJ        J3        starting October 1, 1988 to ending October 1, 1990
          I2        starting October 1, 1985
          I1        starting October 1, 1986 to ending October 1, 1989
II        I2        starting October 1, 1987
          I1        starting October 1, 1988 to ending October 1, 1990

>Who is AA of some CC?

L2        starting October 1, 1986 to ending October 1, 1991
K1        starting October 1, 1989
J3        starting October 1, 1988 to ending October 1, 1990
I2        starting October 1, 1985
I1        starting October 1, 1985 to ending October 1, 1990

>Who is AA of at least 2 CCs?

I2        starting October 1, 1987
I1        starting October 1, 1986 to ending October 1, 1987
          starting October 1, 1988 to ending October 1, 1989

>Who is AA of how many CCs?
There are 19 lines in your answer.
>Which lines do you want?
```

```
>all

AA        # of CCs
L2        1         starting October 1, 1986 to ending October 1, 1991
L2        0         ending October 1, 1986
                    starting October 1, 1991
K1        1         starting October 1, 1989
K1        0         ending October 1, 1989
J3        1         starting October 1, 1988 to ending October 1, 1990
J3        0         ending October 1, 1988
                    starting October 1, 1990
I2        2         starting October 1, 1987
I2        1         starting October 1, 1985 to ending October 1, 1987
I2        0         ending October 1, 1985
I1        1         starting October 1, 1985 to ending October 1, 1986
                    starting October 1, 1987 to ending October 1, 1988
                    starting October 1, 1989 to ending October 1, 1990
I1        2         starting October 1, 1986 to ending October 1, 1987
                    starting October 1, 1988 to ending October 1, 1989
I1        0         ending October 1, 1985
                    starting October 1, 1990
```

Case study for member_cruncher unlabeled_class

```
>class:CC
 The new class CC has been added.
>individual:KK,JJ,II
 The following new individuals have been added:
   KK  JJ  II
>KK is a CC starting October 1, 1988 and ending October 1, 1990.
 KK has been added to the class CC.
>JJ is a CC starting October 1, 1987.
 JJ has been added to the class CC.
>II is a CC starting October 1, 1986 and ending October 1, 1989.
 II has been added to the class CC.
>What CC are there?

II        starting 12:00 AM, October 1, 1986 to ending 12:00 AM, October 1, 1989
JJ        starting 12:00 AM, October 1, 1987
KK        starting 12:00 AM, October 1, 1988 to ending 12:00 AM, October 1, 1990

>There are at least 2 CC?

yes       starting 12:00 AM, October 1, 1987 to ending 12:00 AM, October 1, 1990
no        ending 12:00 AM, October 1, 1987
          starting 12:00 AM, October 1, 1990

>There are at most 2 CC?

yes       ending 12:00 AM, October 1, 1988
          starting 12:00 AM, October 1, 1989
no        starting 12:00 AM, October 1, 1988 to ending 12:00 AM, October 1, 1989

>There are how many CCs?

3         starting 12:00 AM, October 1, 1988 to ending 12:00 AM, October 1, 1989
2         starting 12:00 AM, October 1, 1987 to ending 12:00 AM, October 1, 1988
          starting 12:00 AM, October 1, 1989 to ending 12:00 AM, October 1, 1990
1         starting 12:00 AM, October 1, 1986 to ending 12:00 AM, October 1, 1987
```

```
          starting 12:00 AM, October 1, 1990
0         ending 12:00 AM, October 1, 1986
```

Case study for member_cruncher labeled_class

```
>class:CC
 The new class CC has been added.
>individual:II,I1,I2,JJ,J1,J2,J3,KK,K1,LL,L1,L2
 The following new individuals have been added:
   II  I1  I2  JJ  J1  J2  J3  KK  K1  LL  L1  L2
>relation:AA
 The new individual relation AA has been added.
>II, JJ, KK and LL are CC.
 II  JJ  KK  LL  have been added to the class CC.
>AA of II is I1 starting October 1, 1988 and ending October 1, 1990.
 I1 was added as AA of II.
>AA of II is I2 starting October 1, 1987.
 I2 was added as AA of II.
>AA of JJ is J1 starting October 1, 1986 and ending October 1, 1989.
 J1 was added as AA of JJ.
>AA of JJ is J2 starting October 1, 1985.
 J2 was added as AA of JJ.
>AA of JJ is J3 starting October 1, 1988 and ending October 1, 1990.
 J3 was added as AA of JJ.
>AA of KK is K1 starting October 1, 1989.
 K1 was added as AA of KK.
>AA of LL is L1 starting October 1, 1985 and ending October 1, 1987.
 L1 was added as AA of LL.
>AA of LL is L2 starting October 1, 1986 and ending October 1, 1991.
 L2 was added as AA of LL.
>There are what AA of each CC?
CC        AA
LL        L2        starting October 1, 1986 to ending October 1, 1991
          L1        starting October 1, 1985 to ending October 1, 1987
KK        K1        starting October 1, 1989
JJ        J3        starting October 1, 1988 to ending October 1, 1990
```

```
          J2        starting October 1, 1985
          J1        starting October 1, 1986 to ending October 1, 1989
II        I2        starting October 1, 1987
          I1        starting October 1, 1988 to ending October 1, 1990

>There are at least 2 AA of each CC?

LL        yes       starting October 1, 1986 to ending October 1, 1987
          no        ending October 1, 1986
                    starting October 1, 1987
JJ        yes       starting October 1, 1986 to ending October 1, 1990
          no        ending October 1, 1986
                    starting October 1, 1990
II        yes       starting October 1, 1988 to ending October 1, 1990
          no        ending October 1, 1988
                    starting October 1, 1990

>There are how many AAs of each CC?
 There are 18 lines in your answer.
>Which lines do you want?
>all

LL        1         starting October 1, 1985 to ending October 1, 1986
                    starting October 1, 1987 to ending October 1, 1991
LL        2         starting October 1, 1986 to ending October 1, 1987
LL        0         ending October 1, 1985
                    starting October 1, 1991
KK        1         starting October 1, 1989
KK        0         ending October 1, 1989
JJ        3         starting October 1, 1988 to ending October 1, 1989
JJ        2         starting October 1, 1986 to ending October 1, 1988
                    starting October 1, 1989 to ending October 1, 1990
JJ        1         starting October 1, 1985 to ending October 1, 1986
                    starting October 1, 1990
JJ        0         ending October 1, 1985
II        2         starting October 1, 1988 to ending October 1, 1990
II        1         starting October 1, 1987 to ending October 1, 1988
                    starting October 1, 1990
```

II	0	ending October 1, 1987

Case study for eval_np_np

```
>class:CC,DD
 The following new classes have been added:
   CC  DD
>individual:II,JJ,KK,LL
 The following new individuals have been added:
   II  JJ  KK  LL
>II is a CC starting October 1, 1988 and ending October 1, 1990.
 II has been added to the class CC.
>II is a DD starting October 1, 1987.
 II has been added to the class DD.
>JJ is a CC starting October 1, 1986 and ending October 1, 1989.
 JJ has been added to the class CC.
>JJ is a DD starting October 1, 1988.
 JJ has been added to the class DD.
>KK is a CC starting October 1, 1985 and ending October 1, 1989.
 KK has been added to the class CC.
>LL is a DD starting October 1, 1987 and ending October 1, 1988.
 LL has been added to the class DD.
>What DD is a CC?

JJ        starting October 1, 1988 to ending October 1, 1989
II        starting October 1, 1988 to ending October 1, 1990

>What DD is not a CC?

LL        starting October 1, 1987 to ending October 1, 1988
JJ        starting October 1, 1989
II        starting October 1, 1987 to ending October 1, 1988
          starting October 1, 1990

>how many DDs are CCs?

1         starting October 1, 1989 to ending October 1, 1990
2         starting October 1, 1988 to ending October 1, 1989
```

```
0          ending October 1, 1988
           starting October 1, 1990

>DDs are CCs?

yes        starting October 1, 1988 to ending October 1, 1990
no         starting October 1, 1987 to ending October 1, 1988
           starting October 1, 1990
```

Case study: eval_nu_nu, boolean_cruncher and onee_class_cruncher

```
>class:employee
 The new class employee has been added.
>individual:John,Mary,Bob,Betty
 The following new individuals have been added:
   John  Mary  Bob  Betty
>number attribute:salary
 Please enter the two adjectives associated
 with salary (similar to "large" and "small" as
 associated with "size"); the maximizing adjective
 (e.g. "large") before the minimizing
 adjective (e.g. "small"):high,low
 The adjectives high and low have been
 added associated with salary.
 The new number attribute salary has been added.
>John,Mary,Bob and Betty are employees.
 John  Mary  Bob  Betty
   have been added to the class employees.
>John's salary is 2000 starting October 1, 1988 and ending October 1, 1990.
 2000 was added as salary of John.
>John's salary is 3000 starting October 1, 1990.
 2000 has been replaced by 3000 as salary of John.
>Mary's salary is 2100 starting October 1, 1986 to May 1, 1988.
 2100 was added as salary of Mary's.
>Mary's salary is 3500 starting May 1, 1988.
 2100 has been replaced by 3500 as salary of Mary's.
>Bob's salary is 2800 starting January 1, 1986.
 2800 was added as salary of Bob.
>Betty's salary is 3000 starting May 1, 1986.
 3000 was added as salary of Betty.
>Is John's salary greater than 2500?

yes       starting October 1, 1990
no        starting October 1, 1988 to ending October 1, 1990

>salary of employees is greater than 2500?
yes       starting January 1, 1986
```

```
no          starting October 1, 1986 to ending May 1, 1988
            starting October 1, 1988 to ending October 1, 1990

>salary of what employee is greater than 2500?
Betty       starting May 1, 1986
Bob         starting January 1, 1986
Mary        starting May 1, 1988
John        starting October 1, 1990

>salary of how many employees is greater than 2500?
4           starting October 1, 1990
3           starting May 1, 1988 to ending October 1, 1990
1           starting January 1 to ending May 1, 1986
2           starting May 1, 1986 to ending May 1, 1988
0           ending January 1, 1986

>salary of at least 2 employees is greater than 2500?
yes         starting May 1, 1986
no          ending May 1, 1986

>2500 is less than the salary of employees?
yes         starting January 1, 1986
no          starting October 1, 1986 to ending May 1, 1988
            starting October 1, 1988 to ending October 1, 1990

>2500 is less than the salary of what employee?
Betty       starting May 1, 1986
Bob         starting January 1, 1986
Mary        starting May 1, 1988
John        starting October 1, 1990
```

```
   >2500 is less than the salary of how many employees?
4          starting October 1, 1990
3          starting May 1, 1988 to ending October 1, 1990
1          starting January 1 to ending May 1, 1986
2          starting May 1, 1986 to ending May 1, 1988
0          ending January 1, 1986

>2500 is less than the salary of at least 2 employees?
yes        starting May 1, 1986
no         ending May 1, 1986
```