1 Zeitspannen reloaded 3
1.1 Lösungsidee 3
1.1.1 TimeSpanToSeconds / SecondsToTimeSpan 3
1.1.2 Zeitdifferenz 4
1.2 Source 4
1.3 Tests 15
1.3.1 TimeSpanToSeconds / SecondsToTimeSpan 15
1.3.1.1 Aktion nicht definiert 15
1.3.1.2 Grenzüberschreitungen (Neue Funktionen der Fehlerbehandlung) 15
1.3.1.3 Gültige Werte 16
1.3.2 TimeDifference 16
1.3.2.1 Grenzbereichsüberschreitung 16
1.3.2.2 Gültige Werte 17
2 Summarizer reloaded 18
2.1 Lösungsidee 18
2.2 Sonderfälle 18
2.2.1 Array beinhaltet nur 0 18
2.2.2 Array beinhaltet keine 0 19
2.2.3 Array beinhaltet am Ende 0 19
2.2.4 Array beinhaltet in der Mitte 0 19
2.2.5 Verschiedene Indizes der Arrays 19
2.3 Source 19
2.4 Tests 22
2.4.1 Sonderfälle 22
2.4.2 Normal mit Zufallswerten 23
2.4.3 Nicht initialisiert 23
3 Römische Zahlen 24
3.1 Lösungsidee 24
3.1.1 IntToRoman (ohne Subtraktionsregel) 24

3.1.2 IntToRoman (mit Subtraktionsregel) 25
3.1.3 RomanToInt 25
3.2 Source 26
3.3 Tests.....37
3.3.1 IntToRoman (ohne Subtraktionsregel) 37
3.3.2 IntToRoman (mit Subtraktionsregel) 38
3.3.3 RomanToInt 38
4 Diskussion 41

1 Zeitspannen reloaded

Folgend ist die Dokumentation der Aufgabe 1 der Übung 3 angeführt, wo die bereits implementierten Funktionen und Prozeduren der Übung2 (Zeitkonvertierung) wiederverwendet und aufgabengemäß angepasst werden sollen.

1.1 Lösungsidee

Folgend ist die Lösungsidee für die Aufgabe 1 angeführt, welche eine Neuimplementierung der Aufgabe Zeitkonvertierung der Übung 2 mit Hilfe eigens eingeführter Datentypen verlangt und die Berechnung der Zeitdifferenz zweier Sekundenwerte.
Dieses Mal wird auf eine doppelte Dokumentation (Dokument und Source) verzichtet, daher bitte unbedingt die Dokumentation im Source, bezüglich Änderungen und neuer Funktionen und Prozeduren, beachten.

1.1.1 TimeSpanToSeconds / SecondsToTimeSpan

Für diese Aufgabenstellung sollen die Algorithmen der Übung 2 weitestgehend wiederverwendet werden. Es soll darauf geachtet werden, dass die Funktionen und Prozeduren so effizient wie möglich angepasst werden. Diese Änderungen werden im Source bei der Schnittstellendokumentation dokumentiert, daher sei hier auf den Source verwiesen.
Es gab bereits eine Fehlerbehandlung, die in dieser Implementierung verfeinert und erweitert werden soll. Die möglichen Fehler sollen über eine Enumeration spezifiziert werden. Insbesondere soll es diesmal möglich sein alle aufgetretenen Fehler zu behandeln und dem Benutzer über die Konsole anzuzeigen.
Da hier ein Datentyp namens TimeSpan eingeführt werden soll, der das Datenobjekt für die Zeitwerte darstellt, soll dieser Datentyp auch die Information der Aktion, die auf die Instanz angewendet werden kann, zur Verfügung stellen. Als Beispiel wenn eine Zeitdarstellung hinterlegt ist, kann auch nur eine Konvertierung dieser Zeitdarstellung zu einer Sekundendarstellung einen sinnvollen Wert ergeben. Daher soll hier auch ein Datentyp eingeführt werden der spezifiziert welche Art der Aktion auf die Instanz von TimeSpan ausgeführt soll. Des Weiteren soll versucht werden alle oft verwendeten Definitionen über Typen zu spezifizieren, um die Definitionen zu kapseln, sodass sie an einer Stelle geändert werden können, ohne dass der gesamte Code durchsucht werden muss. Bei diesen Änderungen und Definitionen sei auf die Dokumentation im Source verwiesen.

1.1.2 Zeitdifferenz

Es soll die Zeitdifferenz in Sekunden zweier Zeitwerte errechnet werden. Diese beiden Zeitwerte sollen über zwei Instanzen von TimeSpan zur Verfügung gestellt werden. Das Ergebnis inklusive der verwendeten Zeitwerte für die Berechnung sowie etwaige aufgetretene Fehler sollen über eine drei TimeSpan Instanzen nach außen zurückgegeben werden.

1. Before TimeSpan result
2. After TimeSpan result
3. Calculation result

Die existierende Funktion, die für die Fehlerbehandlung der vorherig beschriebenen Aufgabe (a) verwendet wird, soll so erweitert werden, damit sie auch Fehler behandeln kann, welche bei der Berechnung der Differenz auftreten können. Es soll darauf geachtet werden, dass die beiden Zeitwerte nicht einander überschreiten. Dies bedeutet das die TimeSpan Instanz, die den älteren Zeitwert darstellt, nicht die TimeSpan Instanz, die den jüngeren Zeitwert darstellt, überschreiten darf. Ein solcher Fall ist als Fehler zu interpretieren und soll dementsprechend behandelt werden.

1.2 Source

Folgend ist der Source des Programms SummarizerReloaded angeführt. Hierbei sind alle vorherig dokumentierten Teilaufgaben inkludiert.

```
{
  This program is the refactored program of TimeConverter.
  It uses now custom spezified Types and not plain attributes anymore.
  For the made changes please see interface documentation.
}
PROGRAM TimeConverter;

  { ################### Type definitions ################### }
  TYPE

    {
      Enumeration which specifies the errors which can occur during an action invocation.
      These enumerations are used to identify the occurred errors during the error handling.
      The TimeSpan instance will hold any error which has occurred during an action
invocation.
    }
    ErrorCodes = (NO_ERROR, INVALID_SECONDS, INVALID_MINUTES, INVALID_HOURS, SECOND_OVERFLOW);

    {
      This enumeration specifies the actions which can be performed on the TimeSpan instance.
      These enumerations are set on the TimeSpan instance to indicate which action shall be
performed
      on this instance, to get a proper result.
    }
    ActionType = (NO_CONVERSION, TIME_TO_SECOND, SECOND_TO_TIME, CALC_DIFFERENCE);
```

```
    {
      Enumeration which specifies the possible error types and is used to act as an array
index.
      These enumerations are used to identify the type of error.
      This means it indicates on which value the error has occurred.
    }
    ErrorType = (HOUR, MINUTE, SECOND, OVERFLOW);
    {
      Enumeration which specifies the types of the time constants and is used to act as an
array index.
      These enumerations specify the time attributes which do have constants settings.
    }
    TimeConstantType = (HOURS_PER_DAY, MINUTES_PER_HOUR, SECONDS_PER_MINUTE);

    {
      Enumeration which specifies the types of the seconds hold by the TimeSpan instance.
      It acts as an array index, and is used to identify the before and after seconds during
the error handling
      when calculate time difference action is performed.
    }
    SecondType = (BEFORE_TIME, AFTER_TIME);

    {
      The array type which specifies an array with an index of the type 'TimeConstantType'
      Holds every time constants used for the conversion actions.
    }
    TimeConstantArray = ARRAY [TimeConstantType] OF INTEGER;

    {
      The array type which specifies an array with an index of the type 'SecondType'.
      Holds the before and after seconds used for the error handling during calculate time
difference action invocation.
    }
    SecondArray = ARRAY [SecondType] OF LONGINT;

    {
      The array type which specifies an array with an index of the type 'ErrorType'
      Holds the occurred errors which can occur during an action invocation.
    }
    ErrorArray = ARRAY [ErrorType] OF ErrorCodes;

    {
      Record which holds the information of the time representations.
      Acts as the data object used for all of the specified and implemented actions.
    }
    TimeSpan = RECORD
        hour: INTEGER;
        minute: INTEGER;
        second: INTEGER;
        timeInSeconds: LONGINT;
        secondsArray: SecondArray;
```

```
        actionType: ActionType;
        errorArray: ErrorArray;
    END;


  { ################### Constant definitions ################### }
  {
    Constants which define the time constants.
    The values are accessible via the 'TimeConstantType' enum values acting as the index of
this array type.
    These constants are used for the time conversion actions.
    ATTENTION: The order of the values must fit the 'TimeConstantType' enumeration
specification.
  }
  CONST
    timeConstant: TimeConstantArray = (24, 60,60);


  { ################### Helper Functions and Procedures ################### }
  {
    Converts a integer to a string representation.

    @param value:
              the integer to be converted to a string
    @return
        the string representation of the given integer value
  }
  FUNCTION ToString(value: LONGINT): STRING;
  VAR
    s: STRING;
  BEGIN

    Str(value, s);
    ToString := s

  END;

  {
     This function is used to initiate the TimeSpan instance with the given values for the
defined attributes.
     The 'errorArray' will be set to 'NO_ERROR' by default.

     @param
        h: the hour to be set on the instance
     @param
        m: the minute to be set on the instance
     @param
        s: the second to be set on the instance
     @param
        timeInSeconds: the second representation to be set on the instance
     @param
```

```
        actionType: the action type to be set on the instance
     @return
         the instantiated TimeSpan instance
   }
   FUNCTION InitTimeSpan(h, m, s: INTEGER; timeInSeconds: LONGINT; actionType: ActionType):
TimeSpan;
   VAR
      i: ErrorType;
      j: SecondType;
      span: TimeSpan;
   BEGIN

      span.hour := h;
      span.minute := m;
      span.second := s;
      span.timeInSeconds := timeInSeconds;
      span.actionType := actionType;

      FOR i := Low(span.errorArray) TO High(span.errorArray) DO
        span.errorArray[i] := NO_ERROR;

      FOR j := Low(span.secondsArray) TO High(span.secondsArray) DO
        span.secondsArray[j] := 0;

      InitTimeSpan := span;

   END;

 {
    Function which validates the range of the given 'value'.

    @param
        value: the value which range shall be validated
    @param
        bottom: the bottom border of the given 'value'
    @param
        top: the top border of the given 'value'
    @return
        true if the value is within the defined borders, false otherwise or if bottom > top
  }
  FUNCTION ValidateRange(value, bottom, top: INTEGER): BOOLEAN;
  BEGIN

      ValidateRange := false;

      IF (bottom <= top) AND (value >= bottom) AND (value <= top) THEN
        ValidateRange := true

  END;

  {
```

```
    Function which detects an occurred error on the TimeSpan instance.
    An error is detected when at least one array element is set with an
    error not equal to 'NO_ERROR'.

    @param
        span: the TimeSpan instance to detect set errors on.
    @return
        true if an error has been detected, false otherwise
  }
  FUNCTION HasErrors(span: TimeSpan): BOOLEAN;
  VAR
    i: ErrorType;
    code: ErrorCodes;
    error: BOOLEAN;
  BEGIN

    error := false;
    i := Low(span.errorArray);
    code := span.errorArray[i];
    IF code <> NO_ERROR THEN
        error := true;

    WHILE (NOT error) AND (i < High(span.errorArray)) DO BEGIN
      Inc(i);
      code := span.errorArray[i];

      IF code <> NO_ERROR THEN
        error := true;

    END;

    HasErrors := error;

  END;

  {
    Function to handle any error occurred on an action invocation and set on the given
TimeSpan instance.

    @param
        span: the TimeSpan instance on which an action was performed and which could have any
errors set.
    @return
        the proper error message depending if errors were set on the TimeSpan instance.
    @changes
        This function now handles all of the occurred errors and displays them to the console.
        Formerly only the first detected error was handled.
  }
  FUNCTION HandleErrors(span: TimeSpan): STRING;
  VAR
    errors: STRING;
```

```
  BEGIN

    errors := '';
    HandleErrors := 'No error detected';

    IF HasErrors(span) THEN BEGIN
      {
        Get errors for 'TIME_TO_SECOND' conversion type
      }
      IF ((span.actionType = CALC_DIFFERENCE) OR (span.actionType = TIME_TO_SECOND)) AND
(span.errorArray[OVERFLOW] <> SECOND_OVERFLOW) THEN BEGIN
        IF span.errorArray[HOUR] = INVALID_HOURS THEN
          errors := errors + '[' + ToString(span.hour) + ']:'
        ELSE
          errors := errors + ToString(span.hour) + ':';
        IF span.errorArray[MINUTE] = INVALID_MINUTES THEN
          errors := errors + '[' + ToString(span.minute) + ']:'
        ELSE
          errors := errors + ToString(span.minute) + ':';
        IF span.errorArray[SECOND] = INVALID_SECONDS THEN
          errors := errors + '['+ ToString(span.second) + ']'
        ELSE
          errors := errors + ToString(span.second) + '';
      END;
      {
        Get error for 'SECOND_TO_TIME'
      }
      IF (span.actionType = SECOND_TO_TIME) AND (span.errorArray[SECOND] = INVALID_SECONDS)
THEN
        errors := errors + '[' + ToString(span.timeInSeconds) + ']';
      {
        Get the error for 'CALC_DIFFERENCE'
      }
      IF (span.actionType = CALC_DIFFERENCE) AND (span.errorArray[OVERFLOW] = SECOND_OVERFLOW)
THEN
        errors := errors + 'Overflow. before: ' + ToString(span.secondsArray[BEFORE_TIME]) + '
| after: ' + ToString(span.secondsArray[AFTER_TIME]);

      HandleErrors := 'One or more errors have been detected. ' + errors;

    END;
  END;

  { ################### Action Functions and Procedures ################### }
  {
    Function which converts the 'timeInSeocnds' set on the given TimeSpan instance to hour,
minute and second values, which will be set on the returned TimeSpan instance.

    @param
        span: the TimeSpan instance which holds the to convert seconds
    @return
```

```
        the TimeSpan instance containing the calculated time representation of the set
'timeInSeconds' value
    @changes
        Formerly this function was a procedure and this procedure had formal parameters for
the timeInSeoonds and each calculated value to be returned.
        Now all values are provided by the given TimeSpan instance and the result will be set
on this instance as well.
        This is the reason why this is now an function and not a procedure anymore.
  }
  FUNCTION SecondsToTimeSpan(span: TimeSpan): TimeSpan;
  BEGIN

    IF span.timeInSeconds > 0 THEN BEGIN
      span.hour := span.timeInSeconds DIV (timeConstant[SECONDS_PER_MINUTE] *
timeConstant[MINUTES_PER_HOUR]);
      span.minute := (span.timeInSeconds - (span.hour * (timeConstant[SECONDS_PER_MINUTE] *
timeConstant[MINUTES_PER_HOUR]))) DIV timeConstant[MINUTES_PER_HOUR];
      span.second := (span.timeInSeconds - (span.hour * (timeConstant[SECONDS_PER_MINUTE] *
timeConstant[MINUTES_PER_HOUR])) - (span.minute * timeConstant[MINUTES_PER_HOUR]));
    END
    ELSE
      span.errorArray[SECOND] := INVALID_SECONDS;

    SecondsToTimeSpan := span;

  END;

  {
    Function which calculates the second representation of the given time 'hh:mm:ss' set on
the TimeSpan instance.

    @param
        span: the TimeSpan instance which holds the time representation.
    @return
        the TimeSpan instance with the set calculated 'timeInSeconds' value or any occurred
errors set on 'errorArray'
    @see ValidateRange(value, range: INTEGER): BOOLEAN;
    @see CONST at program head
        timeConstant Array with all used time constants
    @changes
        Formerly this function had formal parameters for all time values and returned the
calculated seconds.
        Also this function was only able to detect one error, because only one error was
handleable.
        The error was saved on the returned second value with a value less then zero.
        No all needed time values are hold by the given TimeSpan instance and also the result
or all occurred errors
        are set on the returned TimeSpan instance.
  }
  FUNCTION TimeSpanToSeconds(span: TimeSpan): TimeSpan;
  BEGIN
```

```
    IF span.hour < 0 THEN
      span.errorArray[HOUR] := INVALID_HOURS;
    IF NOT ValidateRange(span.minute, 0, (timeConstant[MINUTES_PER_HOUR] - 1)) THEN
      span.errorArray[MINUTE] := INVALID_MINUTES;
    IF NOT ValidateRange(span.second, 0, (timeConstant[SECONDS_PER_MINUTE] - 1)) THEN
      span.errorArray[SECOND] := INVALID_SECONDS;

    IF NOT HasErrors(span) THEN
      span.timeInSeconds := (span.hour * timeConstant[MINUTES_PER_HOUR] *
timeConstant[SECONDS_PER_MINUTE]) + (span.minute * timeConstant[SECONDS_PER_MINUTE]) +
span.second;

    TimeSpanToSeconds := span;

  END;

  {
    Procedure which calculates the time difference in seconds between the two given TimeSpan
    instances.

    @param
        before: the TimeSpan instance representing the lower time value
    @param
        after: the TimeSpan instance representing the higher time value
    @return
        calcResult: the TimeSpan instance which holds the calculated time difference, or the
set error
                    which would occur if 'before TimeSpan ' has a second value which is after
the 'after TimeSpan' second value.
        beforeResult: the before TimeSpan instance which holds the result or any occurres
errors during seocnd conversion
        beforeResult: the after TimeSpan instance which holds the result or any occurres
errors during seocnd conversion
  }
  PROCEDURE TimeDifference(before, after:TimeSpan; VAR beforeResult, afterResult,
calcResult:TimeSpan);
  BEGIN
    calcResult := InitTimeSpan(0,0,0,0, CALC_DIFFERENCE);
    beforeResult := TimeSpanToSeconds(before);
    afterResult := TimeSpanToSeconds(after);
    calcResult.secondsArray[BEFORE_TIME] := beforeResult.timeInSeconds;
    calcResult.secondsArray[AFTER_TIME] := afterResult.timeInSeconds;

    IF (NOT HasErrors(beforeResult)) AND (NOT HasErrors(afterResult)) THEN BEGIN
      IF beforeResult.timeInSeconds <= afterResult.timeInSeconds THEN
        calcResult.timeInSeconds := afterResult.timeInSeconds - beforeResult.timeInSeconds
      ELSE
        calcResult.errorArray[OVERFLOW] := SECOND_OVERFLOW;
    END;
  END;
```

```
  { ################### Test Procedures ################### }
  {
    Procedure to test the Function TimeSpanToSeconds.
    Prints the result or the error to the console.

    @param
        span: the TimeSpan instance which holds the to convert time representation
    @changes
        This procedure now works with the TimeSpan instance and not with the formerly given
time values.
  }
  PROCEDURE TestTimeSpanToSeconds(span: TimeSpan);
  BEGIN

    span := TimeSpanToSeconds(span);
    WriteLn('Time: ', span.hour, ':', span.minute, ':', span.second, ' / ');

    IF HasErrors(span) THEN
      WriteLn(HandleErrors(span))
    ELSE
      WriteLn('Seconds: ', span.timeInSeconds);

  END;

  {
    Tests the procedure SecondsToTimeSpan.
    Prints the result or the error to the console.
    @param sec:
             the seconds to convert
    @changes
        This procedure now works with the given TimeSpan instance and not with the formerly
given second value
  }
  PROCEDURE TestSecondsToTimeSpan(span: TimeSpan);
  BEGIN

    WriteLn('Seconds: ', span.timeInSeconds);
    span := SecondsToTimeSpan(span);

    IF HasErrors(span) THEN
      WriteLn(HandleErrors(span))
    ELSE
      WriteLn('Time: ', span.hour, ':', span.minute, ':', span.second, ' / ');

  END;

  {
    Tests the function SecondsToTimeSpan.
    Prints the result or the error to the console.
    @param
```

```
        before: the TimeSpan instance containing the lower second value.
    @param
        after: the TimeSpan instance containing the higher second value.
  }
  PROCEDURE TestTimeDifference(before, after: TimeSpan);
  VAR
    result, beforeResult, afterResult: TimeSpan;
  BEGIN

    TimeDifference(before, after, beforeResult, afterResult, result);

    WriteLn('Before       : ', beforeResult.hour, ':', beforeResult.minute, ':',
beforeResult.second);
    WriteLn('Before error : ', HandleErrors(beforeResult));
    WriteLn('After        : ', afterResult.hour, ':', afterResult.minute, ':',
afterResult.second);
    WriteLn('After error  : ', HandleErrors(afterResult));
    WriteLn('Result: error: ', HandleErrors(result));
    WriteLn('Difference   : ', result.timeInSeconds);

  END;

  {
    Procedure which tests the actions which can be performed on the given TimeSpan instances
depending on the set 'actionType'.
    If an unexpected action type is detected a error message will e printed to the console,
    otherwise the proper function/procedure is invoked.

    @param
        span1: the first TimeSpan instance to test action invocation on.
    @param
        span2: the second TimeSpan instance to test action invocation on. Only used when
actionType = 'CALC_DIFFERENCE'
  }
  PROCEDURE Test(span1, span2: TimeSpan);
  BEGIN

    IF span1.actionType = TIME_TO_SECOND THEN
      TestTimeSpanToSeconds(span1)
    ELSE IF span1.actionType = SECOND_TO_TIME THEN
      TestSecondsToTimeSpan(span1)
    ELSE IF (span1.actionType = CALC_DIFFERENCE) AND (span2.actionType = CALC_DIFFERENCE) THEN
      TestTimeDifference(span1, span2)
    ELSE
      WriteLn('No action type defined !!!');

    WriteLn('-----------------------------------------------------------------------');

  END;

  { ################### Main program ################### }
```

```
VAR
  temp: TimeSpan;
BEGIN

  {
    Dummy TimeSpan instance, needed because two TimeSpan instances are defined a formal
parameters on Test procedure.
    SO only one test procedure is needed, and this one can handle an undefined actionType.
  }
  temp := InitTimeSpan(-1,0,0,0,NO_CONVERSION);
  WriteLn('#########################################################################');
  WriteLn('#                  Tests for TimeSpanToSeconds                          #');
  WriteLn('#########################################################################');
  WriteLn();
  Test(InitTimeSpan(-1,0,0,0,NO_CONVERSION), temp);
  Test(InitTimeSpan(-1,-1,0,0,TIME_TO_SECOND), temp);
  Test(InitTimeSpan(-1,-1,-1,0,TIME_TO_SECOND), temp);
  Test(InitTimeSpan(120,60,0,0,TIME_TO_SECOND), temp);
  Test(InitTimeSpan(120,60,60,0,TIME_TO_SECOND), temp);
  Test(InitTimeSpan(0,0,0,-1,TIME_TO_SECOND), temp);
  Test(InitTimeSpan(120,59,59,-1,TIME_TO_SECOND), temp);

  WriteLn();
  WriteLn('#########################################################################');
  WriteLn('#                  Tests for SecondsToTimeSpan                          #');
  WriteLn('#########################################################################');
  WriteLn();
  Test(InitTimeSpan(-1,-1,-1,0,SECOND_TO_TIME), temp);
  Test(InitTimeSpan(-1,-1,-1,-1,SECOND_TO_TIME), temp);
  Test(InitTimeSpan(0,0,0,59,SECOND_TO_TIME), temp);
  Test(InitTimeSpan(0,0,0,60,SECOND_TO_TIME), temp);
  Test(InitTimeSpan(0,0,0,3599,SECOND_TO_TIME), temp);
  Test(InitTimeSpan(0,0,0,3600,SECOND_TO_TIME), temp);

  WriteLn();
  WriteLn('#########################################################################');
  WriteLn('#                  Tests for TimeDifference                             #');
  WriteLn('#########################################################################');
  WriteLn();
  Test(InitTimeSpan(1,60,60,0,CALC_DIFFERENCE), InitTimeSpan(-1,60,59,0,CALC_DIFFERENCE));
  Test(InitTimeSpan(1,1,59,0,CALC_DIFFERENCE), InitTimeSpan(1,1,58,0,CALC_DIFFERENCE));
  Test(InitTimeSpan(1,1,58,0,CALC_DIFFERENCE), InitTimeSpan(1,1,59,0,CALC_DIFFERENCE));
  Test(InitTimeSpan(1,1,59,0,CALC_DIFFERENCE), InitTimeSpan(1,1,59,0,CALC_DIFFERENCE));
  Test(InitTimeSpan(1,1,59,0,CALC_DIFFERENCE), InitTimeSpan(3,2,59,0,CALC_DIFFERENCE));

END.
```

1.3 Tests

Folgend sind die Tests für die Aufgabe 1 der Übung 3 dokumentiert. Insbesondere werden hier die Änderungen an der Fehlerbehandlung hervorgehoben.

1.3.1 TimeSpanToSeconds / SecondsToTimeSpan

Folgend sind die Tests der Zeitkonvertierungen von einer Zeitdarstellung zu einer Sekundendarstellung angeführt.

Testszenarien:

1. Aktion nicht definiert
2. Grenzüberschreitungen (Neue Funktionen der Fehlerbehandlung)
3. Gültige Werte

1.3.1.1 Aktion nicht definiert

Folgend ist der Test angeführt, welcher testet wie sich der Algorithmus verhält wenn keine gültige Aktion definiert ist.

```
No action type defined !!!
```

Sollte keine gültige Aktion in der TimeSpan Instanz definiert worden sein, so wir dies über eine Fehlermeldung angezeigt. In diesem Fall ist das Programm nicht in der Lage eines angemessenen Algorithmus zu aktivieren.

1.3.1.2 Grenzüberschreitungen (Neue Funktionen der Fehlerbehandlung)

Folgend sind die Tests angeführt, welche die Grenzbereich und die Fehlerbehandlung dieser testen. Grenzüberschreitungen können bei den Stunden, Minuten und Sekunden auftreten.

```
Time: -1:-1:0 /
One or more errors have been detected. [-1]:[-1]:0
Time: -1:-1:-1 /
One or more errors have been detected. [-1]:[-1]:[-1]
Time: 120:60:0 /
One or more errors have been detected. 120:[60]:0
Time: 120:60:60 /
One or more errors have been detected. 120:[60]:[60]

Seconds: 0
One or more errors have been detected. [0]
Seconds: -1
One or more errors have been detected. [-1]
```

Sollten ungültige Werte für entweder Stunden, Minuten oder Sekunden angegeben werden, so werden diese über **[WERT]** angezeigt. Im Gegensatz zur früheren Implementierung werden hier alle Fehler und nicht nur die zuerst erkannten Fehler angezeigt.

1.3.1.3 Gültige Werte

Folgend sind die Tests angeführt, die testen ob sich der Algorithmus wie gewünscht verhält.

```
------------------------------------------------------------------------
Time: 0:0:0 /
Seconds: 0
------------------------------------------------------------------------
Time: 120:59:59 /
Seconds: 435599
------------------------------------------------------------------------

------------------------------------------------------------------------
Seconds: 59
Time: 0:0:59 /
------------------------------------------------------------------------
Seconds: 60
Time: 0:1:0 /
------------------------------------------------------------------------
Seconds: 3599
Time: 0:59:59 /
------------------------------------------------------------------------
Seconds: 3600
Time: 1:0:0 /
------------------------------------------------------------------------
```

1.3.2 TimeDifference

Folgend sind Tests der Zeitdifferenzberechnung angeführt.

Testszenarien:

1. Grenzbereichsüberschreitung
2. Gültige Werte

1.3.2.1 Grenzbereichsüberschreitung

Folgend sind die Tests für die Grenzbereichsüberschreitung angeführt.

```
Before       : 1:60:60
Before error : One or more errors have been detected. 1:[60]:[60]
After        : -1:60:59
After error  : One or more errors have been detected. [-1]:[60]:59
Result: error: No error detected
Difference   : 0
------------------------------------------------------------------------
Before       : 1:1:59
Before error : No error detected
After        : 1:1:58
After error  : No error detected
Result: error: One or more errors have been detected. Overflow. before: 3719 ! a
fter: 3718
Difference   : 0
------------------------------------------------------------------------
```

Wenn die Zeitdarstellungen Fehler beinhalten oder sie einander überschreiten, so verursacht dies einen Fehler, welcher behandelt und ausgeben wird.

1.3.2.2 Gültige Werte

Folgend sind die Tests angeführt, die testen ob sich der Algorithmus wie gewünscht verhält.

```
------------------------------------------------------------------------------
Before        : 1:1:58
Before error  : No error detected
After         : 1:1:59
After error   : No error detected
Result: error: No error detected
Difference    : 1
------------------------------------------------------------------------------
Before        : 1:1:59
Before error  : No error detected
After         : 1:1:59
After error   : No error detected
Result: error: No error detected
Difference    : 0
------------------------------------------------------------------------------
Before        : 1:1:59
Before error  : No error detected
After         : 3:2:59
After error   : No error detected
Result: error: No error detected
Difference    : 7260
------------------------------------------------------------------------------
```

Wenn keinen ungültigen Sekundenwerte angegeben werden, so verhält sich dieser Algorithmus wie erwartet.

2 Summarizer reloaded

Folgend ist die Dokumentation der Aufgabe 2 der Übung 3 angeführt. Hierbei soll die bereits implementierte Lösung der Aufgabe 1 so geändert werden, sodass das Aufsummieren von nummerischen Werten über Felder realisiert wird. Hierbei soll die implementierte Funktion als formalen Parameter ein offenes Array definieren, damit diese Funktion mit Arrays unterschiedlicher Größe und verwendeter Indizes arbeiten kann.

2.1 Lösungsidee

Folgend ist die Lösungsidee der Aufgabe 2 der Übung 3 beschrieben.
Für die beiden gebildeten Summen soll ein Datentyp definiert werden, der diese hält und welcher von der Funktion zurückgegeben wird.
Die bestehende Prozedur der Übung 1 soll so geändert werden, das diese ein offenes Array als formalen Parameter definiert und eine Instanz des Datentyp zurückgibt, welcher die gebildeten Summen beinhaltet.
Da jetzt die Werte nicht mehr eingelesen werden sondern über ein Array zu beziehen sind, soll die Funktion so implementiert werden, sodass folgende besonderen Zustände korrekt behandelt werden:

1. Array beinhaltet nur 0
2. Array beinhaltet keine 0
3. Array beinhaltet am Ende 0
4. Array beinhaltet in der Mitte 0
5. Verschiedene Indizes der Arrays

Außerhalb der Schleife müssen alle Variablen initialisiert werden. Insbesondere muss hierbei bereits der erste Wert des Arrays ermittelt und gesetzt werden, da dieser entscheidet, in welche Summe die darauffolgenden Werte aufgenommen werden. Da der konkrete Typ des Index nicht bekannt ist, wird der DatenTyp LONGINT als Indextyp verwendet. Ein Indexoverflow stellt solange kein Problem dar solange nicht versucht wird über diesen ungültigen Index ein Element aus den Array zu adressieren. Bei einem konkreten Indextyp wie einer Enumeration würde dies zu einem Programmabruch führen.

2.2 Sonderfälle

Folgend sind die oben angeführten Sonderfälle beschrieben, die durch die Neuimplmenetierung mit offenen Feldern entstanden sind.

2.2.1 Array beinhaltet nur 0

Sollte ein Array nur Nullen beinhalten so wird die Schleife 0 mal durchlaufen. Da der erste Wert des Arrays bereits außerhalb der Schleife ermittelt, der aktuelle Wert mit dem ersten Wert initialisiert wird und auch die Summen korrekt initialisiert werden (countGreater = first, countSmaller = 0), stellt dieser Zustand kein Problem dar.

2.2.2 Array beinhaltet keine 0

In diesem Fall wird die Schleife bis zum letzten Element durchlaufen und beendet hier, da die Schleifenbedingung definiert, das die Schleife nur solange durchlaufen wird solange der aktuelle Wert <> 0 und der aktuelle Index <= als der höchstmögliche Index, des Arrays ist.

2.2.3 Array beinhaltet am Ende 0

Sollte das Array am Ende, also das letzte Element, eine 0 haben, so spielt dies keine Rolle, da dies keine besondere Handhabung erfordert und über die Schleifenbedingung value <> 0 gesteuert wird.

2.2.4 Array beinhaltet in der Mitte 0

Auch dieser Zustand erfordert keine besondere Handhabung, da dies auch über die Schleifenbedingung gesteuert wird. Alle nach der 0 folgenden nummerischen Werte werden ignoriert.

2.2.5 Verschiedene Indizes der Arrays

Dieser Zustand spielt in dieser Implementierung keine Rolle, da als Index ein LONGINT verwendet wird und nicht der konkrete Indextyp. Dies ist möglich da es sich hier um offene Arrays handelt, die keinen konkreten IndexTyp definieren. Sollte der konkrete Indextyp verwendet werden, so würde diese Implementierung nicht funktionieren, da versucht werden würde den Index um eines zu erhöhen, obwohl kein Element des Typs mehr zur Verfügung stehen würde.

2.3 Source

```
{
  This program calculates two sums over an open array where the used index
  does not matter for the ComputeSums function.
}
PROGRAM Summarizer;

  TYPE
    {
      Type whih specifies the index for a array with length of 7;
    }
    IntegerIDx1To7 = 1..7;
    {
      Specifies an array of (long)integer type which uses the index type 'IntegerIDx1To7'
    }
    ArrayWithIntegerIdx1To7 = ARRAY[IntegerIDx1To7] OF LONGINT;
    {
      Enumeration which gets used as index for an array
    }
    EnumIdx = (FIRST, SECOND, THRID, FOURTH, FIFTH);
    {
      Summary record which is used to hold the calculated values.
    }
    Summary = RECORD
      countGreater: LONGINT;
      countSmaller: LONGINT;
    END;

  {
    Function which calculates two sums over all numbers contained in the given 'numbers'
array,
    as long as the current value is not equal to '0'.

    @param
        numbers: the array of (long)integers to build sum
    @return
        the Summary containing the builded summaries
```

```
  }
  FUNCTION ComputeSums(numbers: ARRAY OF LONGINT): Summary;
  VAR
    i, first, value: LONGINT;
    summaryHolder: Summary;
  BEGIN

    {
      Init the variables each time the procedure gets invoked.
      Get first 'value', set 'first' with 'value', increase i and add 'value' to countGreater
sum
    }
    i := Low(numbers);
    summaryHolder.countSmaller := 0;
    summaryHolder.countGreater := 0;
    first := numbers[i];

    {
      Start at second element because first has already been handled.
    }
    WHILE (value <> 0) AND (i <= High(numbers)) DO BEGIN

      value := numbers[i];

      IF value > 0 THEN BEGIN
        IF value >= first THEN
          summaryHolder.countGreater := summaryHolder.countGreater + value
        ELSE
          summaryHolder.countSmaller := summaryHolder.countSmaller + value
      END;

      Inc(i);

    END;

    ComputeSums := summaryHolder;

  END;

  {
    Tests the ComputeSum function, by printing the given values and results to the console
    and if intended generates random values for the given array of numbers.
    If the array is intended to be filled with random numbers then at the middle of the array
    a zero will be added, so that the COmputeSum method will have to stop there.

    @param
        numbers: the array of (long)integers to build sum over it, or generates random values
                 for this array if fillRandom is set to true
    @param
        fillRandom: true if random values shall be generated for the array.
  }
  PROCEDURE Test(numbers: ARRAY OF LONGINT; fillRandom: BOOLEAN);
  VAR
    i, middleBorder: LONGINT;
    summaryHolder: Summary;
    print: BOOLEAN;
  BEGIN

    print := true;
    WriteLn('Given array of numbers: ');
    middleBorder := High(numbers) DIV 2;
    FOR i := Low(numbers) TO High(numbers) DO BEGIN
      IF fillRandom THEN
        IF middleBorder = i THEN
          numbers[i] := 0
        ELSE
          numbers[i] := Random(High(numbers)) + 1;
      IF print THEN BEGIN
        Write(numbers[i]);
        IF i < High(numbers) THEN
          Write(', ');
        IF numbers[i] = 0 THEN BEGIN
          print := false;
          IF i < High(numbers) THEN BEGIN
            WriteLn('');
            WriteLn('## stoped print on 0, but values are following ##');
```

```
          END;
          BREAK;
        END;
      END;
    END;

    summaryHolder := ComputeSums(numbers);
    WriteLn('');
    WriteLn('');
    WriteLn('Sum A: ', summaryHolder.countGreater, ' / Sum B: ', summaryHolder.countSmaller);
    WriteLn('------------------------------------------------------');
    WriteLn('');

  END;

{
  Some arrays with different indexes and values for the tests.
}
CONST
    arrayIntegerIdxSingleZero: ARRAY[1..1] OF LONGINT = (0);
    arrayIntegerIdxNoZero: ArrayWithIntegerIdx1To7 = (1,2,3,4,5,6,7);
    arrayIntegerIdxMidZero: ArrayWithIntegerIdx1To7 = (5,4,6,3,0,4,5);
    arrayIntegerIdxEndZero: ArrayWithIntegerIdx1To7 = (1,2,3,4,5,6,0);

{
  Some array with different indexes which are supposed to be filled before used for the test.
}
VAR
    arrayInteger1To7Idx: ArrayWithIntegerIdx1To7;
    arrayEnumIdx: ARRAY[EnumIdx] OF LONGINT;
    arrayByteIdx: ARRAY[BYTE] OF LONGINT;
    arrayCharIdx: ARRAY[CHAR] OF LONGINT;

BEGIN

  { Fixed numbers }
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: 1..1');
  Test(arrayIntegerIdxSingleZero, false);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: 1..7');
  Test(arrayIntegerIdxNoZero, false);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: 1..7');
  Test(arrayIntegerIdxMidZero, false);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: 1..7');
  Test(arrayIntegerIdxEndZero, false);

  { Random numbers }
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: 1..7 || random: true');
  Test(arrayInteger1To7Idx, true);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: EnumIdx || random: true');
  Test(arrayEnumIdx, true);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: BYTE || random: true');
  Test(arrayByteIdx, true);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: CHAR || random: true');
  Test(arrayCharIdx, true);

  { All zero or undefined values }
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: BYTE || not initialized');
  Test(arrayByteIdx, false);
    WriteLn('------------------------------------------------------');
  WriteLn('Index type: INTEGER || not initialized');
  Test(arrayCharIdx, false);

END.
```

2.4 Tests

Folgend sind die Test angeführt, welche den Algorithmus SummarizerReloaded testen. Es werden verschiedene Indextypen verwendet. Des Weiteren werden die oben angeführten Sonderfälle getestet.

Testszenarien:

1. Sonderfälle (Verschiedene Indizes)
2. Normal mit Zufallswerten (Verschiedene Indizes)
3. Nicht initialisiert (Alle Elemente haben Wert 0)

2.4.1 Sonderfälle

Diese Test testen das Verhalten, wenn Arrays mit vordefinierten Werten und verschiedenen Indizes vom Algorithmus verarbeitet werden.

```
------------------------------------------------------
Index type: 1..1
Given array of numbers:
0

Sum A: 0 / Sum B: 0
------------------------------------------------------

------------------------------------------------------
Index type: 1..7
Given array of numbers:
1, 2, 3, 4, 5, 6, 7

Sum A: 28 / Sum B: 0
------------------------------------------------------

------------------------------------------------------
Index type: 1..7
Given array of numbers:
5, 4, 6, 3, 0,
## stoped print on 0, but values are following ##

Sum A: 11 / Sum B: 7
------------------------------------------------------

------------------------------------------------------
Index type: 1..7
Given array of numbers:
1, 2, 3, 4, 5, 6, 0

Sum A: 21 / Sum B: 0
------------------------------------------------------
```

Der Algorithmus kann hierbei mit allen vorherig beschriebenen Sonderfällen umgehen.

2.4.2 Normal mit Zufallswerten

Da auch andere Indizes getestet werden sollten deren Wertebereich relativ groß ist, wurden auch Test implementiert, die mit Zufallswerten arbeiten. Diese Test testen den Algorithmus mit Arrays, deren Werte mittels Zufallsgenerator generiert wurde.

```
------------------------------------------------------
Index type: 1..7 !! random: true
Given array of numbers:
4, 4, 5, 0,
## stoped print on 0, but values are following ##

Sum A: 13 / Sum B: 0
------------------------------------------------------

------------------------------------------------------
Index type: EnumIdx !! random: true
Given array of numbers:
4, 3, 0,
## stoped print on 0, but values are following ##

Sum A: 4 / Sum B: 3
------------------------------------------------------

------------------------------------------------------
Index type: BYTE !! random: true
Given array of numbers:
219, 139, 217, 109, 160, 165, 99, 112, 76, 228, 15, 246, 70, 98, 122, 202, 208,
135, 123, 145, 101, 237, 214, 19, 87, 23, 166, 6, 94, 213, 245, 199, 36, 222, 22
2, 250, 121, 204, 205, 118, 133, 200, 174, 31, 184, 164, 149, 37, 138, 241, 194,
 134, 28, 106, 121, 68, 48, 198, 188, 117, 56, 145, 35, 5, 83, 158, 39, 157, 57,
 158, 99, 241, 231, 174, 115, 92, 157, 112, 231, 178, 26, 16, 248, 171, 167, 172
, 44, 54, 92, 33, 192, 81, 155, 93, 83, 146, 10, 112, 162, 253, 245, 27, 167, 54
, 162, 42, 254, 167, 149, 65, 106, 119, 122, 63, 159, 41, 87, 29, 173, 168, 81,
36, 199, 51, 243, 95, 169, 0,
## stoped print on 0, but values are following ##

Sum A: 4056 / Sum B: 12573
------------------------------------------------------

------------------------------------------------------
Index type: CHAR !! random: true
Given array of numbers:
210, 4, 25, 159, 214, 172, 25, 248, 249, 224, 120, 130, 250, 15, 155, 116, 189,
6, 10, 113, 73, 250, 31, 92, 76, 123, 31, 176, 82, 225, 106, 235, 17, 56, 177, 1
45, 145, 221, 68, 130, 134, 234, 24, 235, 147, 22, 237, 71, 82, 3, 171, 215, 34,
 166, 183, 215, 74, 68, 47, 102, 150, 141, 6, 43, 212, 95, 2, 38, 173, 146, 69,
180, 188, 74, 246, 111, 64, 193, 147, 102, 151, 229, 146, 163, 57, 228, 243, 174
, 115, 115, 216, 250, 179, 30, 76, 196, 208, 106, 102, 173, 225, 64, 149, 80, 22
5, 247, 177, 151, 185, 169, 128, 136, 244, 59, 165, 101, 109, 158, 155, 122, 5,
120, 77, 183, 169, 74, 74, 0,
## stoped print on 0, but values are following ##

Sum A: 6027 / Sum B: 10818
------------------------------------------------------
```

Bei diesen Tests wurden die Arrays bis zur Hälfte mit Zufallswerten befüllt und die restlichen Elemente blieben nicht initialisiert, wobei die nicht initialisierten Element alle den Wert 0 haben. Es wurde hierbei auf eine Ausgabe dieser Nullen verzichtet.

2.4.3 Nicht initialisiert

Diese Test testen den Algorithmus mit nicht initialisierten Arrays, wobei deren Elemente alle den Wert 0 haben.

```
------------------------------------------------------
Index type: BYTE !! not initialized
Given array of numbers:
0,
## stoped print on 0, but values are following ##

Sum A: 0 / Sum B: 0
------------------------------------------------------

------------------------------------------------------
Index type: INTEGER !! not initialized
Given array of numbers:
0,
## stoped print on 0, but values are following ##

Sum A: 0 / Sum B: 0
------------------------------------------------------
```

Hierbei verhält sich der Algorithmus wie spezifiziert und bricht die Verarbeitung nach dem ersten Element ab, da diese bereits den Wert 0 hat.

3 Römische Zahlen

Folgend ist die Dokumentation der Aufgabe 3 der Übung 3 angeführt, wobei eine Konvertierung von ganzzahligen Zahlenwerten zu einer römischen Darstellung und umgekehrt erfolgen soll. Bei der Konvertierung von ganzzahligen Zahlenwerten, soll es zwei Möglichkeiten der Konvertierung geben.

1. Ohne Beachtung der Subtraktionsregel
2. Unter Beachtung der Subtraktionsregel

3.1 Lösungsidee

Folgend sind die Lösungsideen der Aufgabe 3 der Übung 3 angeführt.
Für alle implementierten Funktionen soll ein Datentyp eingeführt werden, der das Mapping zwischen einer römischen Zahlendarstellung und seiner ganzzahligen Zahlendarstellung hält. Ebenso soll der erlaubte Prefix für die römische Zahlendarstellung bei Verwendung der Subtraktionsregel von diesen Datentyp definiert werden.
Es soll eine Enumeration spezifiziert werden, die alle erlaubten römischen Zahlendarstellungen enthält. Diese Enumeration soll in Weiterer Folge als Index für das Array dienen, in dem als Konstante die Instanzen für den vorherig spezifizierten Datentyp gehalten werden. Es sollen alle Datenobjekte erstellt werden, sodass alle römische Zahlenwerte, deren dazugehörige ganzzahligen Zahlenwerte und der erlaubte Prefix enthalten sind. Diese Konstanten werden über alle implementierten Funktionen verwendet.
Es soll ein Datentyp spezifiziert werden, der für das Halten der zu berechnenden Werte, etwaiger Fehler verwendet und Fehlermeldungen verwendet werden soll.
Sollte in den Funktionen ein Wert größer 5000 erkannt werden soll dies als Fehler behandelt werden, da bei dieser Darstellungsform Werte größer 5000 nicht darstellbar sind.

3.1.1 IntToRoman (ohne Subtraktionsregel)

In dieser Funktion soll ein ganzzahliger Zahlenwert in seine adäquate römische Zahlendarstellung ohne Berücksichtigung der Subtraktionsregel konvertiert werden. Dies soll unter der Verwendung der vorherig beschriebenen Konstanten erfolgen. Dabei wird der Zahlenwert durch den höchstmöglichen römischen Zahlenwert bzw. seinen hinterlegten ganzzahligen Zahlenwert dividiert (Integerdivision). Das Ergebnis gibt die Anzahl der zu verwendenden römischen Zahlen an. Die Anzahl multipliziert mit dem ganzzahligen Zahlenwert der verwendeten römischen Zahl wird vom Gesamtwert abgezogen. Dies soll solange erfolgen bis der Rest 0 ist. Das Ergebnis wird in der Konsole ausgegeben.

3.1.2 IntToRoman (mit Subtraktionsregel)

In dieser Funktion soll ein ganzzahliger Zahlenwert in seine adäquate römische Zahlendarstellung unter Berücksichtigung der Subtraktionsregel konvertiert werden.

Wird folgender Ansatz angewendet:

Beispiel: Wert = 999;

1. Ermittle die Zehnerpotenz des aktuellen Wertes.
2. Führe eine Integerdivision (Wer DIV Potenz) durch und multipliziere das Ergebnis mit der ermittelten Zehnerpotenz.
3. Suche nächst höheren römischen Wert.
4. Versuche durch Subtrahieren mit dem erlaubten Prefix der römischen Zahl den Wert zu erreichen.
5. Ist dies möglich so ist die Subtraktionsregel anwendbar und das erste Paar wurde ermittelt.
    a. Subtrahiere den ermittelten Wert vom Zahlenwert
    b. Füge die ermittelten römischen Zahlen zum Ergebnis hinzu.
    c. Fahre fort mit Schritt 1 sofern der Rest <> 0.
6. Ist dies nicht möglich so ermittle den höchstmöglichen römischen Wert, führe eine Integerdivision am aktuellen Wert durch. Dies stellt die Anzahl dar wie oft die Zeichen verwendet werden müssen.
    a. Subtrahiere den ermittelten Wert vom Zahlenwert
    b. Füge die ermittelten römischen Zahlen zum Ergebnis hinzu.
    c. Fahre fort mit Schritt 1 sofern der Rest <> 0.

Beispiel:

Current value	Ten potenzy	Check substr.	Roman Char	Rest
999	10^2	1000 – 100 = 900	CM	999 – 900 = 99
99	10^2	100 – 10 = 90	XC	99 – 90 = 9
9	10^0 = 1	10 – 1 = 9	IX	9 -9 = 0

Ergebnis: CMXCIX

3.1.3 RomanToInt

In dieser Funktion sollen römische Zahlenwerte in ihre ganzzahlige Zahlendarstellung konvertiert werden, wobei fehlerhafte Eingaben wie eine ungültige Syntax oder Semantik erkannt und behandelt werden sollen. Hierbei sollen die Positionen der ermittelten fehlerhaften Eingaben gekennzeichnet und folgend auch ausgegeben werden.

Bevor mit der Konvertierung begonnen wird, soll über zwei ausgelagerte Funktionen zuerst die Syntax und anschließend die Semantik geprüft werden. Sollten hier Fehler auftreten, so sollen diese auf der aktuellen Instanz des ValueHolder gesetzt und in weiterer Folge behandelt werden.

Damit auch die Subtraktionsregel behandelt werden kann, müssen in diesem Algorithmus die römischen Paare ermittelt und korrekt ausgewertet werden.

Es wird folgender Ansatz angewendet:

Beispiel: MDCLXXIV

Current	Current + Next	Is Pair	Arabic value
M	MD	Invalid pair	1000
D	DC	Invalid pair	500
C	CL	Invalid pair	100
L	LX	Invalid pair	50
X	XX	Invalid pair	10
X	XI	Invalid pair	10
I	IV	Pair	4
			1674

3.2 Source

Folgend ist der Source des Programms RomanCalculator angeführt.

```
{
  This porgam is used to calculate the roman number representation form
  an arabic number representation and visa versa.
  There are two ways to calculate the roman number representation.

  1. With no considering of the subtraction rule
  2. With considering of the subtraction rule.
  3. Conversion of roman to arabic representation with syntax and semantic check
}
PROGRAM RomanNumberCalculator;

{ ################### Type definitions ############################## }
TYPE
  {
    Enumeration Type which specifies the legal roman numbers.
    Used to act as the index for the constant array which holds
    all of the roman numbers with their necessary information
  }
  RomanNumbers = (M, D, C, L, X, V, I);
  {
    Enumeration which specifies the action types which identify the action to be performed on
the ValueHolder instance.
  }
  ActionTypes = (ACTION_ROMAN_TO_INT, ACTION_INT_TO_ROMAN_1, ACTION_INT_TO_ROMAN_2,
NO_ACTION);
  {
    Enumeration Type which specifies the error codes which can occur during conversion
  }
  ErrorCodes = (INVALID_CHARACTER, INVALID_SEMANTIC, OVERFLOW, NO_ERROR);
  {
    Error Array Type which is used to hold the determined error values
  }
  ErrorArray = ARRAY [ActionTypes] OF ErrorCodes;
  {
    Type which holds the information of a roman number, its corresponding
    arabic number and valid roman number prefix (subtraction rule)
  }
  RomanArabic = RECORD
    roman: CHAR;
    romanEnum: RomanNumbers;
    arabic: INTEGER;
    romanPrefix: RomanNumbers;
  END;
  {
    ValueHolder type which is used to hold set value and result of the conversion
    and all occurred errors.
  }
  ValueHolder = RECORD
    roman: STRING;
    arabic: LONGINT;
```

```
    errors: ErrorArray;
    action: ActionTypes;
    errorMessage: STRING;
  END;
  {
    Array type used for the holding of the valid roman number instances
  }
  RomanArray = ARRAY[RomanNumbers] OF RomanArabic;
  {
    Array type which holds the translated roman string RomanArray instances
  }
  CharToRomanArray = ARRAY OF RomanArabic;


{ ################### Constant definitions ############################### }
CONST
  {
    Array constant to define all valid roman numbers from
    the highest value to the lowest value.
  }
  romanAsArabicValueArray: RomanArray = (
    (roman: 'M'; romanEnum: M; arabic: 1000; romanPrefix: C),
    (roman: 'D'; romanEnum: D; arabic: 500; romanPrefix: C),
    (roman: 'C'; romanEnum: C; arabic: 100; romanPrefix: X),
    (roman: 'L'; romanEnum: L; arabic: 50; romanPrefix: X),
    (roman: 'X'; romanEnum: X; arabic: 10; romanPrefix: I),
    (roman: 'V'; romanEnum: V; arabic: 5; romanPrefix: I),
    (roman: 'I'; romanEnum: I; arabic: 1; romanPrefix: I)
  );
  {
    Constant which specifies the highest valid arabic number
    which is able to be converted.
  }
  maxArabicValue: INTEGER = 4999;
  {
    Constant which specifies the lowest valid arabic number.
    0 is not allowed to be converted, because there is no
    roman number for this.
  }
  minArabicValue: INTEGER = 1;


{ ################### Util Functions and Procedures ############################### }
  {
    Function which creates a ValueHolder instance with the given values.
    This instance will not have any errors set.

    @param
        roman: the roman value to set on the ValueHolder instance
    @param
        arabic: the arabic value to be set on the ValueHolder instance
    @return
        the created ValueHolder instance
  }
  FUNCTION CreateValueHolder(roman: STRING; arabic: LONGINT; action: ActionTypes):
ValueHolder;
  VAR
    i: ActionTypes;
    holder: ValueHolder;
  BEGIN
    holder.roman := roman;
    holder.arabic := arabic;
    holder.errorMessage := '';
    holder.action := action;

    FOR i := Low(holder.errors) TO High(holder.errors) DO
      holder.errors[i] := NO_ERROR;

    CreateValueHolder := holder;
  END;

  {
    Converts a integer to a string representation.

    @param value:
              the integer to be converted to a string
```

```
    @return
        the string representation of the given integer value
  }
  FUNCTION ToString(value: LONGINT): STRING;
  VAR
    s: STRING;
  BEGIN

    Str(value, s);
    ToString := s

  END;

  {
    Calculates the ten potency of the given value, which must not be less than zero.
    @param
        value: the value to get container ten potency
    @return
        the ten potency contained by the given value, or -1 if value is less than 0
  }
  FUNCTION GetTensPontency(value: LONGINT): LONGINT;
  VAR
    pot: LONGINT;
    count: INTEGER;
  BEGIN

    count := 0;
    pot := 1;

    IF value >= 0 THEN BEGIN
      WHILE ((value DIV pot) >= 10) DO BEGIN
        count := count + 1;
        pot := pot * 10
      END;
      GetTensPontency := count;
    END
    ELSE
      GetTensPontency := -1;

  END;

  {
    Translates the given valid roman string to a array with the corresponding RomanArabic
instance set.
    Invalid characters will be ignored, but the array will have a random character set on the
missing indexes.
    So be aware to call this function only with a valid roman string.

    @param
        roman: the roman string to be translated to an array with its corresponding
RomanArabic instances
    @return
        the translated roman string represented by an array with the corresponding ROmanArabic
instances set
  }
  FUNCTION TranslateRomanString(roman: STRING): CharToRomanArray;
  VAR
    i: INTEGER;
    j: RomanNumbers;
    romanArabicArray: CharToRomanArray;
  BEGIN
    {
      Set the length of the array to fit character count of roman string
    }
    SetLength(romanArabicArray, Length(roman) + 1);
    {
      Iterate over each character and search for corresponding RomanArabic instances
    }
    FOR i:= 1 TO Length(roman) DO BEGIN
      FOR j := Low(romanAsArabicValueArray) TO High(romanAsArabicValueArray) DO BEGIN
        IF romanAsArabicValueArray[j].roman = roman[i] THEN BEGIN
          romanArabicArray[i] := romanAsArabicValueArray[j];
          BREAK;
        END;
      END;
    END;
```

```
  TranslateRomanString := romanArabicArray;
END;

{
  Function which validates the range of the given 'value'.

  @param
      value: the value which range shall be validated
  @param
      bottom: the bottom border of the given 'value'
  @param
      top: the top border of the given 'value'
  @return
      true if the value is within the defined borders, false otherwise or if bottom > top
}
FUNCTION ValidateRange(value, bottom, top: INTEGER): BOOLEAN;
BEGIN

  ValidateRange := false;

  IF (bottom <= top) AND (value >= bottom) AND (value <= top) THEN
    ValidateRange := true

END;

{
  Function which detects an occurred error on the ValueHolder instance.
  An error is detected when at least one array element is set with an
  error not equal to 'NO_ERROR'.

  @param
      span: the ValueHolder instance to detect set errors on.
  @return
      true if an error has been detected, false otherwise
}
FUNCTION HasErrors(value: ValueHolder): BOOLEAN;
VAR
  i: ActionTypes;
  code: ErrorCodes;
  error: BOOLEAN;
BEGIN

  error := false;
  i := Low(value.errors);
  code := value.errors[i];

  IF code <> NO_ERROR THEN
      error := true;

  WHILE (NOT error) AND (i < High(value.errors)) DO BEGIN
    Inc(i);
    code := value.errors[i];

    IF code <> NO_ERROR THEN
      error := true;

  END;

  HasErrors := error;

END;

{
  Function which validates if the given ValueHolder instance contains
  only valid roman characters on its roman attribute.

  @param
      value: the given ValuHolder instance to be validated
  @return
      the ValueHolder instance containing any errors which have occurred.
}
FUNCTION ValidateRomanSyntax(value: ValueHolder): ValueHolder;
VAR
  result: ValueHolder;
  i: INTEGER;
```

```
    j: RomanNumbers;
    found: BOOLEAN;
    errorMessage: STRING;
  BEGIN

    result := CreateValueHolder(value.roman, value.arabic, NO_ACTION);
    errorMessage := '';

    {
      Iterate over each roman character
    }
    IF Length(result.roman) > 0 THEN BEGIN
      FOR i := 1 TO Length(value.roman) DO BEGIN
        found := false;
        {
          Search for valid roman character
        }
        FOR j := Low(romanAsArabicValueArray) TO High(romanAsArabicValueArray) DO BEGIN
          IF romanAsArabicValueArray[j].roman = value.roman[i] THEN BEGIN
            found := true;
            BREAK;
          END;
        END;

        {
          Build error message with marked errors if current character is no valid roman
character.
        }
        IF NOT found THEN BEGIN
          result.errors[result.action] := INVALID_CHARACTER;
          errorMessage := errorMessage + '[' + value.roman[i] + ']'
        END
        ELSE
          errorMessage := errorMessage + value.roman[i];
      END;
    END
    ELSE BEGIN
      result.errors[result.action] := INVALID_CHARACTER;
      errorMessage := errorMessage + '[' + value.roman + ']'
    END;


    {
      Set error message if an error have occurred
    }
    IF result.errors[result.action] <> NO_ERROR THEN
      result.errorMessage := errorMessage;

    ValidateRomanSyntax := result;

  END;


  {
    Function which answers the question if the given RomanArabic instance has a
    lower value then all other RomanArabic instances before the given index
    It has a lower value when the roman number occurs on or after the given index.

    @param
        roman: the RomanArabic instance to check for a higher value
    @param
        idx: the idx where to start the comparison to the given instance
    @return
        true if the given RomanArabic instance has a higher value than all others in the
constant array
        started on given idx
  }
  FUNCTION IsLowerValue(roman: RomanArabic; idx: RomanNumbers):BOOLEAN;
  VAR
    i: RomanNumbers;
    current: RomanArabic;
  BEGIN
    IsLowerValue := false;

    FOR i := idx TO High(romanAsArabicValueArray) DO BEGIN
      current := romanAsArabicValueArray[i];
```

```
      IF current.roman = roman.roman THEN BEGIN
        IsLowerValue := true;
        EXIT;
      END;
    END;
  END;

  {
    Function which checks if the given ValueHolder instance contains a roman number with
    a correct syntax.
    It is assumed that the roman string only contains valid roman characters.
    If a incorrect syntax is detected a error will be created and set on the returned
    ValueHolder instance. Here all following invalid character are marked.
    E.g.: IMDMXLIVI -> I [M] [D] [M] X [L] IV [I]

    @param
        value: the ValueHolder instance to check for correct syntax on set roman string.
    @return
        the ValueHolder instance containing all occurred errors.
  }
  FUNCTION ValidateRomanSemantic(value: ValueHolder): ValueHolder;
    VAR
      result: ValueHolder;
      i, j, count, formerIdx: INTEGER;
      k: 1..2;
      temp: STRING;
      pairFound, error: BOOLEAN;
      romanArabicInst, formerRoman: RomanArabic;
      formerRomanPair: ARRAY[1..2] OF INTEGER;
      romanArabicArray: ARRAY OF RomanArabic;
    BEGIN

      {
        Initiate preconditions
      }
      pairFound := false;
      error := false;
      result := CreateValueHolder(value.roman, -1, NO_ACTION);
      romanArabicArray := TranslateRomanString(value.roman);
      formerIdx := 1;
      count := 1;

      FOR i := 1 TO High(romanArabicArray) DO BEGIN
        romanArabicInst := romanArabicArray[i];

        { Handle former single roman }
        IF i <> 1 THEN BEGIN
          IF NOT pairFound THEN BEGIN
            formerRoman := romanArabicArray[formerIdx];
            IF (IsLowerValue(romanArabicInst, formerRoman.romanEnum)) OR  (formerRoman.roman =
romanAsArabicValueArray[romanArabicInst.romanPrefix].roman) THEN BEGIN (* (formerRoman.roman =
romanArabicInst.roman) OR *)
              { set detected pair for validation on next step }
              IF (formerRoman.roman =
romanAsArabicValueArray[romanArabicInst.romanPrefix].roman) AND (romanArabicInst.roman <>
romanAsArabicValueArray[romanArabicInst.romanPrefix].roman) THEN BEGIN
                formerRomanPair[1] := formerIdx;
                formerRomanPair[2] := i;
                (* writeln('pair : ', romanArabicArray[formerRomanPair[1]].roman, '/',
romanArabicArray[formerRomanPair[2]].roman);  *)
                pairFound := true;
              END;
            END
            { Detected error }
            ELSE
              error := true;
          END

          { Handle former detected pair }
          ELSE BEGIN
            {
              Current roman character must not be equal to one of the detected pair.
            }
            FOR k := Low(formerRomanPair) TO High(formerRomanPair) DO BEGIN
              formerRoman := romanArabicArray[formerRomanPair[k]];
```

```
              IF (NOT IsLowerValue(romanArabicInst,
romanArabicArray[formerRomanPair[k]].romanEnum)) OR (formerRoman.roman =
romanArabicInst.roman) THEN BEGIN
                error := true;
                (* writeln('paired error: ', romanArabicInst.roman);*)
                BREAK;
              END;
            END;
            { If error has been detected compare pair to following roman }
            IF NOT error THEN BEGIN
              (* writeln('no error: ', romanArabicInst.roman);*)
              pairFound := false;
            END;
          END;

        END;

        { Handle multiple following same roman (max 4 occurrences allowed) }
        IF NOT error THEN BEGIN
          count := 0;
          temp := '[';
          { Search for same occurrences and build error string }
          FOR j := i TO High(romanArabicArray) DO BEGIN
            IF romanArabicInst.roman = romanArabicArray[j].roman THEN BEGIN
              count := count + 1;
              temp := temp + romanArabicInst.roman;
            END
            ELSE
              BREAK;
          END;
          { If to much found set proper message and set idx of last element which is the next
after current.roman }
          IF count > 4 THEN BEGIN
            temp := temp + ']';
            result.errorMessage := result.errorMessage + temp;
            Dec(j);
            i := j;
            error := true;
          END;
        END;

        { Handle detected error }
        IF error THEN BEGIN
          error := false;
          result.errors[result.action] := INVALID_SEMANTIC;
          { If no multiple occurrences have been occurred mark current error }
          IF count < 5 THEN BEGIN
            result.errorMessage := result.errorMessage + '[' + romanArabicInst.roman + ']';
          END;
        END
        ELSE BEGIN
          { Add valid roman to error message }
          result.errorMessage := result.errorMessage + romanArabicInst.roman;
        END;
        { preconditions for next step }
        formerIdx := i;
        count := 1;
      END;

      ValidateRomanSemantic := result;

  END;

  {
    Gets the current roman representation of the given value for the given valid roman
    number. If the current roman does not fit for this value an empty string is returned.

    @param
        araic: the arabic number to get the proper roman number for
    @param
        idx: the current index of the RomanNumbersArray to check if this roman number fits the
             given arabic value
    @return
        rest: the rest of the arabic number, means given arabic - calculated roman
number/arabic value
        roman: the roman number if the current roman number fits, an empty string otherwise
```

```
  }
  PROCEDURE GetCurrentRoman(arabic: LONGINT; idx: RomanNumbers; VAR rest: LONGINT; VAR roman:
STRING);
  VAR
    romanArabicInst: RomanArabic;
    temp, i: INTEGER;
  BEGIN

    { Initiate preconditions }
    romanArabicInst := romanAsArabicValueArray[idx];
    rest := arabic;
    roman := '';

    {
      Search for proper roman character which can be used for a part of the given arabic value
    }
    IF arabic >= romanArabicInst.arabic THEN BEGIN
      temp := arabic DIV romanArabicInst.arabic;

      if temp > 0 THEN
        FOR i := 1 TO temp DO
          roman := roman + romanArabicInst.roman;

      rest := arabic - (temp *  romanArabicInst.arabic);
    END;
  END;

  {
    Gets the roman representation for the given arabic number.
    The subtraction rule is not considered here.

    @param
        arabic: the arabic number to be converted
    @return
        the calculated roman representation represented by a string
  }
  FUNCTION IntToRoman(value: ValueHolder): ValueHolder;
  VAR
    i: RomanNumbers;
    currentRoman: STRING;
    first: BOOLEAN;
    result: ValueHolder;
  BEGIN

    { Init preconditions }
    first := true;

    result := CreateValueHolder('', value.arabic, value.action);

    { Check for border overflow }
    IF ValidateRange(result.arabic, minArabicValue, maxArabicValue) THEN BEGIN
      i := LOW(romanAsArabicValueArray);

      WHILE (i <= High(romanAsArabicValueArray)) AND (result.arabic > 0) DO BEGIN
        IF NOT first THEN
          Inc(i);

        { Calculate next roman character which fits current arabic number }
        GetCurrentRoman(result.arabic, i, result.arabic, currentRoman);
        result.roman := result.roman + currentRoman;

        first := false;

      END;
    END
    { Add error if border overflows }
    ELSE BEGIN
      result.errors[value.action] := OVERFLOW;
      result.errorMessage := 'The given arabic value overflows borders: min ' +
ToString(minArabicValue) + ' / max: ' + ToString(maxArabicValue);
    END;

    result.arabic := value.arabic;
    IntToRoman := result;

  END;
```

```
  {
    Gets the roman representation for the given arabic number.
    The subtraction rule is considered here.

    @param
        arabic: the arabic number to be converted
    @return
        the ValueHolder instance containing the result or all occurred errors
  }
  FUNCTION IntToRoman2(value: ValueHolder): ValueHolder;
  VAR
    i: RomanNumbers;
    j, temp, potValue: INTEGER;
    currentRoman: STRING;
    first, substractionDone: BOOLEAN;
    result: ValueHolder;
  BEGIN

    {
      Initiate preconditions
    }
    result :=  CreateValueHolder('', value.arabic, value.action);
    first := true;
    i := LOW(romanAsArabicValueArray);

    {
      Validate arabic set borders
    }
    IF ValidateRange(result.arabic, minArabicValue, maxArabicValue) THEN BEGIN
      {
        Iterate from highest to lowest roman number and try to get proper arabic
representation
      }
      WHILE (i <= High(romanAsArabicValueArray)) AND (result.arabic > 0) DO BEGIN
        IF NOT first THEN
          Inc(i);

        first := false;
        substractionDone := false;

        {
          Check for subtraction rule
        }
        IF i > Low(RomanNumbers) THEN BEGIN
          Dec(i);
          potValue := 1;
          { Calculate roman representation with subtraction rule }
          FOR j := 1 TO GetTensPontency(result.arabic) DO
            potValue := potValue * 10;
          temp := (result.arabic DIV potValue) * potValue;
          IF (romanAsArabicValueArray[i].arabic -
romanAsArabicValueArray[romanAsArabicValueArray[i].romanPrefix].arabic = temp) THEN BEGIN
            result.roman := result.roman +
romanAsArabicValueArray[romanAsArabicValueArray[i].romanPrefix].roman +
romanAsArabicValueArray[i].roman;
            result.arabic := result.arabic - temp;
            substractionDone := true;
          END;
          Inc(i);
        END;

        {
          Normal roman representation resolving
        }
        IF NOT substractionDone THEN BEGIN
          GetCurrentRoman(result.arabic, i, result.arabic, currentRoman);
          result.roman := result.roman + currentRoman;
        END;

      END;

    END
    { Handle border overflow }
    ELSE BEGIN
      result.errors[result.action] := OVERFLOW;
```

```
      result.errorMessage := 'The given arabic value overflows borders: min ' +
ToString(minArabicValue) + ' / max: ' + ToString(maxArabicValue);
    END;

    result.arabic := value.arabic;
    IntToRoman2 := result;
  END;

  {
    Function which calculates the arabic representation out of a roman
    number representation.
    This function can handle roman representations with subtraction rule
    contained, or not or mixed.

    @param
        valueHolder: the ValueHolder instance to retrieve value from
    @return
        A ValueHolder instance which contains the calculated arabic value and all occurred
errors.
  }
  FUNCTION RomanToInt(value: ValueHolder): ValueHolder;
  VAR
    i: INTEGER;
    result: ValueHolder;
    current, next: RomanArabic;
    add: BOOLEAN;
    romanArabicArray: ARRAY OF RomanArabic;
  BEGIN

    {
      Validate for valid characters and syntax
    }
    result := ValidateRomanSyntax(value);
    IF NOT HasErrors(result) THEN
      result := ValidateRomanSemantic(result);

    {
      Initiate preconditions
    }
    add := false;
    result.arabic := 0;

    {
      Convert if no error occurred during validation
    }
    IF NOT HasErrors(result) THEN BEGIN
      {
        Translate string to RomanArabic instance array
      }
      romanArabicArray := TranslateRomanString(result.roman);
      FOR i := 1 TO High(romanArabicArray) DO BEGIN
        {
          Calculate with subtraction rule considered.
        }
        current := romanArabicArray[i];
        (*writeln('i: ', i , ' roman ', current.arabic);*)
        IF i < High(romanArabicArray) THEN BEGIN
          next := romanArabicArray[(i + 1)];
          IF (romanAsArabicValueArray[next.romanPrefix].roman = current.roman) AND
(romanAsArabicValueArray[next.romanPrefix].roman <> next.roman) THEN BEGIN
            result.arabic := result.arabic + (next.arabic - current.arabic);
            Inc(i);
          END
          ELSE
            add := true;
        END
        ELSE
          add := true;

        {
          Calculate plain roman numbers
        }
        IF add THEN BEGIN
          add := false;
          result.arabic := result.arabic + current.arabic;
        END;
```

```
        {
          Validate if result overflows max representable arabic value
        }
        IF result.arabic > maxArabicValue THEN BEGIN
          result.errors[value.action] := OVERFLOW;
          result.arabic := 0;
          result.errorMessage := 'Roman String overflows maxValue "'+ ToString(maxArabicValue)
+ '" !!!';
        END;


      END;
    END;

    RomanToInt := result;
  END;

  {
    Procedure for testing the main functions.
    This procedure ahndles the occurres error and prints them to the cosnole.

    @param
        value: the ValueHolder instance to use for the test
  }
  PROCEDURE Test(value: ValueHolder);
  VAR
    validAction: BOOLEAN;
    result: ValueHolder;
  BEGIN

    validAction := true;
    WriteLn('-----------------------------------------');

    IF value.action = ACTION_ROMAN_TO_INT THEN
      result := RomanToInt(value)
    ELSE IF value.action = ACTION_INT_TO_ROMAN_1 THEN
      result := IntToRoman(value)
    ELSE IF value.action = ACTION_INT_TO_ROMAN_2 THEN
      result := IntToRoman2(value)
    ELSE
      validAction := false;

    IF validAction THEN BEGIN
      WriteLn('Action           : ', result.action);
      WriteLn('Rroman           : ', result.roman);
      WriteLn('Arabic           : ', result.arabic);
      IF HasErrors(result) THEN BEGIN
       WriteLn('Error Code       : ', result.errors[result.action]);
        WriteLn('Error message    : ', result.errorMessage);
      END
    END;

    WriteLn('-----------------------------------------');
  END;

BEGIN

  WriteLn('');
  WriteLn('########### IntToRoman ###################');
  Test(CreateValueHolder('', 0, ACTION_INT_TO_ROMAN_1));
  Test(CreateValueHolder('', 5000, ACTION_INT_TO_ROMAN_1));
  Test(CreateValueHolder('', 4999, ACTION_INT_TO_ROMAN_1));
  Test(CreateValueHolder('', 1000, ACTION_INT_TO_ROMAN_1));
  Test(CreateValueHolder('', 500, ACTION_INT_TO_ROMAN_1));
  Test(CreateValueHolder('', 999, ACTION_INT_TO_ROMAN_1));

  WriteLn('');
  WriteLn('########### IntToRoman2 ##################');
  { Invalid values }
  Test(CreateValueHolder('', 0, ACTION_INT_TO_ROMAN_2));
  Test(CreateValueHolder('', 5000, ACTION_INT_TO_ROMAN_2));
  Test(CreateValueHolder('', 4999, ACTION_INT_TO_ROMAN_2));
  Test(CreateValueHolder('', 1000, ACTION_INT_TO_ROMAN_2));
  Test(CreateValueHolder('', 500, ACTION_INT_TO_ROMAN_2));
  Test(CreateValueHolder('', 999, ACTION_INT_TO_ROMAN_2));
```

```
  WriteLn('');
  WriteLn('############ RomanToInt ##################');
  { Invalid roman count }
  WriteLn('----------- Invalid characters -----------');
  Test(CreateValueHolder('', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder(' MM12MXX ', 0, ACTION_ROMAN_TO_INT));
  WriteLn('----------- Invalid roman count ----------');
  Test(CreateValueHolder('MMMMMCMXCIX', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder('MMMMCMXXXXXIV', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder('CMXLVVVVVIIII', 0, ACTION_ROMAN_TO_INT));
  WriteLn('----------- Invalid pair -----------------');
  Test(CreateValueHolder('CMCDVI', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder('MMMMCMCDXXXXIV', 0, ACTION_ROMAN_TO_INT));
  WriteLn('----------- Mixed ------------------------');
  Test(CreateValueHolder('CMCDXXXXXIVIIIII', 0, ACTION_ROMAN_TO_INT));
  WriteLn('----------- Correct behavior -------------');
  Test(CreateValueHolder('MMMMXLIV', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder('MMMMCMXLIIII', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder('MMMMXXXXVVVVIIII', 0, ACTION_ROMAN_TO_INT));
  Test(CreateValueHolder('MMMMXXXXVVVV', 0, ACTION_ROMAN_TO_INT));

END.
```

3.3 Tests

Folgend sind die Test angeführt, die oben beschriebenen Algorithmen testen.

3.3.1 IntToRoman (ohne Subtraktionsregel)

Folgend sind die Tests angeführt, die die Konvertierung von einem ganzzahligen Zahlenwert zu seiner römischen Zahlendarstellung ohne Berücksichtigung der Subtraktionsregel testen.

```
############ IntToRoman ####################

------------------------------------------
Action            : ACTION_INT_TO_ROMAN_1
Rroman            :
Arabic            : 0
Error Code        : OVERFLOW
Error message     : The given arabic value overflows borders: min 1 / max: 4999
------------------------------------------

------------------------------------------
Action            : ACTION_INT_TO_ROMAN_1
Rroman            :
Arabic            : 5000
Error Code        : OVERFLOW
Error message     : The given arabic value overflows borders: min 1 / max: 4999
------------------------------------------

------------------------------------------
Action            : ACTION_INT_TO_ROMAN_1
Rroman            : MMMMDCCCCLXXXXVIIII
Arabic            : 4999
------------------------------------------

------------------------------------------
Action            : ACTION_INT_TO_ROMAN_1
Rroman            : M
Arabic            : 1000
------------------------------------------

------------------------------------------
Action            : ACTION_INT_TO_ROMAN_1
Rroman            : D
Arabic            : 500
------------------------------------------

------------------------------------------
Action            : ACTION_INT_TO_ROMAN_1
Rroman            : DCCCCLXXXXVIIII
Arabic            : 999
------------------------------------------
```

Es ist zu sehen, das die Grenzwerte geprüft und auch behandelt werden, sollte der Grenzwert hinaufgesetzt werden, so würde dies kein Problem darstellen, aber es in den römischen Zahlen sind nur Darstellungen bis 5000 und größer 0 erlaubt.
Es ist ebenso zu sehen, das die Subtraktionsregel nicht angewendet wird.

3.3.2 IntToRoman (mit Subtraktionsregel)

Folgend sind die Tests angeführt, die die Konvertierung von einem ganzzahligen Zahlenwert zu seiner römischen Zahlendarstellung unter der Berücksichtigung der Subtraktionsregel testen.

```
############ IntToRoman2 ####################

-----------------------------------------
Action            : ACTION_INT_TO_ROMAN_2
Rroman            :
Arabic            : 0
Error Code        : OVERFLOW
Error message     : The given arabic value overflows borders: min 1 / max: 4999
-----------------------------------------

-----------------------------------------
Action            : ACTION_INT_TO_ROMAN_2
Rroman            :
Arabic            : 5000
Error Code        : OVERFLOW
Error message     : The given arabic value overflows borders: min 1 / max: 4999
-----------------------------------------

-----------------------------------------
Action            : ACTION_INT_TO_ROMAN_2
Rroman            : MMMMCMXCIX
Arabic            : 4999
-----------------------------------------

-----------------------------------------
Action            : ACTION_INT_TO_ROMAN_2
Rroman            : M
Arabic            : 1000
-----------------------------------------

-----------------------------------------
Action            : ACTION_INT_TO_ROMAN_2
Rroman            : D
Arabic            : 500
-----------------------------------------

-----------------------------------------
Action            : ACTION_INT_TO_ROMAN_2
Rroman            : CMXCIX
Arabic            : 999
-----------------------------------------
```

Hierbei ist zu sehen dass sich die Fehlerbehandlung wie im vorherigen Algorithmus verhält, aber dass beim Ermitteln der römischen Zahlen die Subtraktionsregel berücksichtigt wurde.

3.3.3 RomanToInt

Folgend sind die Tests angeführt, welche die Konvertierung von einer römischen Darstellung in eine ganzzahlige Darstellung testen. Hierbei wurden exzessive Tests durchgeführt, da hier die meiste Logik enthalten ist und dies der komplexeste der Algorithmen ist. Es wird getestet, wie die die ungültige Syntax und Semantik erkannt und behandelt werden.
Hierbei ist anzumerken, dass zuerst die Syntax und anschließend die Semantik geprüft wird. Sollten Syntaxfehler auftreten, so werden diese behandelt und die Semantikfehler außer Acht gelassen. Es greift also eine zwei Stufen Validierung.

```
---------- Invalid characters ----------
----------------------------------------
Action            : NO_ACTION
Rroman            :
Arabic            : 0
Error Code        : INVALID_CHARACTER
Error message     : []
----------------------------------------
----------------------------------------
Action            : NO_ACTION
Rroman            :  MMMMM12MXX
Arabic            : 0
Error Code        : INVALID_CHARACTER
Error message     : [ ]MMMMM[1][2]MXX[ ]
----------------------------------------
```

Hier werden die Sematikfehler aufgrund erkannter Syntaxfehler außer Acht gelassen.

```
---------- Invalid roman count ----------
-----------------------------------------
Action            : NO_ACTION
Rroman            : MMMMMCMXCIX
Arabic            : 0
Error Code        : INVALID_SEMANTIC
Error message     : [MMMMM]CMXCIX
-----------------------------------------
-----------------------------------------
Action            : NO_ACTION
Rroman            : MMMMCMXXXXXIV
Arabic            : 0
Error Code        : INVALID_SEMANTIC
Error message     : MMMMCM[XXXXX]IV
-----------------------------------------
-----------------------------------------
Action            : NO_ACTION
Rroman            : CMXLVVVVVIIII
Arabic            : 0
Error Code        : INVALID_SEMANTIC
Error message     : CMXL[VVVVV]IIII
-----------------------------------------
```

Hierbei werden ungültige Folgen (mehr als 4 römische Zahlen hintereinander nicht erlaubt) als Fehler erkannt.

```
---------- Invalid pair ----------
----------------------------------
Action            : NO_ACTION
Rroman            : CMCDVI
Arabic            : 0
Error Code        : INVALID_SEMANTIC
Error message     : CM[C][D]VI
----------------------------------
----------------------------------
Action            : NO_ACTION
Rroman            : MMMMCMCDXXXXIV
Arabic            : 0
Error Code        : INVALID_SEMANTIC
Error message     : MMMMCM[C][D]XXXXIV
----------------------------------
```

Hierbei werden römische Zahlen nach Paaren (Subtraktionsregel) als Fehler erkannt. Römische Zahlen, die danach folgen werden wieder als gültig angesehen, sofern sie nicht eine der des Paares sind oder einen größeren Wert haben als eines der beiden.

```
---------- Mixed ----------------------
---------------------------------------
Action            : NO_ACTION
Rroman            : CMCDXXXXXIVIIIII
Arabic            : 0
Error Code        : INVALID_SEMANTIC
Error message     : CM[C][D][XXXXX]IV[I][I][I][I][I]
---------------------------------------
```

Hierbei werden auch gemischte Fehler, außer Syntaxfehler, die vorherig behandelt werden, erkannt.

```
---------- Correct behavior ----------
--------------------------------------
Action          : NO_ACTION
Rroman          : MMMMXLIV
Arabic          : 4044
--------------------------------------
--------------------------------------
Action          : NO_ACTION
Rroman          : MMMMCMXLIIII
Arabic          : 4944
--------------------------------------
--------------------------------------
Action          : NO_ACTION
Rroman          : MMMMXXXXVUUUUIIII
Arabic          : 4064
--------------------------------------
--------------------------------------
Action          : NO_ACTION
Rroman          : MMMMXXXXVUUUU
Arabic          : 4060
--------------------------------------
```

Wenn korrekte römische Zahlen angegeben werden, dann erfolgt eine korrekte Berechnung. Es ist anzumerken, dass römische Zahlen, die die Subtraktionsregel und die normale Darstellung beinhalten, korrekt berechnet werden können.

4 Diskussion

Im Nachhinein hätte vielleicht ein anderer Ansatz besser funktioniert oder hätte weniger Code verursacht, ebenso die Möglichkeit dynamische Felder zu verwenden. Aber da ich bereits diesen Ansatz gewählt hatte und die Zeit knapp bemessen ist, ist der Code so wie er ist.
Vor Allem der Algorithmus ValidateRomanSemantic hätte vielleicht besser oder schlanker ausfallen können. Ich habe mich dazu entschieden, die Teilfunktionen nicht auszulagern, da diese nicht wiederverwendet werden.