Getting Started with Conversion Agent

SAP Conversion Agent by Informatica

(Version 8.5)

Getting Started with Conversion Agent

Version 8.5
June 2008

Part Number: DT-GST-86000-0001

Table of Contents

Preface . . . vii

Quick Reference . . . vii

Obtaining Conversion Agent Documentation . . . ix

Chapter 1: Introducing Conversion Agent . . . 1

Overview . . . 1

- Introduction to XML . . . 1
- How Conversion Agent Works . . . 2
- Using Conversion Agent in Integration Applications . . . 3

Installation . . . 4

- System Requirements . . . 4
- Installation Procedure . . . 4
- Default Installation Folder . . . 4
- Tutorials and Workspace Folders . . . 4
- Exercises and Solutions . . . 4
- XML Editor . . . 5

Chapter 2: Basic Parsing Techniques . . . 7

Overview . . . 7

Opening Conversion Agent Studio . . . 8

Importing the Tutorial_1 Project . . . 9

A Brief Look at the Studio Window . . . 10

Defining the Structure of a Source Document . . . 12

- Correcting Errors in the Parser Configuration . . . 17
- Techniques for Defining Anchors . . . 17
- Tab-Delimited Format . . . 18

Running the Parser . . . 18

- Running the Parser on Additional Source Documents . . . 19

Points to Remember . . . 20

What's Next? . . . 21

Chapter 3: Defining an HL7 Parser . . . 23

Overview . . . 23

- Requirements Analysis . . . 23

Creating a Project . . . 25

- Using XSD Schemas in Transformations . . . 27

Defining the Anchors . . . 27

- Editing the IntelliScript . . . 30

Testing the Parser . . . 30

Points to Remember . . . 32

Chapter 4: Positional Parsing of a PDF Document 35

Overview 35

Requirements Analysis 36

Creating the Project 38

Defining the Anchors 40

More About Search Scope 44

Defining the Nested Repeating Groups 44

Basic and Advanced Properties 49

Using an Action to Compute Subtotals 50

Actions 52

Potential Enhancement: Handling Page Breaks 52

Points to Remember 52

Chapter 5: Parsing Word and HTML Documents 55

Overview 55

Scope of the Exercise 56

Prerequisites 56

Requirements Analysis 56

Source Document 56

XML Output 57

The Parsing Problem 58

Creating the Project 59

Defining a Variable 60

Parsing the Name and Address 61

Why the Output Contains Empty Elements 63

Parsing the Optional Currency Line 63

Parsing the Order Table 65

Why the Output does not Contain HTML Code 68

Using Count to Resolve Ambiguities 70

Using Transformers to Modify the Output 70

Global Components 73

Testing the Parser on Another Source Document 73

Points to Remember 74

Chapter 6: Defining a Serializer 77

Overview 77

Prerequisite 77

Requirements Analysis 77

Creating the Project 78

Determining the Project Folder Location 79

Configuring the Serializer 80

Calling the Serializer Recursively 81

Defining Multiple Components in a Project 83

Points to Remember 84

Chapter 7: Defining a Mapper 85

Overview 85

Requirements Analysis 85

Creating the Project 86

Configuring the Mapper 87

Points to Remember 88

Chapter 8: Running Conversion Agent Engine. 91

Overview 91

Deploying a Transformation as a Service 92

COM API Application 92

Source Code 92

Explanation of the API Calls 93

Running the COM API Application 94

Points to Remember 95

Index 97

Preface

Getting Started with Conversion Agent is written for developers and analysts who are responsible for implementing transformations. As you read it, you will perform several hands-on exercises that teach how to use Conversion Agent in real-life transformation scenarios. When you finish the lessons, you will be familiar with the Conversion Agent procedures, and you will be able to apply them to your own transformation needs.

We recommend that all users perform the first and second lessons, which teach the basic techniques for working in Conversion Agent Studio. You can then proceed through the other lessons in sequence, or you can skim the chapters and skip to the ones that you need.

Quick Reference

The following table is a guide to the Conversion Agent concepts, features, and components that are introduced and explained in each lesson.

Concept	Feature or component	For more information, see...
Working in Conversion Agent Studio	Viewing IntelliScript	"Basic Parsing Techniques" on page 7
	Editing IntelliScript	"Defining an HL7 Parser" on page 23
	Multiple script files	"Defining a Serializer" on page 77
	Color coding of anchors	"Basic Parsing Techniques" on page 7
	Basic and advanced properties	"Positional Parsing of a PDF Document" on page 35
	Global components	"Parsing Word and HTML Documents" on page 55
Projects	Importing	"Basic Parsing Techniques" on page 7
	Creating	"Defining an HL7 Parser" on page 23
	Containing multiple parsers or serializers	"Defining a Serializer" on page 77
	Project properties	"Defining a Serializer" on page 77
	Determining the project folder location	"Defining a Serializer" on page 77

Concept	Feature or component	For more information, see...
Parsers	Example source documents	“Basic Parsing Techniques” on page 7
	Creating	“Defining an HL7 Parser” on page 23
	Running	“Basic Parsing Techniques” on page 7
	Viewing results	“Basic Parsing Techniques” on page 7
	Testing on example source	“Defining an HL7 Parser” on page 23
	Testing on additional source documents	“Basic Parsing Techniques” on page 7 “Parsing Word and HTML Documents” on page 55
	Document processors	“Positional Parsing of a PDF Document” on page 35 “Parsing Word and HTML Documents” on page 55
	Calling a secondary parser	“Defining a Serializer” on page 77
Formats	Text	“Basic Parsing Techniques” on page 7
	Tab-delimited	“Basic Parsing Techniques” on page 7
	HL7	“Defining an HL7 Parser” on page 23
	Positional	“Positional Parsing of a PDF Document” on page 35
	PDF	“Positional Parsing of a PDF Document” on page 35
	Microsoft Word	“Parsing Word and HTML Documents” on page 55
	HTML	“Parsing Word and HTML Documents” on page 55
Data holders	Using an XSD schema	“Basic Parsing Techniques” on page 7
	Adding a schema to a project	“Defining an HL7 Parser” on page 23
	Creating and editing schemas	“Defining an HL7 Parser” on page 23
	Using multiple schemas	“Defining a Mapper” on page 85
	Variables	“Parsing Word and HTML Documents” on page 55
Anchors	`Marker`	“Basic Parsing Techniques” on page 7
	`Marker` with count	“Parsing Word and HTML Documents” on page 55
	`Content`	“Basic Parsing Techniques” on page 7
	`Content` with positional offsets	“Positional Parsing of a PDF Document” on page 35
	`Content` with opening and closing markers	“Parsing Word and HTML Documents” on page 55
	`EnclosedGroup`	“Parsing Word and HTML Documents” on page 55
	`RepeatingGroup`	“Defining an HL7 Parser” on page 23
	Search scope	“Positional Parsing of a PDF Document” on page 35
	Nested `RepeatingGroup`	“Positional Parsing of a PDF Document” on page 35
	Newlines as markers and separators	“Positional Parsing of a PDF Document” on page 35
Transformers	Default transformers	“Parsing Word and HTML Documents” on page 55
	`AddString`	“Parsing Word and HTML Documents” on page 55
	`Replace`	“Parsing Word and HTML Documents” on page 55
Actions	`SetValue`	“Positional Parsing of a PDF Document” on page 35
	`CalculateValue`	“Positional Parsing of a PDF Document” on page 35
	`Map`	“Defining a Mapper” on page 85

Concept	Feature or component	For more information, see...
Serializers and serialization anchors	Creating	“Defining a Serializer” on page 77
	`ContentSerializer`	“Defining a Serializer” on page 77
	`RepeatingGroupSerializer`	“Defining a Serializer” on page 77
	`EmbeddedSerializer`	“Defining a Serializer” on page 77
	Calling a secondary serializer	“Defining a Serializer” on page 77
Mappers and mapper anchors	Creating	“Defining a Mapper” on page 85
	`RepeatingGroupMapping`	“Defining a Mapper” on page 85
Testing	Using color coding	“Defining an HL7 Parser” on page 23
	Viewing events	“Basic Parsing Techniques” on page 7
	Interpreting events	“Defining an HL7 Parser” on page 23
	Testing and debugging techniques	“Defining an HL7 Parser” on page 23
	Selecting which parser or serializer to run	“Defining a Serializer” on page 77
Running services in Conversion Agent Engine	Deploying a Conversion Agent service	“Running Conversion Agent Engine” on page 91
	API	“Running Conversion Agent Engine” on page 91

Obtaining Conversion Agent Documentation

On Windows platforms, the Conversion Agent documentation is supplied as online help with the software. You can download PDF copies of the Conversion Agent manuals from:

http://help.sap.com/saphelp_nw70/helpdata/en/43/fc39c16bfb025ee10000000a1553f7/content.htm

CHAPTER 1

Introducing Conversion Agent

This chapter includes the following topics:

- Overview, 1
- Installation, 4

Overview

SAP Conversion Agent enables you to transform data efficiently from any format to any other format, via XML-based representations.

To configure a transformation, you can use approaches such as parsing by example. According to this approach, you can teach Conversion Agent how to convert data to XML by marking up an example in a visual editor environment. You do not need to do any programming to configure the transformation. You can configure even a complex transformation in just a few hours or days, saving weeks or months of programming time.

Conversion Agent can process fully structured, semi-structured, or unstructured data. You can configure the software to work with text, binary data, messaging formats, HTML pages, PDF documents, word-processor documents, and any other format that you can imagine.

You can configure a Conversion Agent parser to transform the data to any standard or custom XML vocabulary. In the reverse direction, you can configure a Conversion Agent serializer to transform the XML data to any other format. You can configure a Conversion Agent mapper to perform XML to XML transformations.

This book is a tutorial introduction, intended for users who are new to Conversion Agent. As you perform the exercises in this book, you will learn to configure and run your own transformations.

Introduction to XML

XML (Extensible Markup Language) is the de facto standard for cross-platform information exchange. For the benefit of Conversion Agent users who may be new to XML, we present a brief introduction here. If you are already familiar with XML, you can skip this section.

The following is an example of a small XML document:

```
<Company industry="metals">
  <Name>Ore Refining Inc.</Name>
  <WebSite>http://www.ore_refining.com</WebSite>
  <Field>iron and steel</Field>
  <Products>
    <Product id="1">cast iron</Product>
    <Product id="2">stainless steel</Product>
  </Products>
```

```
</Company>
```

This sample is called a well-formed XML document because it complies with the basic XML syntactical rules. It has a tree structure, composed of elements. The top-level element in this example is called `Company`, and the nested elements are `Name`, `WebSite`, `Field`, `Products`, and `Product`.

Each element begins and ends with tags, such as `<Company>` and `</Company>`. The elements may also have attributes. For example, `industry` is an attribute of the `Company` element, and `id` is an attribute of the `Product` element.

To explain the hierarchical relationship between the elements, we sometimes refer to parent and child elements. For example, the `Products` element is the child of `Company` and the parent of `Product`.

The particular system of elements and attributes is called an XML vocabulary. The vocabulary can be customized for any application. In the example of a small XML document above, we made up a vocabulary that might be suitable for a commercial directory.

The vocabulary can be formalized in a syntax specification called a schema. The schema might specify, for example, that `Company` and `Name` are required elements, that the other elements are optional, and that the value of the `industry` attribute must be a member of a predefined list. If an XML document conforms to a rigorous schema definition, the document is said to be valid, in addition to being well-formed.

To make the XML document easier to read, we have indented the lines to illustrate how the elements are nested. The indentation and whitespace are not essential parts of the XML syntax. We could have written a long, unbroken string such as the following, which does not contain any extra whitespace:

```
<Company industry="metals"><Name>Ore Refining Inc.</Name><WebSite>
http://www.ore_refining.com</WebSite><Field>iron and steel</Field>
<Products><Product id="1">cast iron</Product><Product id="2">stainless
steel</Product></Products></Company>
```

The unbroken-string representation is identical to the indented representation. In fact, a computer might store the XML as a string like this. The indented representation is how XML is conventionally presented in a book or on a computer screen because it is easier to read.

For More Information

You can get information about XML from many books, articles, or web sites. For an excellent tutorial, see http://www.w3schools.com. To obtain copies of the XML standards, see http://www.w3.org.

How Conversion Agent Works

The Conversion Agent system has two main components:

Component	Description
Conversion Agent Studio	The design and configuration environment of Conversion Agent.
Conversion Agent Engine	The transformation engine.

Conversion Agent Studio

The Studio is a visual editor environment where you can design and configure transformations such as parsers, serializers, and mappers.

Use the Studio to configure Conversion Agent to process data of a particular type. If you are building a parser, you can use a select-and-click approach to identify the data fields in an example source document, and define how the software should transform the fields to XML. This procedure is called parsing by example.

Note that we use the term document in the broadest possible sense. A document can contain text or binary data, and it can have any size. It can be stored or accessed in a file, buffer, stream, URL, database, messaging system, or any other location.

Conversion Agent Engine

Conversion Agent Engine is an efficient transformation processor. It has no user interface. It works entirely in the background, executing the transformations that you have previously defined in Studio.

To move a transformation from the Studio to the Engine, you must deploy the transformation as a Conversion Agent service.

An integration application can communicate with the Engine by submitting requests in a number of ways, for example, by calling the Conversion Agent API. A request specifies the data to be transformed and the service that should perform the transformation. The Engine executes the request and returns the output to the calling application.

Using Conversion Agent in Integration Applications

The following paragraphs present some typical examples of how transformations are used in system integration applications.

As you perform the exercises in this book, you will get experience using these types of transformations. The chapter on HL7 parsers, for example, illustrates how to use Conversion Agent to parse HL7 messages. For more information, see "Defining an HL7 Parser" on page 23.

The chapters on positional parsing and parsing Word documents describe parsers that process various types of unstructured documents. For more information, see:

- "Positional Parsing of a PDF Document" on page 35
- "Parsing Word and HTML Documents" on page 55.

HL7 Integration

HL7 is a messaging standard used in the health industry. HL7 messages have a flexible structure that supports optional and repetitive data fields. The fields are separated by a hierarchy of delimiter symbols.

In a typical integration application, a major health maintenance organization (HMO) uses Conversion Agent to transform messages that are transmitted to and from its HL7-based information systems.

Processing PDF Forms

The PDF file format has become a standard for formatted document exchange. The format permits users to view fully formatted documents—including the original layout, fonts, and graphics,—on a wide variety of supported platforms. PDF files are less useful for information processing, however, since applications cannot access and analyze their unstructured, binary representation of data

Conversion Agent solves this problem by enabling conversion of PDF documents to an XML representation. For example, Conversion Agent can convert invoices that suppliers send in PDF format to XML, for storage in a database.

Converting HTML Pages to XML

Information in HTML documents is usually presented in unstructured and unstandardized formats. The goal of the HTML presentation is visual display, rather than information processing.

Conversion Agent has many features that can navigate, locate, and store information found in HTML documents. The software enables conversion of information from HTML to a structured XML representation, making the information accessible to software applications. For example, retailers who present their stock on the web can convert the information to XML, letting them share the information with a clearing house or with other retailers.

Installation

Before you continue in this book, you should install the Conversion Agent software, if you have not already done so.

For more information about the system requirements, installation, and registration, see the *Conversion Agent Administrator Guide*. The following paragraphs contain brief instructions to help you get started.

System Requirements

To perform the exercises in this book, you should install Conversion Agent on a computer that meets the following minimum requirements:

- Microsoft Windows 2000, XP Professional, or 2003 Server
- Microsoft Internet Explorer, version 6.0 or higher
- Microsoft .NET Framework, version 1.1 or higher
- At least 128 MB of RAM

Installation Procedure

To install the software, double-click the setup file and follow the instructions. Be sure to install at least the following components:

Component	Description
Engine	The Conversion Agent Engine component, required for all lessons in this book.
Studio	The Conversion Agent Studio design and configuration environment, required for all lessons in this book.
Document Processors	Optional components, required for the lessons on parsing PDF and Microsoft Word documents.

Default Installation Folder

By default, Conversion Agent is installed in the following location:

```
c:\Program Files\SAP\ConversionAgent
```

The setup prompts you to change the location if desired.

Tutorials and Workspace Folders

To do the exercises in this book, you need the tutorial files, located in the `tutorials` folder of your main Conversion Agent program folder. By default, the location is:

```
c:\Program Files\SAP\ConversionAgent\tutorials
```

As you perform the exercises, you will import or copy some of the contents of this folder to the Conversion Agent Studio workspace folder. The default location of the workspace is:

```
My Documents\SAP\ConversionAgent\4.0\workspace
```

You should work on the copies in the workspace. We recommend that you do not modify the originals in the `tutorials` folder, in case you need them again.

Exercises and Solutions

The `tutorials` folder contains two subfolders:

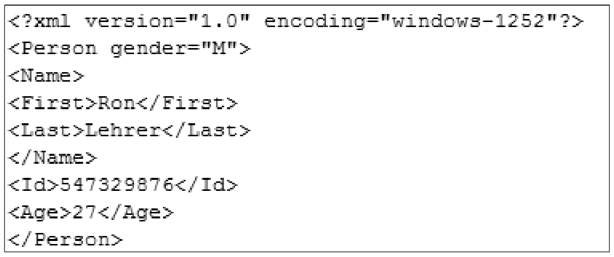
- **Exercises.** This folder contains the files that you need to do the exercises. Throughout this book, we will refer you to files in this folder.

  As you perform the exercises, you will create Conversion Agent projects that have names such as `Tutorial_1` and`Tutorial_2`. The projects will be stored in your Conversion Agent Studio workspace folder.

- **Solutions to Exercises.** This folder contains our proposed solutions to the exercises. The solutions are projects having names such as `TutorialSol_1 and TutorialSol_2`. You can import the projects to your workspace and compare our solutions with yours. Note that there might be more than one correct solution to the exercises.

XML Editor

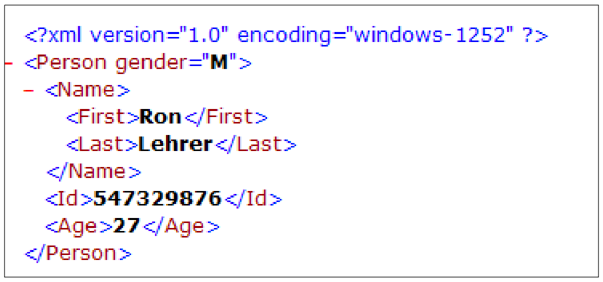
By default, Conversion Agent Studio displays XML files in a plain-text editor.



```
<?xml version="1.0" encoding="windows-1252"?>
<Person gender="M">
<Name>
<First>Ron</First>
<Last>Lehrer</Last>
</Name>
<Id>547329876</Id>
<Age>27</Age>
</Person>
```

Optionally, you can configure the Studio to use Microsoft Internet Explorer as a read-only XML editor. Because Internet Explorer displays XML with color coding and indentation, it can help make a complex XML structure much easier to understand.

The XML illustrations throughout this book use the Internet Explorer display.



```
 <?xml version="1.0" encoding="windows-1252" ?>
- <Person gender="M">
  - <Name>
      <First>Ron</First>
      <Last>Lehrer</Last>
    </Name>
    <Id>547329876</Id>
    <Age>27</Age>
  </Person>
```

To select Internet Explorer as the XML editor:

1. Open Conversion Agent Studio.
2. On the menu, click Window > Preferences.
3. On the left side of the Preferences window, select General> Editors> File Associations.
4. On the upper right, select the `*.xml` file type.

   If it is not displayed, click Add and enter the `*.xml` file type.
5. On the lower right, click Add and browse to `c:\Program Files\Internet Explorer\IEXPLORE.EXE`.
6. Click the Default button to make `IEXPLORE` the default XML editor.
7. Close and re-open Conversion Agent Studio.

CHAPTER 2

Basic Parsing Techniques

This chapter includes the following topics:

- Overview, 7
- Opening Conversion Agent Studio, 8
- Importing the Tutorial_1 Project, 9
- A Brief Look at the Studio Window, 10
- Defining the Structure of a Source Document, 12
- Running the Parser, 18
- Points to Remember, 20
- What's Next?, 21

Overview

To help you start using Conversion Agent quickly, we provide partially configured project containing a simple parser. The project is called `Tutorial_1`.

Working in the Conversion Agent Studio environment, you will edit and complete the configuration. You will then use the parser to convert a few sample text documents to XML.

The main purposes of this exercise are to:

- Demonstrate the parsing-by-example approach
- Start learning how to define and use a parser

Along the way, you will use some of the important Conversion Agent Studio features, such as:

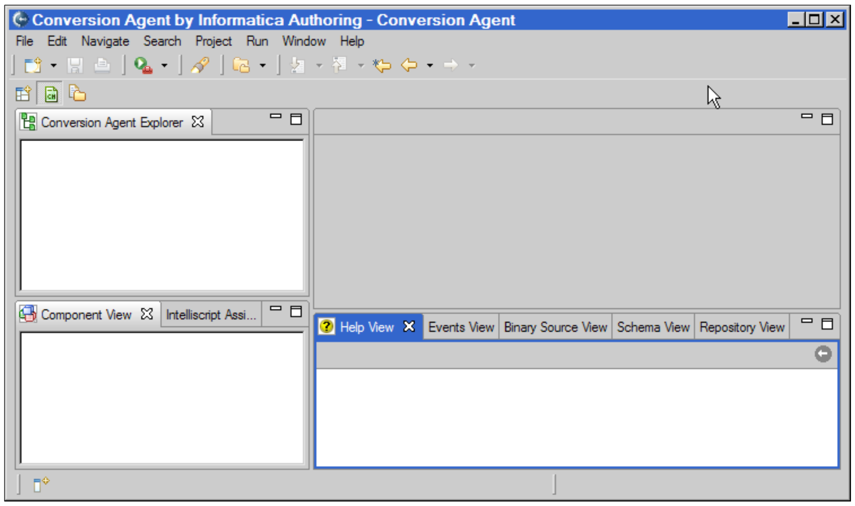
- Importing and opening a project
- Defining the structure of the output XML by using an XSD schema
- Defining the source document structure by using anchors of type `Marker` and `Content`
- Defining a parser based on an example source document
- Using the parser to transform multiple source documents to XML
- Viewing the event log, which displays the operations that the transformation performed

Opening Conversion Agent Studio

To open the Studio:

1. On the Start menu, click Programs > SAP Conversion Agent > Studio.
2. To display the Conversion Agent Studio Authoring perspective, click Window > Open Perspective > Other > Conversion Agent Studio Authoring.

   The Conversion Agent Studio Authoring perspective is a set of windows, menus, and toolbars that you can use to edit projects in Eclipse.



3. Optionally, click Window > Reset Perspective. This is a good idea if you have previously worked in Conversion Agent Studio, and you have moved or resized the windows. The command restores all the windows to their default sizes and locations.
4. To display introductory instructions on how to use the Studio, click Help> Welcome and select the Conversion Agent Studio welcome page.

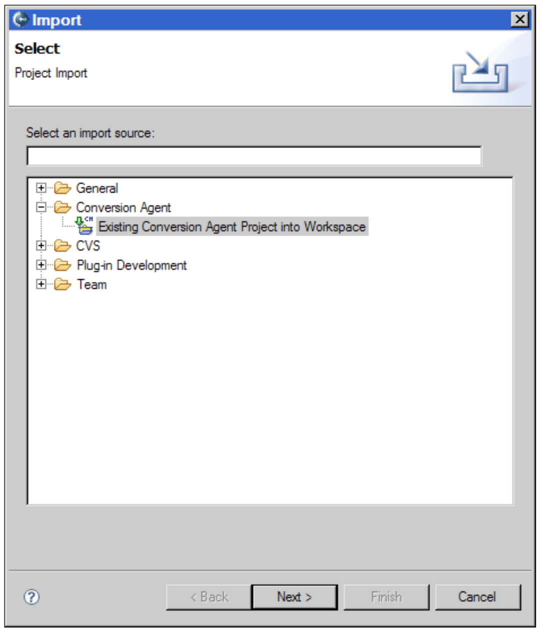
   You can close the welcome page by clicking the X at the right side of the editor tab.

Importing the Tutorial_1 Project

To open the partially configured `Tutorial_1` Project file, you must first import it to the Eclipse workspace.

To import the Tutorial_1 project:

1. On the menu, click File > Import. At the prompt, select the option to import an Existing Conversion Agent Project into Workspace.



2. Click the Next button and browse to the following file:

   `c:\Program Files\SAP\ConversionAgent\tutorials\Exercises\Tutorial_1\Tutorial_1.cmw`

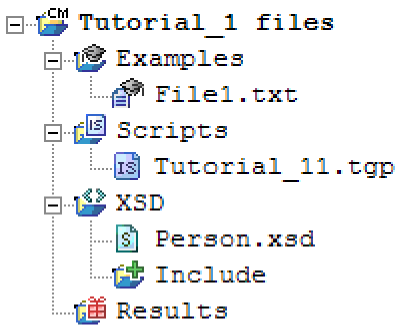
   Note: Substitute your path to the Conversion Agent installation folder.

3. Accept the default options on the remaining wizard pages. Click Finish to complete the import.

   The Eclipse workspace folder now contains the imported `Tutorial_1` folder:

   `My Documents\SAP\ConversionAgent\4.0\workspace\Tutorial_1`

4. In the upper left of the Eclipse window, the Conversion Agent Explorer displays the `Tutorial_1` files that you have imported.



The following is a brief description of the folders and files:

Folder/File	Description
`Examples`	This folder contains an example source document, which is the sample input that you will use to configure the parser.
`Scripts`	A TGP script file, which stores the parser configuration.
`XSD`	An XSD schema file, which defines the XML structure that the parser will create.
`Results`	This folder is temporarily empty. When you configure and run the parser, Conversion Agent will store its output in this folder.

Most of these folders are virtual. They are used to categorize the files in the display, but they do not actually exist on your disk. Only the `Results` folder is a physical directory, which Conversion Agent creates when your transformation generates output.

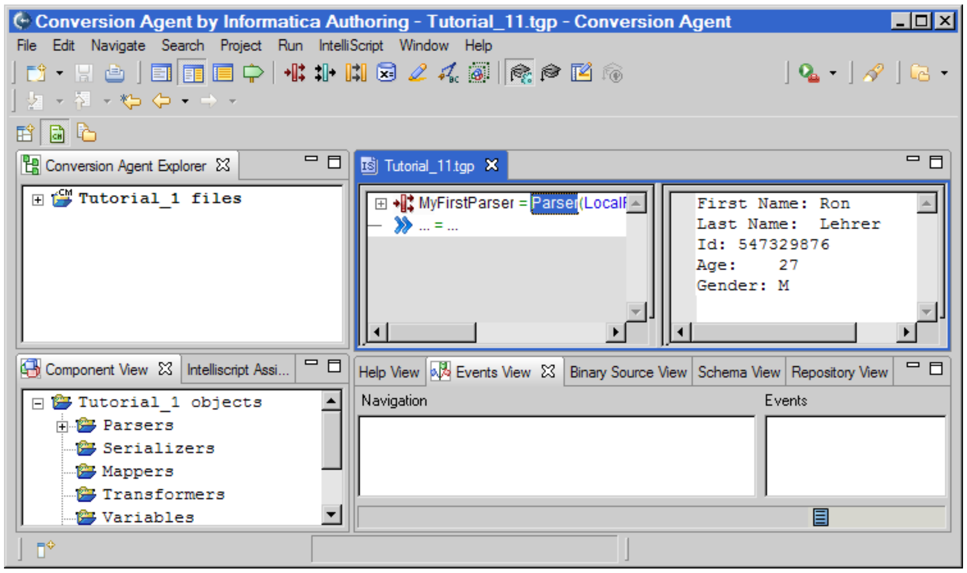
The `Tutorial_1` folder contains additional files that do not display in the Conversion Agent Explorer, for example:

File	Description
`Tutorial_1.cmw`	The main project file, containing the project configuration properties.
`.project`	A file generated by the Eclipse development environment. This is not a Conversion Agent file, but you need it to open the Conversion Agent project in Eclipse.

A Brief Look at the Studio Window

Conversion Agent Studio displays numerous windows. The windows are of two types, called views and editors. A view displays data about a project or lets you perform specific operations. An editor lets you edit the configuration of a project freely.

The following paragraphs describe the views and editors, starting from the upper left and moving counterclockwise around the screen.



```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:     27
Gender: M
```

Upper Left

View	Description
Conversion Agent Explorer view	Displays the projects and files in the Conversion Agent Studio workspace. By right-clicking or double-clicking in this view, you can add existing files to a project, create new files, or open files for editing.

Lower Left

The lower left corner of the Conversion Agent window displays two, stacked views. You can switch between them by clicking the tabs on the bottom.

View	Description
Component view	Displays the main components that are defined in a project, such as parsers, serializers, mappers, transformers, and variables. By right-clicking or double-clicking, you can open a component for editing.
IntelliScript Assistant view	The view helps you configure certain components in the IntelliScript configuration of a transformation. For an explanation, see the description of the IntelliScript editor below.

Lower Right

The lower right displays several views, which you can select by clicking the tabs.

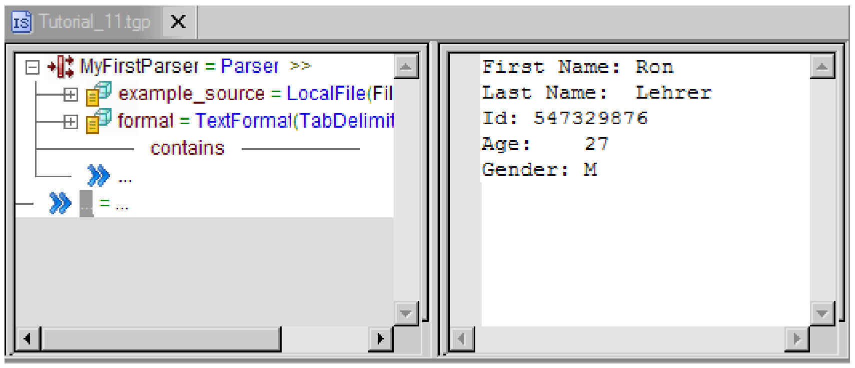
View	Description
Help view	Displays help as you work in an IntelliScript editor. When you select an item in the editor, the help scrolls automatically to an appropriate topic. You can also display the Conversion Agent help from the Conversion Agent Studio Help menu, or from the SAP Conversion Agent folder on the Windows Start menu. These approaches let you access the complete Conversion Agent documentation.
Events view	Displays events that occur as you run a transformation. You can use the events to confirm that a transformation is running correctly or to diagnose problems.
Binary Source view	Displays the binary codes of the example source document. This is useful if you are parsing binary input, or if you need to view special characters such as newlines and tabs.
Schema view	Displays the XSD schemas associated with a project. The schemas define the XML structures that a transformation can process.
Repository view	Displays the services that are deployed for running in Conversion Agent Engine.

Upper Right

Conversion Agent Studio displays editor windows on the upper right. You can open multiple editors, and switch between them by clicking the tabs.

Editor	Description
IntelliScript editor	Used to configure a transformation. This is where you will perform most of the work as you do the exercises in this book. The left pane of the IntelliScript editor is called the IntelliScript pane. This is where you define the transformation. The IntelliScript has a tree structure, which defines the sequence of Conversion Agent components that perform the transformation. The right pane is called the example pane. It displays the example source document of a parser. You can use this pane to configure a parser.

Figure 2-1. IntelliScript Editor



```
MyFirstParser = Parser >>
  example_source = LocalFile(Fil
  format = TextFormat(TabDelimit
  contains
    ...
  = ...
```

```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:    27
Gender: M
```

Defining the Structure of a Source Document

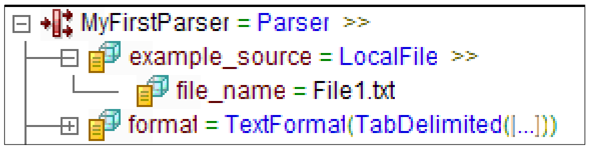
You are ready to start defining the structure of the source document. You will use the parsing-by-example approach to do this.

To define the structure of the source document:

1. In the Conversion Agent Explorer, expand the `Tutorial_1 files` node and double-click the `Tutorial_11.tgp` file. The file opens in an IntelliScript editor.

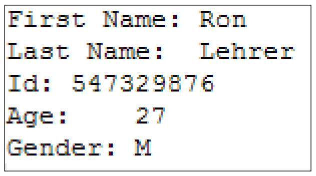
2. The IntelliScript displays a `Parser` component, called `MyFirstParser`. Expand the IntelliScript tree, and examine the properties of the `Parser`. They include the following values, which we have configured for you:

Property	Description
`example_source`	The example source document, which you will use to configure the parser. We have selected a file called `File1.txt` as the example source. The file is stored in the project folder.
`format`	We have specified that the example source has a `TextFormat`, and that it is `TabDelimited`. This means that the text fields are separated from each other by tab characters.



```
MyFirstParser = Parser >>
  example_source = LocalFile >>
    file_name = File1.txt
  format = TextFormat(TabDelimited([...]))
```

The example source, `File1.txt`, should be displayed in the right pane of the editor. If it is not, right-click `MyFirstParser` in the left pane, and click Open Example Source.



```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:    27
Gender: M
```

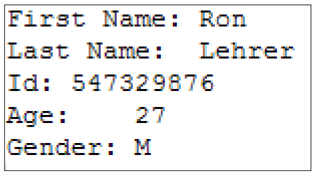
Note: You can toggle the display of the left and right panes. To do this, open the IntelliScript menu and select IntelliScript, Example, or Both. There are also toolbar buttons for these options.

3. Examine the example source more closely. It contains two kinds of information.

The left entries, such as `First Name:`, `Last Name:`, and `Id:` are called `Marker` anchors. They mark the locations of data fields. The right entries, such as `Ron`, `Lehrer`, and `547329876` are the `Content` anchors, which are the values of the data fields

The `Marker` and `Content` anchors are separated by tab characters, which are called tab delimiters.

Figure 2-2. Marker and Content Anchors

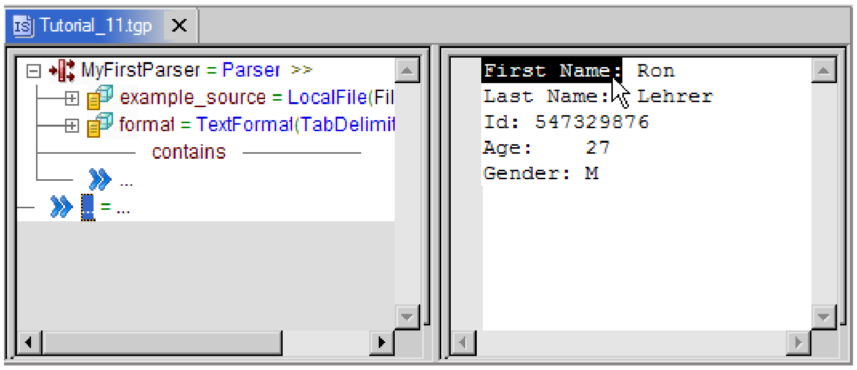


```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:    27
Gender: M
```

To make your job easier, we have already configured the parser with the basic properties, such as the `TabDelimited` format. To complete the configuration of the parser, you will configure the parser to search for each `Marker` anchor and retrieve the data from the `Content` anchor that follows it. The parser will then insert the data in the XML structure.

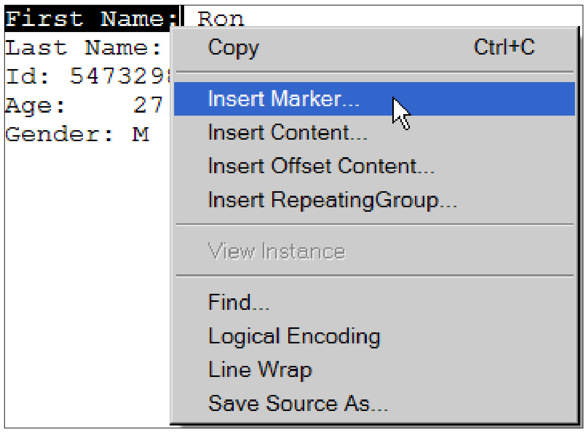
As you continue in this book, you will learn about many other types of anchors and delimiters. Anchors are the places where the parser hooks into a source document and performs an operation. Delimiters are characters that separate the anchors. Conversion Agent lets you define anchors and delimiters in many different ways.

4. You are now ready to start parsing by example. In the example pane, move the mouse over the first `Marker` anchor, which is `First Name:`.


Tutorial_11.tgp
MyFirstParser = Parser >>
example_source = LocalFile(Fil
format = TextFormat(TabDelimit
contains
...
= ...

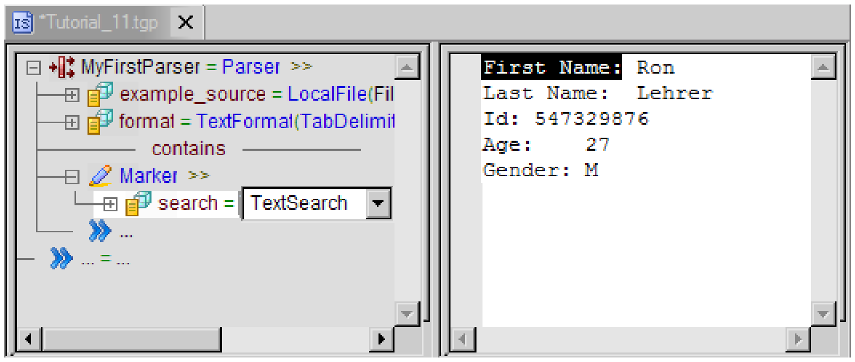
```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:    27
Gender: M
```

5. Click Insert Marker.



```
First Name: Ron
Last Name:
Id: 5473298
Age:    27
Gender: M
```

6. Your last action caused a `Marker` anchor to be inserted in the IntelliScript. You are now prompted for the first property that requires your input, which is `search`. This property lets you specify the type of search, and the default setting is `TextSearch`. Since this is the desired setting, press Enter.

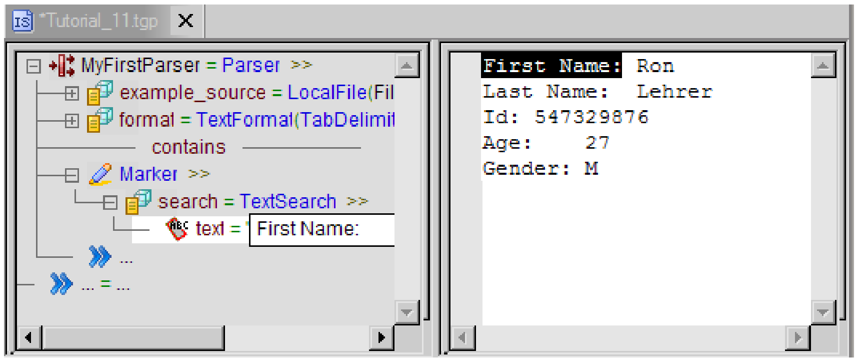


```
*Tutorial_11.tgp
MyFirstParser = Parser >>
  example_source = LocalFile(Fil
  format = TextFormat(TabDelimit
  contains
  Marker >>
    search = TextSearch
  ...
... = ...
```

```
First Name:  Ron
Last Name:   Lehrer
Id:  547329876
Age:     27
Gender:  M
```

7. You are now prompted for the next property that needs your input, which is the `text` of the anchor that the parser searches for. The default is the text that you selected in the example source. Press Enter again.

   In the images displayed here, the selected property is white, and the background is gray. On your screen, the background might be white. You can control this behavior by using the Windows > Preferences command. On the Conversion Agent page of the preferences, select or deselect the option to Highlight Focused Instance.

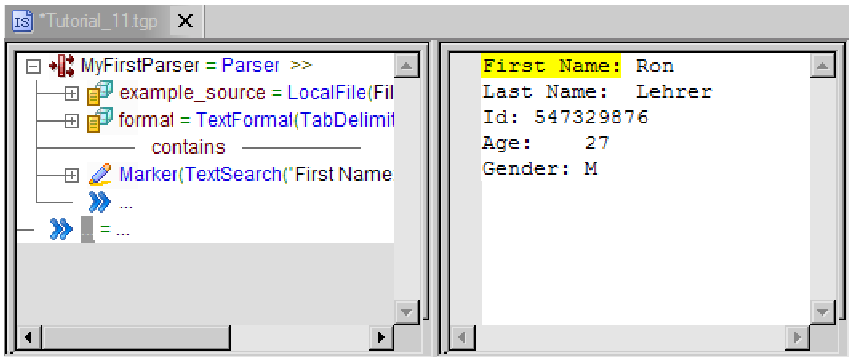


```
*Tutorial_11.tgp
MyFirstParser = Parser >>
  example_source = LocalFile(Fil
  format = TextFormat(TabDelimit
  contains
  Marker >>
    search = TextSearch >>
      text = "First Name:
  ...
... = ...
```

```
First Name:  Ron
Last Name:   Lehrer
Id:  547329876
Age:     27
Gender:  M
```

8. The IntelliScript displays the new `Marker` anchor as part of the `MyFirstParser` definition.

   In the example source, the `Marker` is highlighted in yellow. If the color coding is not immediately displayed, open the IntelliScript menu, and confirm that the option to Learn the Example Automatically is checked.



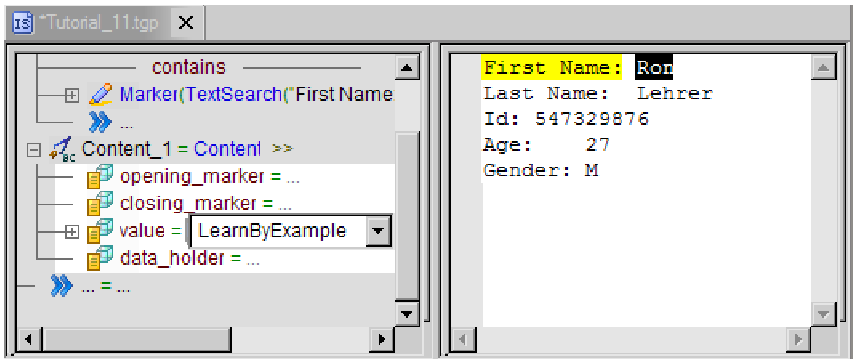
```
*Tutorial_11.tgp
MyFirstParser = Parser >>
  example_source = LocalFile(Fil
  format = TextFormat(TabDelimit
  contains
  Marker(TextSearch("First Name
  ...
  = ...
```

```
First Name:  Ron
Last Name:   Lehrer
Id:  547329876
Age:     27
Gender:  M
```

9. Now, you are ready to create the first `Content` anchor. In the example source, select the word `Ron`. Right-click the selected text. On the pop-up menu, click Insert Content.

10. This inserts a `Content` anchor in the IntelliScript, and prompts you for the first property that requires your input, which is `value`. This property lets you specify how the parsing will be performed.

The default is `LearnByExample`, which means that the parser finds the anchor based on the delimiters surrounding it in the example source. This is the desired behavior. To accept the default, click Enter.



```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:     27
Gender: M
```

11. You can accept the defaults for the next few properties, such as `example`, `opening_marker`, and `closing-marker`, or you can skip them by clicking elsewhere with the mouse.

12. Your next task is to specify where the `Content` anchor stores the data that it extracts from the source document.

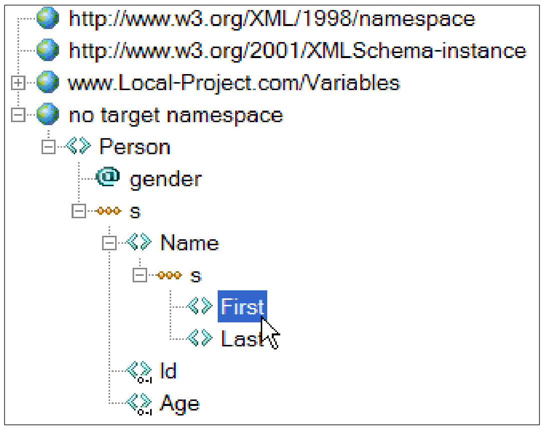
The output location is called a data holder. This is a generic term which means “an XML element, an XML attribute, or a variable”. In this exercise, you will use data holders that are elements or attributes. In a later exercise, you will use variables.

To define the data holder, select the `data_holder` property, and double-click or press Enter. This opens a Schema view, which displays the XML elements and attributes that are defined in the XSD schema of the project, `Person.xsd`.

Expand the `no target namespace` node, and select the `First` element. This means that the `Content` anchor stores its output in an XML element called `First`, like this:

```
<First>Ron</First>
```

Figure 2-3. Selecting a Data Holder

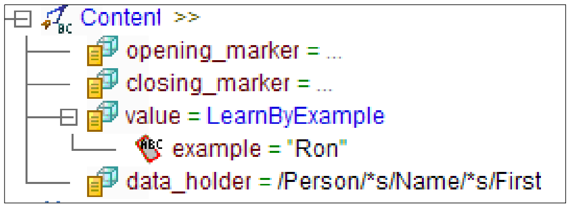

http://www.w3.org/XML/1998/namespace
http://www.w3.org/2001/XMLSchema-instance
www.Local-Project.com/Variables
no target namespace
Person
gender
s
Name
s
First
Last
Id
Age

More precisely, the `First` element is nested inside `Name`, which is nested inside `Person`. The actual output structure generated by the parser will be:

```
<Person>
  <Name>
    <First>Ron</First>
  </Name>
</Person>
```

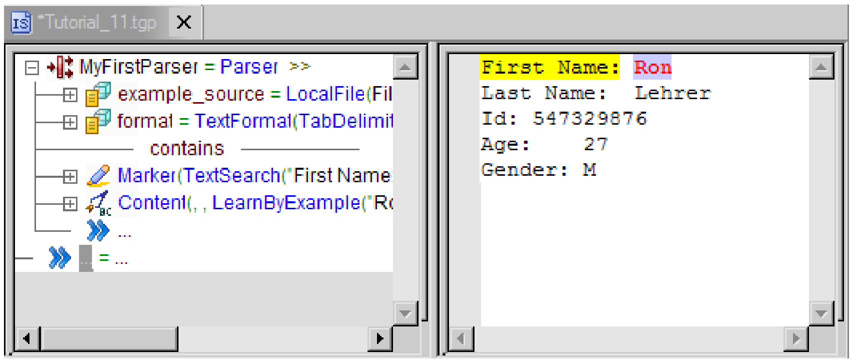
When you click the OK button, the `data_holder` property is assigned the value `/Person/*s/Name/*s/First`. In XML terminology, this is called an XPath expression. It is a representation of the XML structure that we have outlined above.

Actually, the standard XPath expression is `/Person/Name/First`. The `*s` symbols are Conversion Agent extensions of the XPath syntax, which you can ignore.



```
Content >>
  opening_marker = ...
  closing_marker = ...
  value = LearnByExample
    example = "Ron"
  data_holder = /Person/*s/Name/*s/First
```

Conversion Agent Studio highlights the `Content` anchor in the example source. The `Marker` and `Content` anchors are in different colors, to help you distinguish them.



```
MyFirstParser = Parser >>
  example_source = LocalFile(Fil
  format = TextFormat(TabDelimit
  contains
  Marker(TextSearch("First Name
  Content(, , LearnByExample("Ro
  ...
  = ...
```

```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:    27
Gender: M
```

13. Click the File > Save command on the menu, or click the Save button on the toolbar. This ensures that your editing is saved in the project folder.

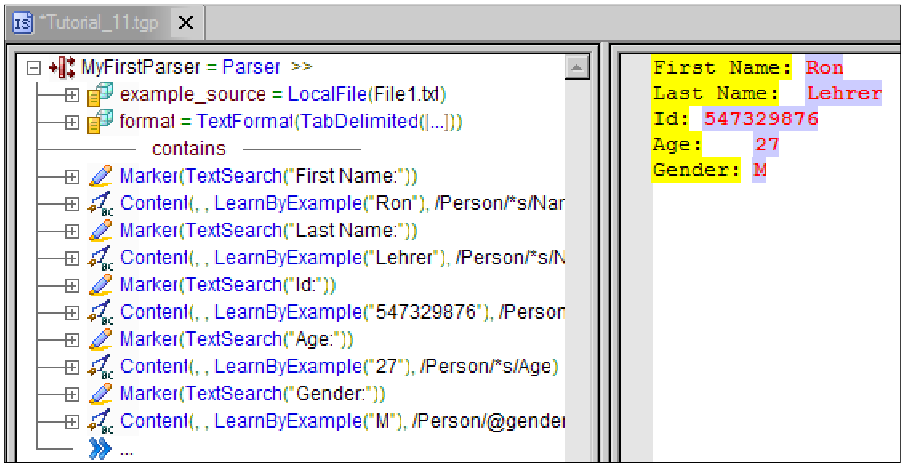
14. So far, you have told the parser to search for the `First Name:` anchor, retrieve the `Ron` anchor that follows it, and store the content in the `First` element of the XSD schema.

Proceed through the source document, defining the other `Marker` and `Content` anchors in the same way. The following table lists the anchors that you need to define.

Anchor	Anchor Type	Data Holder
`Last Name:`	`Marker`	
`Lehrer`	`Content`	`/Person/*s/Name/*s/Last`
`Id:`	`Marker`	
`547329876`	`Content`	`/Person/*s/Id`
`Age:`	`Marker`	
`27`	`Content`	`/Person/*s/Age`
`Gender:`	`Marker`	
`M`	`Content`	`/Person/@gender`

Be sure to define the anchors in the above sequence, which is their sequence in the source document. Using the correct sequence is important in establishing the correct relationship between each `Marker` and the corresponding `Content`. If you make a mistake in the sequence, Conversion Agent might fail to find the text.

When you have finished, the Conversion Agent Studio window should look like the following illustration. You can expand or collapse the tree to view the illustrated lines.



```
MyFirstParser = Parser >>
  example_source = LocalFile(File1.txt)
  format = TextFormat(TabDelimited([...]))
  contains
  Marker(TextSearch("First Name:"))
  Content(, , LearnByExample("Ron"), /Person/*s/Nar
  Marker(TextSearch("Last Name:"))
  Content(, , LearnByExample("Lehrer"), /Person/*s/N
  Marker(TextSearch("Id:"))
  Content(, , LearnByExample("547329876"), /Persor
  Marker(TextSearch("Age:"))
  Content(, , LearnByExample("27"), /Person/*s/Age)
  Marker(TextSearch("Gender:"))
  Content(, , LearnByExample("M"), /Person/@gendei
  ...
```

```
First Name: Ron
Last Name:  Lehrer
Id: 547329876
Age:      27
Gender: M
```

15. Save the project again.

Correcting Errors in the Parser Configuration

As you define anchors, you might occasionally make a mistake such as selecting the wrong text, or setting the wrong property values for an anchor.

If you make a mistake, you can correct it in several ways:

- On the menu, you can click Edit > Undo.
- In the IntelliScript pane, you can select a component that you have added to the configuration and press the Delete key.
- In the IntelliScript pane, you can right-click a component and click Delete.

As you gain more experience working in the IntelliScript pane, you can use the following additional techniques:

- If you create an anchor in the wrong sequence, you can drag it to the correct location in the IntelliScript.
- If you forget to define an anchor, right-click the anchor following the omitted anchor location and click Insert.
- You can copy and paste components such as anchors in the IntelliScript.
- You can edit the property values in the IntelliScript.

Techniques for Defining Anchors

In the above steps, you inserted markers by using the Insert Marker and Insert Content commands.

There are several alternative ways to define anchors:

- You can define a `Content` anchor by dragging text from the example source to a data holder in the Schema view. This inserts the anchor in the IntelliScript, where the `data_holder` property is already assigned.
- You can define anchors by typing in the IntelliScript pane, without using the example source.
- You can edit the properties of `Marker` and `Content` anchors in the IntelliScript Assistant view.

We encourage you to experiment with these features. For more information editing the IntelliScript, see *Using Conversion Agent Studio in Eclipse*.

Tab-Delimited Format

Do you remember that `MyFirstParser` is defined with a `TabDelimited` format? The delimiters define how the parser interprets the example source. In this case, the parser understands that the `Marker` and `Content` anchors are separated by tab characters.

In the instructions, we suggested that you select the `Marker` anchors including the colon character, for example:

```
First Name:
```

Because the tab-delimited format is selected, the colon isn't actually important in this parser. If you had selected `First Name` without the colon, the parser would still find the `Marker` and the tab following the `Marker`, and it would read the `Content` correctly. It would ignore other characters, such as the colon.

The tab-delimited format also explains why you can select a short `Content` anchor such as `Ron`, and not be concerned about the field size. In another source document, a person might have a long first name such as `Rumpelstiltskin`. By default, a tab-delimited parser reads the entire string after the tab, up to the line break. This is the case, unless the line contains another anchor or tab character.

Running the Parser

You are now ready to test the parser that you have configured.

To test the parser:

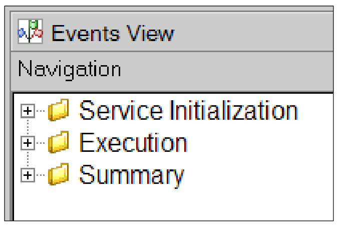
1. In the IntelliScript, right-click the `Parser` component, which is named `MyFirstParser`. Click the option to Set as Startup Component.

   This means that when you run the project, Conversion Agent will activate `MyFirstParser`.

2. On the menu, click Run > Run MyFirstParser.

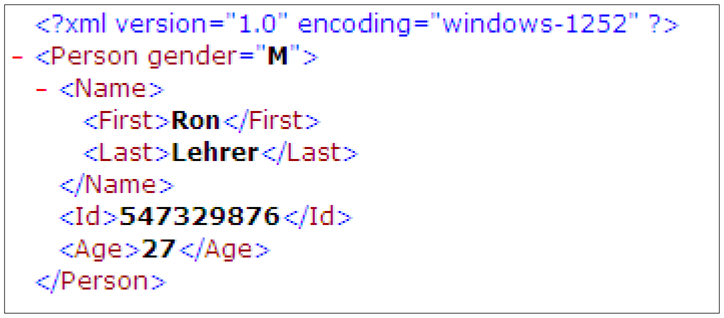
3. After a moment, the Studio displays the Events view. Among other information, the events list all the `Marker` and `Content` anchors that the parser found in the example source.

   You can use the Events view to examine any errors encountered during execution. Assuming that you followed the instructions carefully, the execution should be error-free.



4. Now you can examine the output of the parser process. In the Conversion Agent Explorer, expand the `Tutorial_1 files/Results` node, and double-click the file `output.xml`.

The Studio displays the XML file in an Internet Explorer window. Examine the output carefully, and confirm that the results are correct. If the results are incorrect, examine the parser configuration that you created, correct any mistakes, and try again.



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Person gender="M">
  - <Name>
      <First>Ron</First>
      <Last>Lehrer</Last>
    </Name>
    <Id>547329876</Id>
    <Age>27</Age>
  </Person>
```

If you are using Windows XP SP2 or higher, Internet Explorer might display a warning about active content in the XML results file. You can safely ignore the warning.

Do not worry if the XML processing instruction (`<?xml version="1.0"...?>`) is missing or different from the one displayed above. This is due to the settings in the project properties. You can ignore this issue for now.

Running the Parser on Additional Source Documents

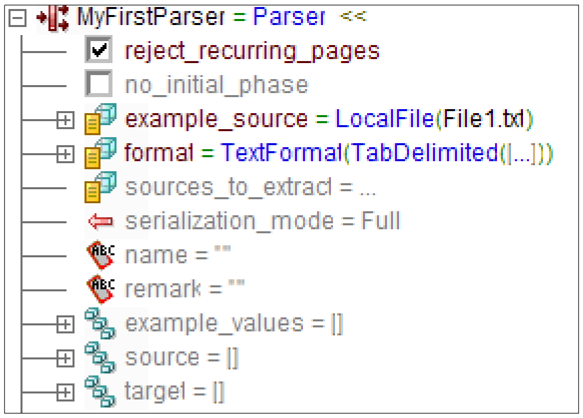
To further test the parser, you can run it on additional source documents, other than the example source.

To test the parser on additional source documents:

1. At the right of the `Parser` component, click the >> symbol. This displays the advanced properties of the component.

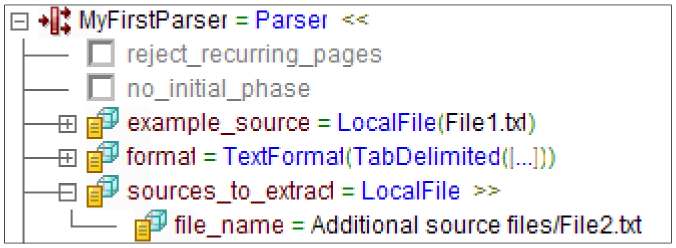
   The >> symbol changes to <<. You can close the advanced properties by clicking the << symbol.

   Figure 2-4. Displaying Advanced Properties

   ```
   MyFirstParser = Parser <<
     [✓] reject_recurring_pages
     [ ] no_initial_phase
     example_source = LocalFile(File1.txt)
     format = TextFormat(TabDelimited([...]))
     sources_to_extract = ...
     serialization_mode = Full
     name = ""
     remark = ""
     example_values = []
     source = []
     target = []
   ```

2. Select the `sources_to_extract` property, and double-click or press Enter. From the drop-down list, select `LocalFile`.

   ```
   MyFirstParser = Parser <<
     [ ] reject_recurring_pages
     [ ] no_initial_phase
     example_source = LocalFile(File1.txt)
     format = TextFormat(TabDelimited([...]))
     sources_to_extract = LocalFile >>
       file_name = Additional source files/File2.txt
   ```

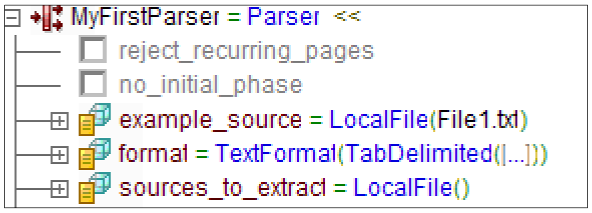
3. Expand the `LocalFile` node of the IntelliScript, and assign the `file_name` property. Conversion Agent Studio displays an Open dialog, where you can browse to the file.

   You can use one of the test files that we have supplied, which are in the folder

   ```
   tutorials\Exercises\Tutorial_1\Additional source files
   ```

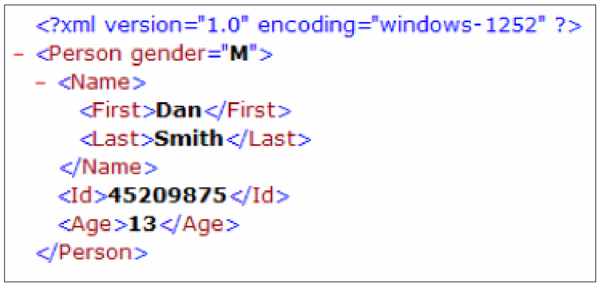
   For example, select `File2.txt`, which has the following content:

   ```
   First Name:    Dan
   Last Name:     Smith
   Id:            45209875
   Age:           13
   Gender:        M
   ```



```
MyFirstParser = Parser <<
  reject_recurring_pages
  no_initial_phase
  example_source = LocalFile(File1.txt)
  format = TextFormat(TabDelimited([...]))
  sources_to_extract = LocalFile()
```

4. Run the parser again. You should get the same XML structure as above, containing the data that the parser found in the test file.



```
<?xml version="1.0" encoding="windows-1252" ?>
- <Person gender="M">
  - <Name>
      <First>Dan</First>
      <Last>Smith</Last>
    </Name>
    <Id>45209875</Id>
    <Age>13</Age>
  </Person>
```

[

Points to Remember

A Conversion Agent project contains the parser configuration and the XSD schema. It usually also contains the example source document, which the parser uses to learn the document structure, and other files such as the parsing output.

The Conversion Agent Explorer displays the projects that exist in your Studio workspace. To copy an existing project into the workspace, use the File > Import command.

To open a transformation for editing in the IntelliScript editor, double-click its TGP script file in the Conversion Agent Explorer. In the IntelliScript, you can add components such as anchors. The anchors define locations in the source document, which the parser seeks and processes. `Marker` anchors label the data fields, and `Content` anchors extract the field values.

To define a `Marker` or `Content` anchor, select the source text, right-click, and choose the anchor type. This inserts the anchor in the IntelliScript, where you can set its properties. For example, you can set the data holder—an XML element or attribute—where a `Content` anchor stores its output.

Use the color coding to review the anchor definitions.

The delimiters define the relation between the anchors. A tab-delimited format means that the anchors are separated by tabs.

To test a parser, set it as the startup component. Then use the Run command on the Conversion Agent Studio menu. To view the results file, double-click its name in the Conversion Agent Explorer.

What's Next?

Congratulations! You have configured and run your first Conversion Agent parser.

Of course, the source documents that you parsed had a very simple structure. The documents contained a few `Marker` anchors, each of which was followed by a tab character and by `Content`.

Moreover, all the documents had exactly the same `Marker` anchors in the same sequence. This made the parsing easy because you did not need to consider the possible variations among the source documents.

In real-life uses of parsers, very few source documents have such a simple, rigid structure. In the following chapters, you will learn how to parse complex and flexible document structures using a variety of parsing techniques. All the techniques are based, however, on the simple steps that you learned in this chapter.

CHAPTER 3

Defining an HL7 Parser

This chapter includes the following topics:

- ♦ Overview, 23
- ♦ Creating a Project, 25
- ♦ Defining the Anchors, 27
- ♦ Testing the Parser, 30
- ♦ Points to Remember, 32

Overview

In this chapter, you will parse an HL7 message. HL7 is a standard messaging format used in medical information systems. The structure is characterized by a hierarchy of delimiters and by repeating elements. After you learn the techniques for processing these features, you will be able to parse a large variety of documents that are used in real applications.

In this lesson, you will configure the parser yourself. We will provide the example source document and a schema for the output XML vocabulary. You will learn techniques such as:

- ♦ Creating a project
- ♦ Creating a parser
- ♦ Parsing a document selectively, that is, retrieving selected data and ignoring the rest
- ♦ Defining a repeating group
- ♦ Using delimiters to define the source document structure
- ♦ Testing and debugging a parser

Requirements Analysis

Before you start the exercise, we will analyze the input and output requirements of the project. As you design the parser, you will use this information to guide the configuration.

HL7 Background

HL7 is a messaging standard for the health services industry. It is used worldwide in hospital and medical information systems.

For more information about HL7, see the Health Level 7 web site, http://www.hl7.org.

Input HL7 Message Structure

The following lines illustrate a typical HL7 message, which you will use as the source document for parsing.

```
MSH|^~\&|LAB||CDB|||ORU^R01|K172|P
PID|||PATID1234^5^M11||Jones^William||19610613|M
OBR||||80004^Electrolytes
OBX|1|ST|84295^Na||150|mmol/l|136-148|Above high normal|||Final results
OBX|2|ST|84132^K+||4.5|mmol/l|3.5-5|Normal|||Final results
OBX|3|ST|82435^Cl||102|mmol/l|94-105|Normal|||Final results
OBX|4|ST|82374^CO2||27|mmol/l|24-31|Normal|||Final results
```

The message is composed of segments, which are separated by carriage returns. Each segment has a three-character label, such as `MSH` (message header) or `PID` (patient identification). Each segment contains a predefined hierarchy of fields and sub-fields, which are delimited by the characters immediately following the `MSH` designator (`|^~\&`).

For example, the patient's name (`Jones^William`) follows the `PID` label by five `|` delimiters. The last and first names (`Jones` and `William`) are separated by a `^` delimiter.

The message type is specified by a field in the `MSH` segment. In the above example, the message type is `ORU`, subtype `R01`, which means Unsolicited Transmission of an Observation Message. The `OBR` segment specifies the type of observation, and the `OBX` segments list the observation results.

In this chapter, you will configure a parser that processes `ORU` messages such as the above example. Some key issues in the parser definition are how to define the delimiters and how to process the repeating `OBX` group.

Output XML Structure

The purpose of this exercise is to create a parser, which will convert the above HL7 message to the following XML output:

```
<Message type="..." id="...">
  <Patient id="..." gender="...">
    <f_name>...</f_name>
    <l_name>...</l_name>
    <birth_date>...</birth_date>
  </Patient>
  <Test_Type test_id="...">...</Test_Type>
  <Result num="...">
    <type>...</type>
    <value>...</value>
    <range>...</range>
    <comment>...</comment>
    <status>...</status>
  </Result>
  <Result num="...">
    <type>...</type>
    <value>...</value>
    <range>...</range>
    <comment>...</comment>
    <status>...</status>
  </Result>
  <!-- Additional Result elements as needed -->
</Message>
```

The XML has elements that can store much—but not all—of the data in the sample HL7 message. That is acceptable. In this exercise, you will build a parser that processes the data in the source document selectively, retrieving the information that it needs and ignoring the rest. The XML structure contains the elements that are required for retrieval.

Notice the repeating `Result` element. This element will store data from the repeating `OBX` segment of the HL7 message.

Creating a Project

First, you will create a project where Conversion Agent Studio can store your work.

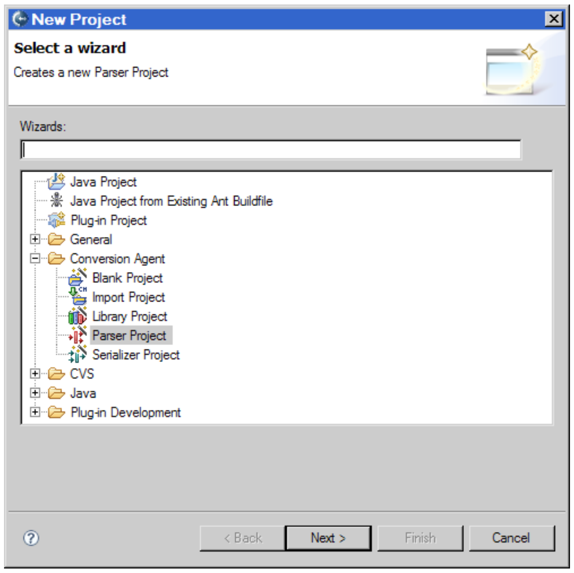
To create a project:

1. On the Conversion Agent Studio menu, click File > New > Project.

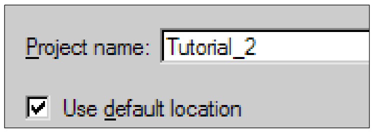
   This opens a wizard where you can select the type of project.

   You don't have to close or remove the `Tutorial_1` project or any other projects that you already have. The Eclipse workspace can contain any number of projects.

2. Under the Conversion Agent node, select a Parser Project.



3. On the next page of the wizard, enter a project name, such as `Tutorial_2`. the software creates a folder and a `*.cmw` configuration file having this name.



Project name: Tutorial_2

☑ Use default location

4. On the following wizard page, enter a name for the `Parser` component. Since this parser will parse HL7 message of type `ORU`, call it `HL7_ORU_Parser`.

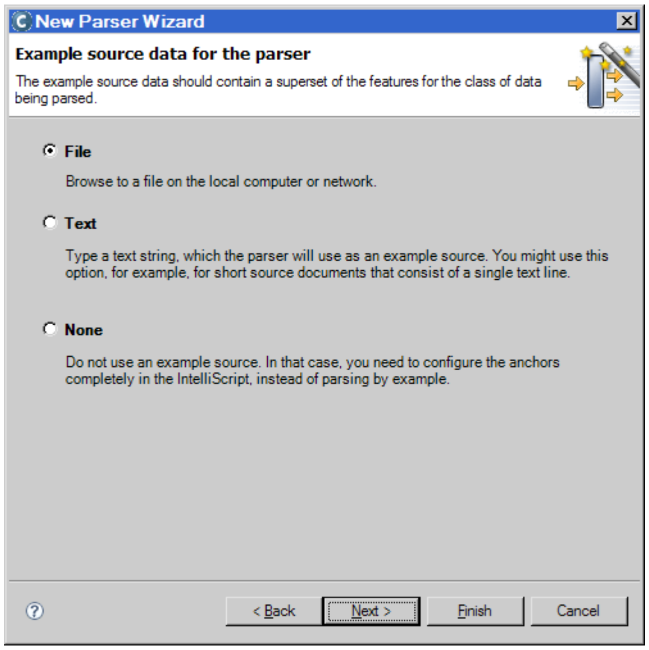
5. On the next page, enter a name for the TGP script file that the wizard creates. A convenient name is `Script_Tutorial_2`.

6. On the next page, you can select an XSD schema, which defines the XML structure where the parser will store its output. We have provided a schema for you. In the Conversion Agent installation folder, the location is:

```
tutorials\Exercises\Files_For_Tutorial_2\HL7_tutorial.xsd
```

   Browse to this file and click Open. The Studio copies the schema to the project folder.

7. On the next page, specify the example source type. In this exercise, it is a file, which we have provided. Select the File option.



New Parser Wizard

Example source data for the parser

The example source data should contain a superset of the features for the class of data being parsed.

[✓ **File**]

Browse to a file on the local computer or network.

Text

Type a text string, which the parser will use as an example source. You might use this option, for example, for short source documents that consist of a single text line.

None

Do not use an example source. In that case, you need to configure the anchors completely in the IntelliScript, instead of parsing by example.

8. The next page prompts you to browse to the example source file. The location is:

   ```
   tutorials\Exercises\Files_For_Tutorial_2\hl7-obs.txt
   ```

   The Studio copies the file to the project folder.

9. On the next page, select the encoding of the source document. In this exercise, the encoding is ASCII, which is the default.

10. You can skip the document preprocessors page. You do not need a document preprocessor in this project.

11. Select the format of the source document. In this project, the format is HL7.

    This causes the parser to assume that the message fields are separated by the HL7 delimiter hierarchy:

    ```
    newline
    |
    Other symbols such as ^ and tab
    ```

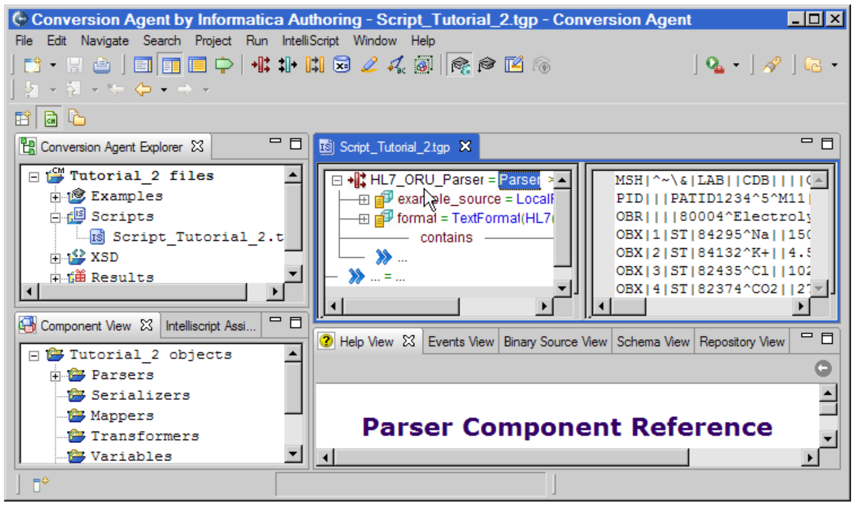
    For example, the parser might learn that a particular `Content` anchor starts at a count of three | delimiters after a `Marker`, and ends at the fourth | delimiter.

12. Review the summary page and click Finish.

13. The software creates the new project. It displays the project in the Conversion Agent Explorer. It opens the script that you have created, `Script_Tutorial_2.tgp`, in an IntelliScript editor.

    In the IntelliScript, notice that the `example_source` and `format` properties have been assigned according to your specifications.

The example source should be displayed. If it is not, right-click the `Parser` component and click Open Example Source.



```
MSH|^~\&|LAB||CDB|||(
PID|||PATID1234^5^M11|
OBR||||80004^Electroly
OBX|1|ST|84295^Na||15(
OBX|2|ST|84132^K+||4.5
OBX|3|ST|82435^Cl||102
OBX|4|ST|82374^CO2||27
```

Using XSD Schemas in Transformations

Transformations require XSD schemas to define the structure of XML documents. Every parser, serializer, or mapper project requires at least one schema.

When you perform the tutorial exercises in this book, we provide the schemas that you need. For your own applications, you might already have the schemas, or you can create new ones.

Learning XSD

For an excellent introduction to the XSD schema syntax, see the tutorial on the W3Schools web site, http://www.w3schools.com. For definitive reference information, see the XML Schema standard at http://www.w3.org.

Editing Schemas

You can use any XSD editor to create and edit the schemas that you use with Conversion Agent. For more information about schemas, see the *Conversion Agent Studio User Guide*.

Defining the Anchors

Now it is time to define the transformation anchors.

To define the anchors:

1. You need to define `Marker` anchors that identify the locations of fields in the source document, and `Content` anchors that identify the field values.

   You will start with the non-repeating portions of the document, which are the first three lines. The most convenient `Marker` anchors are the segment labels, `MSH`, `PID`, and `OBR`. These labels identify portions of the document that have a well-defined structure.

   Within each segment, you need to identify the data fields to retrieve. These are the `Content` anchors. There are several `Content` anchors for each `Marker`.

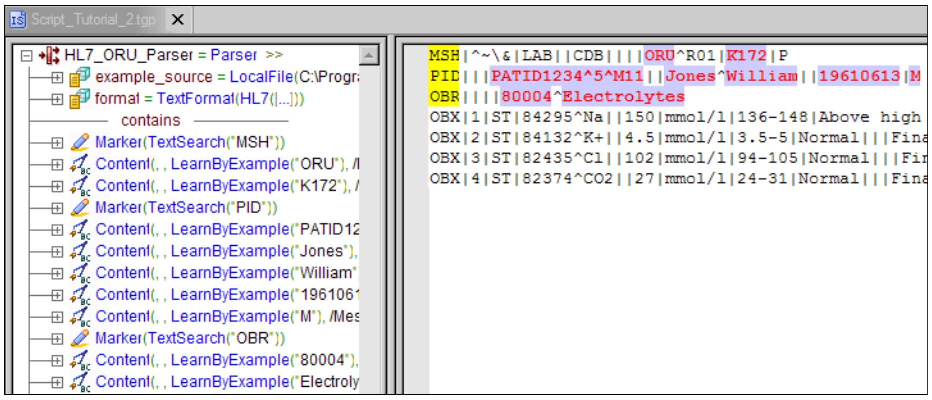
   In addition, you need to define the data holders for each `Content` anchor. The data holders are elements or attributes in the XML output, which is illustrated above.

If you study the HL7 message and the XML structure that should be produced, you can probably figure out which message fields need to be defined as anchors, and you can map the fields to their corresponding data holders. To save time, we have done this job for you. We present the results in the following table:

Anchor	Anchor Type	Data Holder
`MSH`	`Marker`	
`ORU`	`Content`	`/Message/@type`
`K172`	`Content`	`/Message/@id`
`PID`	`Marker`	
`PATID1234^5^M11`	`Content`	`/Message/*s/Patient/@id`
`Jones`	`Content`	`/Message/*s/Patient/*s/l_name`
`William`	`Content`	`/Message/*s/Patient/*s/f_name`
`19610613`	`Content`	`/Message/*s/Patient/*s/birth_date`
`M`	`Content`	`/Message/*s/Patient/@gender`
`OBR`	`Marker`	
`80004`	`Content`	`/Message/*s/Test_Type/@test_id`
`Electrolytes`	`Content`	`/Message/*s/Test_Type`

Note the @ symbol in some of the XPath expressions, such as `/Message/@type`. The symbol means that `type` is an attribute, not an element.

Create the anchors in the parser definition, as you did in the preceding chapter. When you finish, the IntelliScript editor should appear as in the following illustration.



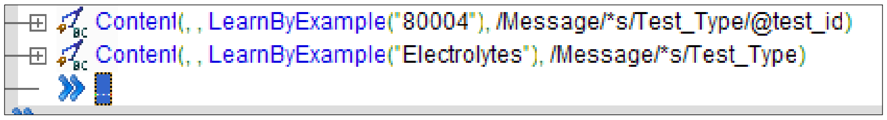
```
Script_Tutorial_2.tgp
HL7_ORU_Parser = Parser >>
  example_source = LocalFile(C:\Progr
  format = TextFormat(HL7(|...|))
  contains
  Marker(TextSearch("MSH"))
  Content(, , LearnByExample("ORU"), /I
  Content(, , LearnByExample("K172"), /
  Marker(TextSearch("PID"))
  Content(, , LearnByExample("PATID12
  Content(, , LearnByExample("Jones"),
  Content(, , LearnByExample("William"
  Content(, , LearnByExample("196106
  Content(, , LearnByExample("M"), /Mes
  Marker(TextSearch("OBR"))
  Content(, , LearnByExample("80004"),
  Content(, , LearnByExample("Electroly
```

```
MSH|^~\&|LAB||CDB|||ORU^R01|K172|P
PID|||PATID1234^5^M11||Jones^William||19610613|M
OBR||||80004^Electrolytes
OBX|1|ST|84295^Na||150|mmol/l|136-148|Above high
OBX|2|ST|84132^K+||4.5|mmol/l|3.5-5|Normal|||Fina
OBX|3|ST|82435^Cl||102|mmol/l|94-105|Normal|||Fir
OBX|4|ST|82374^CO2||27|mmol/l|24-31|Normal|||Fina
```

2. Now, you need to teach the parser how to process the `OBX` lines of the source document. There are several `OBX` lines, each having the same format. The parser should create output for each `OBX` line that it finds.

   To do this, you will define an anchor called a `RepeatingGroup`. The anchor tells Conversion Agent to search for a repeated segment. Inside the `RepeatingGroup`, you will nest several `Content` anchors, which tell the parser how to parse each iteration of the segment.

   In the IntelliScript pane, find `Electrolytes`, the last anchor that you defined. Immediately below the anchor, there is an empty node containing three dots (...). That is where you should define the new anchor.

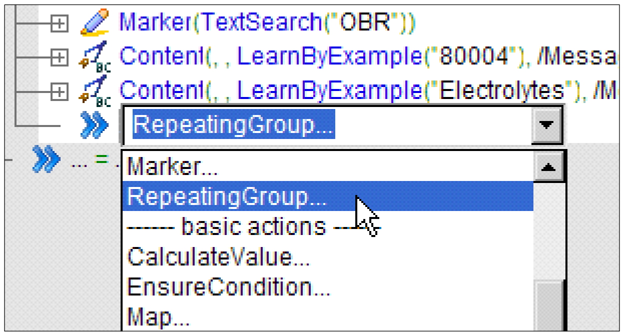


```
Content(, , LearnByExample("80004"), /Message/*s/Test_Type/@test_id)
Content(, , LearnByExample("Electrolytes"), /Message/*s/Test_Type)
```

   Select the three dots and press the Enter key. This opens a drop-down list, which displays the names of the available anchors.

In the list, select the `RepeatingGroup` anchor. Alternatively, you can type the text `RepeatingGroup` in the box. After you type the first few letters, Conversion Agent Studio automatically completes the text.

Press the Enter key again to accept the new entry.



```
Marker(TextSearch("OBR"))
Content(, , LearnByExample("80004"), /Messa
Content(, , LearnByExample("Electrolytes"), /M
RepeatingGroup...
... =
```

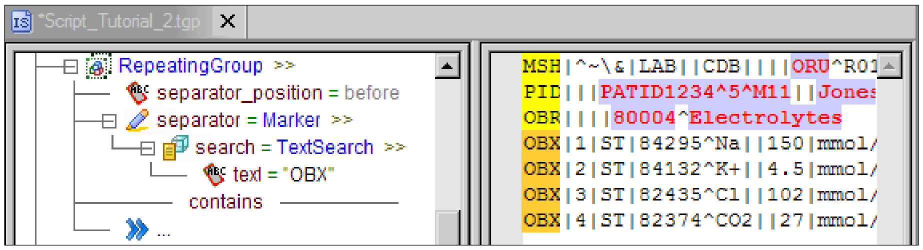
3. Now, you must configure the `RepeatingGroup` so it can identify the repeating segments. You will do this by assigning the `separator` property. You will specify that the segments are separated from each other by a `Marker`, which is the text `OBX`.

   In the IntelliScript pane, expand the `RepeatingGroup`. Find the line that defines the `separator` property.

   By default, the `separator` value is empty, which is symbolized by a `...` symbol. Select the `...` symbol, press Enter, and change the value to `Marker`. Press Enter again to accept the new value. The `Marker` value means that the repeating elements are separated by a `Marker` anchor.

   Expand the `Marker` property, and find its `text` property. Select the value, which is empty by default, and press Enter. Type the value `OBX`, and press Enter again. This means that the separator is the `Marker` anchor `OBX`.

   In the example pane, Conversion Agent Studio highlights all the OBX anchors, signifying that it found them correctly.



```
RepeatingGroup >>
    separator_position = before
    separator = Marker >>
        search = TextSearch >>
            text = "OBX"
        contains
    ...
```

```
MSH|^~\&|LAB||CDB|||ORU^R01
PID|||PATID1234^5^M11||Jones
OBR||||80004^Electrolytes
OBX|1|ST|84295^Na||150|mmol/
OBX|2|ST|84132^K+||4.5|mmol/
OBX|3|ST|82435^Cl||102|mmol/
OBX|4|ST|82374^CO2||27|mmol/
```

4. Now, you will insert the `Content` anchors, which parse an individual `OBX` line.

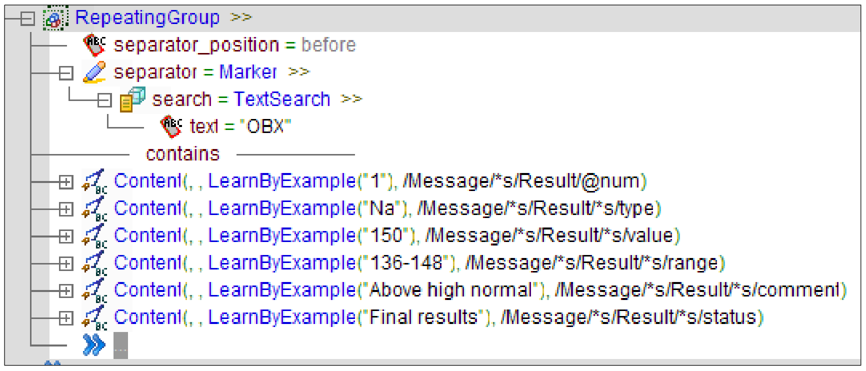
   To do this, keep the `RepeatingGroup` selected. You must nest the `Content` anchors within the `RepeatingGroup`.

   Define the anchors only on the first `OBX` line. Because the anchors are nested in a `RepeatingGroup`, the parser looks for the same anchors in additional `OBX` lines.

   Define the `Content` anchors as follows:

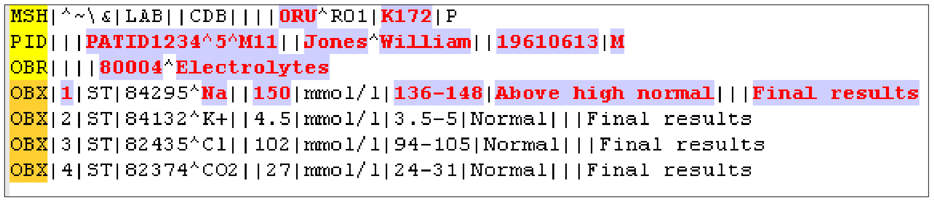
Anchor	Anchor Type	Data Holder
`1`	`Content`	`/Message/*s/Result/@num`
`Na`	`Content`	`/Message/*s/Result/*s/type`
`150`	`Content`	`/Message/*s/Result/*s/value`
`136-148`	`Content`	`/Message/*s/Result/*s/range`
`Above high normal`	`Content`	`/Message/*s/Result/*s/comment`
`Final results`	`Content`	`/Message/*s/Result/*s/status`

Figure 3-1. Nesting Anchors in a RepeatingGroup



```
RepeatingGroup >>
    separator_position = before
    separator = Marker >>
        search = TextSearch >>
            text = "OBX"
    contains
    Content(, , LearnByExample("1"), /Message/*s/Result/@num)
    Content(, , LearnByExample("Na"), /Message/*s/Result/*s/type)
    Content(, , LearnByExample("150"), /Message/*s/Result/*s/value)
    Content(, , LearnByExample("136-148"), /Message/*s/Result/*s/range)
    Content(, , LearnByExample("Above high normal"), /Message/*s/Result/*s/comment)
    Content(, , LearnByExample("Final results"), /Message/*s/Result/*s/status)
```

When you finish, the markup in the example pane should look like this:



```
MSH|^~\&|LAB||CDB|||ORU^R01|K172|P
PID|||PATID1234^5^M11||Jones^William||19610613|M
OBR||||80004^Electrolytes
OBX|1|ST|84295^Na||150|mmol/l|136-148|Above high normal|||Final results
OBX|2|ST|84132^K+||4.5|mmol/l|3.5-5|Normal|||Final results
OBX|3|ST|82435^Cl||102|mmol/l|94-105|Normal|||Final results
OBX|4|ST|82374^CO2||27|mmol/l|24-31|Normal|||Final results
```

Editing the IntelliScript

The procedure illustrated above is the general way to edit the IntelliScript.

To edit the IntelliScript:

1. Select the desired location for editing.
2. Press Enter. In most locations, you can also double-click instead of pressing Enter.
3. Choose or type a value.
4. Press Enter again to accept the edited value.

Testing the Parser

There are several ways to test the parser and confirm that it works correctly:

- You can view the color coding in the example source. This tests the basic anchor configuration.
- You can run the parser, confirm that the events are error-free, and view the XML output. This tests the parser operation on the example source.
- You can run the parser on additional source documents. This confirms that the parser can process variations of the source structure that occur in the documents.

In this exercise, you will use the first two methods to test the parser. We will not take the time for the third test, although it is easy enough to do. For more information, see "Running the Parser on Additional Source Documents" on page 19.

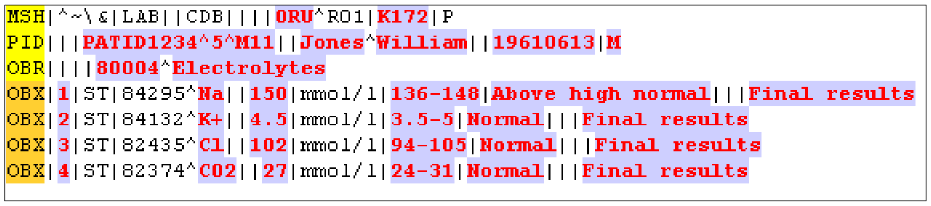
To test the parser:

1. On the menu, click IntelliScript > Mark Example. Alternatively, you can click the button near the right end of the toolbar, which is labeled Mark the Entire Example According to the Current Script.

   Notice that the color-coding is extended throughout the example source document. Previously, only the anchors that you defined in the first line of the repeating group were highlighted. When you ran the Mark Example command, Conversion Agent ran the parser and found the anchors in the other lines.

   Confirm that the marking is as you expect. For example, check that the test value, range, and comment are correctly identified in each `OBX` line.

   If the marking is not correct, or if there is no marking, there is a mistake in the parser configuration. Review the instructions, correct the mistake, and try again.

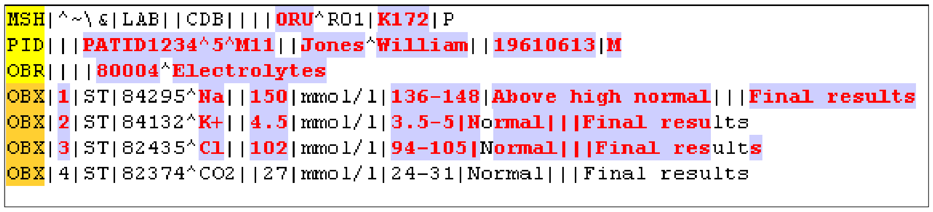


```
MSH|^~\&|LAB||CDB|||ORU^R01|K172|P
PID|||PATID1234^5^M11||Jones^William||19610613|M
OBR||||80004^Electrolytes
OBX|1|ST|84295^Na||150|mmol/l|136-148|Above high normal|||Final results
OBX|2|ST|84132^K+||4.5|mmol/l|3.5-5|Normal|||Final results
OBX|3|ST|82435^Cl||102|mmol/l|94-105|Normal|||Final results
OBX|4|ST|82374^CO2||27|mmol/l|24-31|Normal|||Final results
```

2. As an experiment, you can test what would happen if you made a deliberate error in the configuration.

   Do you remember the HL7 delimiters option, which you set in the New Parser wizard? Try changing the option to another value:

   - Save your work. This can be helpful if you make a serious error during this experiment.
   - Edit the `delimiters` property. It's located under `Parser/format`.
   - Change the value from `HL7` to `Positional`. This means that the `Content` anchors are located at fixed positions, measured by numbers of characters, after the `Marker` anchors.
   - Try the Mark Example command again. The anchors are incorrectly identified. For example, in the second OBX line, the comment is reported as `rmal|||Final resu` instead of `Normal`. The problem occurs because the parser thinks that the anchors are located at fixed positions on the line.



```
MSH|^~\&|LAB||CDB|||ORU^R01|K172|P
PID|||PATID1234^5^M11||Jones^William||19610613|M
OBR||||80004^Electrolytes
OBX|1|ST|84295^Na||150|mmol/l|136-148|Above high normal|||Final results
OBX|2|ST|84132^K+||4.5|mmol/l|3.5-5|Normal|||Final results
OBX|3|ST|82435^Cl||102|mmol/l|94-105|Normal|||Final results
OBX|4|ST|82374^CO2||27|mmol/l|24-31|Normal|||Final results
```

   - Change the `delimiters` property back to `HL7`. Confirm that Mark Example now works correctly. Alternatively, click Edit > Undo to restore the previous configuration.
   - Save your work again.

3. Now, it is time to run the parser. Right-click the `Parser` component and set it as the startup component. Then use the Run > Run command to run it.

4. After a few moments, the Events view appears.

   Notice that most of the events are labeled with the information event icon (ℹ). This is normal when the parser contains no mistakes.

   If you search the event tree, you can find an event that is labeled with an optional failure icon (⚑). The event is located in the tree under `Execution/RepeatingGroup`, and it is labeled `Separator before 5`. This means that the `RepeatingGroup` failed to find a fifth iteration of the `OBX` separator. This is expected because the example source contains only four iterations. The failure is called optional because it is permitted for the separator to be missing at the end of the iterations.

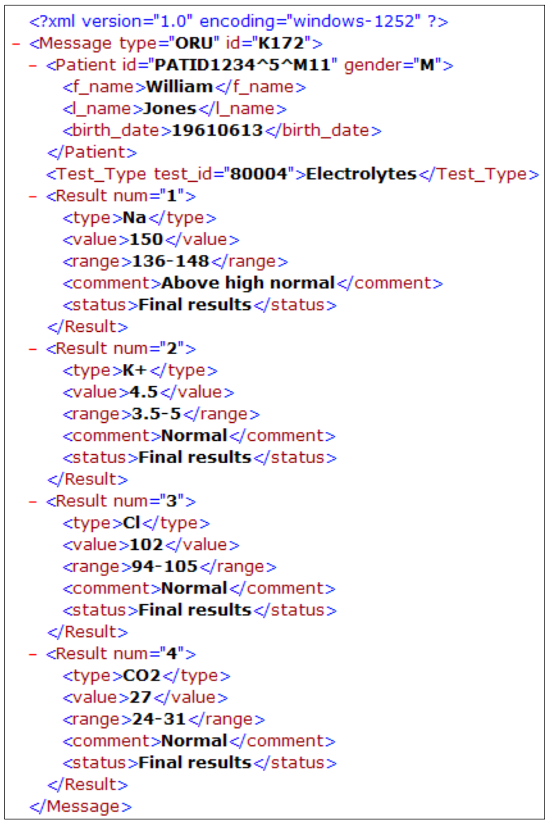
   Nested within the optional failure event, you can find a failure event icon (⚑). This event means that Conversion Agent failed to find the `Marker` anchor, which defines the `OBX` separator. Because the failure is nested within an optional failure, it is not a cause for concern. In general, however, you should pay attention to a failure event and make sure you understand what caused it. Often, a failure indicates a problem in the parser.

In addition to failure events, pay attention to warning event icons (⚠) and to fatal event icons (✖). Warnings are less severe than failures. Fatal errors are the most severe. They stop the transformation from running.

5. In the right pane of the Events view, try double-clicking one of the `Marker` or `Content` events.

   When you do this, Conversion Agent highlights the anchor that caused the event in the IntelliScript and example panes. This is a good way to find the source of failure or error events.

6. In the Conversion Agent Explorer, double-click the `output.xml` file, located under the `Tutorial_2 files/Results` node. You should see the following XML:



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Message type="ORU" id="K172">
  - <Patient id="PATID1234^5^M11" gender="M">
      <f_name>William</f_name>
      <l_name>Jones</l_name>
      <birth_date>19610613</birth_date>
    </Patient>
    <Test_Type test_id="80004">Electrolytes</Test_Type>
  - <Result num="1">
      <type>Na</type>
      <value>150</value>
      <range>136-148</range>
      <comment>Above high normal</comment>
      <status>Final results</status>
    </Result>
  - <Result num="2">
      <type>K+</type>
      <value>4.5</value>
      <range>3.5-5</range>
      <comment>Normal</comment>
      <status>Final results</status>
    </Result>
  - <Result num="3">
      <type>Cl</type>
      <value>102</value>
      <range>94-105</range>
      <comment>Normal</comment>
      <status>Final results</status>
    </Result>
  - <Result num="4">
      <type>CO2</type>
      <value>27</value>
      <range>24-31</range>
      <comment>Normal</comment>
      <status>Final results</status>
    </Result>
  </Message>
```

Points to Remember

To create a new project, run the File > New > Project command. This displays a wizard, where you can set options such as:

- The parser name
- The XSD schema for the output XML
- The example source document, such as a file
- The source format, such as text or binary
- The delimiters that separate the data fields

After you create the project, edit the IntelliScript and add the anchors, such as `Marker` and `Content` for simple data structures, or `RepeatingGroup` for repetitive structures.

To edit the IntelliScript, use the Select-Enter-Assign-Enter approach. That is, select the location that you want to edit. Press the Enter key. Assign the property value, and press Enter again.

Click IntelliScript > Mark Example to color-code the markers.

Click Run > Run to execute the parser. View the results file, which contains the output XML.

CHAPTER 4

Positional Parsing of a PDF Document

This chapter includes the following topics:

- ♦ Overview, 35
- ♦ Creating the Project, 38
- ♦ Defining the Anchors, 40
- ♦ Defining the Nested Repeating Groups, 44
- ♦ Using an Action to Compute Subtotals, 50
- ♦ Points to Remember, 52

Overview

In many parsing applications, the source documents have a fixed page layout. This is true, for example, of bills, invoices, and account statements. In such cases, you can configure a parser that uses a positional format to find the data fields.

In this chapter, you will use a positional strategy to parse an invoice form. You will define the Content anchors according to their character offsets from the Marker anchors.

In addition to the positional strategy, the exercise illustrates several other important Conversion Agent features:

- ♦ The source document is a PDF file. The parser uses a document processor to convert the document from the binary PDF format to a text format, which is suitable for further processing.
- ♦ The data is organized in nested repeating groups.
- ♦ The parser uses actions to compute subtotals, which are not present in the source document.
- ♦ The document contains a large quantity of irrelevant data that is not required for parsing. Some of the irrelevant data contains the same marker strings as the desired data. The exercise introduces the concept of search scope, which you can use to narrow the search for anchors and identify the desired data reliably.
- ♦ To configure the parser, you will use both the basic properties and the advanced properties of components. The advanced properties are hidden but can be displayed on demand.

In this exercise, you will solve a complex, real-life parsing problem. Have patience as you do the exercise. Do not be afraid to delete or undo your work if you make a mistake. With a little practice, you will be able to create parsers like this easily

Requirements Analysis

Before you start to configure a Conversion Agent project, examine the source document and the desired XML output, and analyze what the transformation needs to do.

Source Document

To view the PDF source document, you need the Adobe Reader, which you can download from http://www.adobe.com. You need the Reader only for viewing the document. Conversion Agent has a built-in component for processing PDF documents and does not require any additional PDF software.

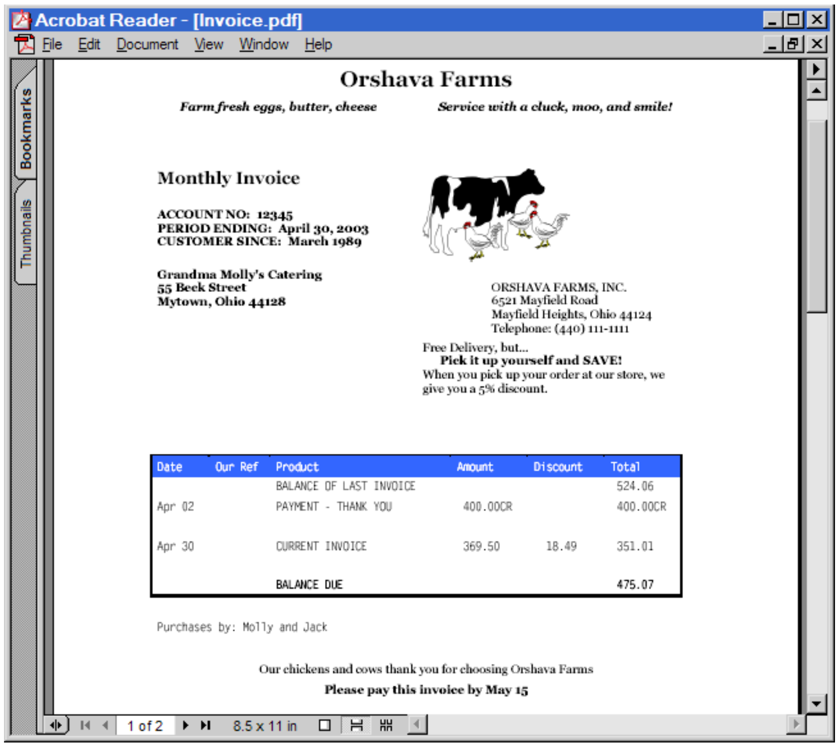
In the Adobe Reader, open the file in the Conversion Agent installation folder:

```
tutorials\Exercises\Files_for_Tutorial_3\Invoice.pdf
```

The document is an invoice that a fictitious egg-and-dairy wholesaler, called Orshava Farms, sends to its customers. The first page of the invoice displays data such as:

- The customer's name, address, and account number
- The invoice date
- A summary of the current charges
- The total amount due

The top and bottom of the first page display boilerplate text and advertising.


Acrobat Reader - [Invoice.pdf]
File Edit Document View Window Help
Bookmarks
Thumbnails
1 of 2
8.5 x 11 in

Orshava Farms

Farm fresh eggs, butter, cheese *Service with a cluck, moo, and smile!*

Monthly Invoice

ACCOUNT NO: 12345
PERIOD ENDING: April 30, 2003
CUSTOMER SINCE: March 1989

Grandma Molly's Catering
55 Beck Street
Mytown, Ohio 44128

ORSHAVA FARMS, INC.
6521 Mayfield Road
Mayfield Heights, Ohio 44124
Telephone: (440) 111-1111

Free Delivery, but...
Pick it up yourself and SAVE!
When you pick up your order at our store, we give you a 5% discount.

Date	Our Ref	Product	Amount	Discount	Total
		BALANCE OF LAST INVOICE			524.06
Apr 02		PAYMENT - THANK YOU	400.00CR		400.00CR
Apr 30		CURRENT INVOICE	369.50	18.49	351.01
		BALANCE DUE			475.07

Purchases by: Molly and Jack

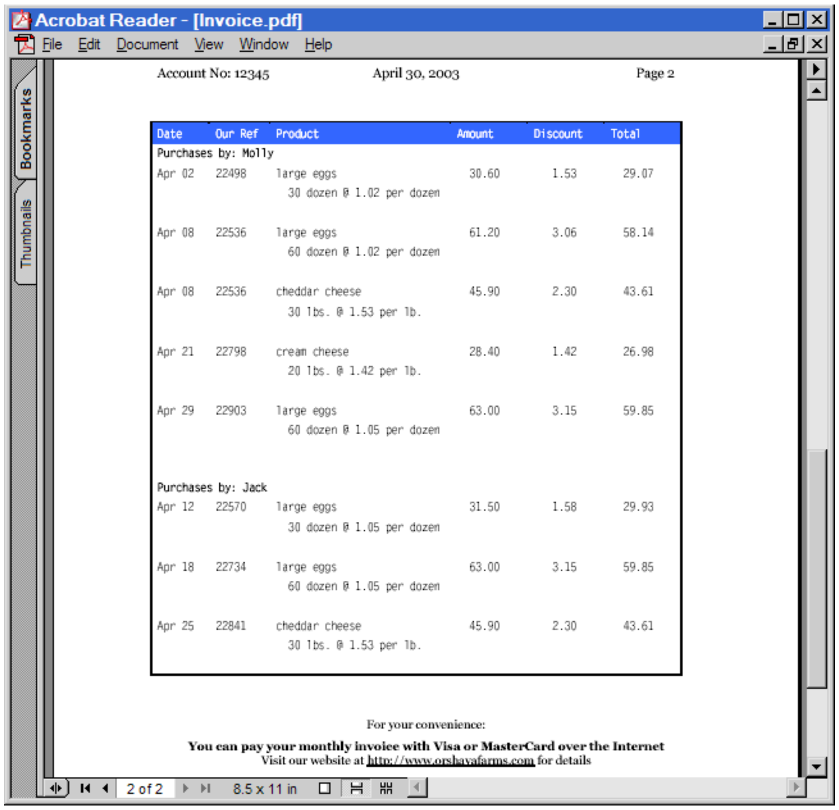
Our chickens and cows thank you for choosing Orshava Farms

Please pay this invoice by May 15

The second page displays the itemized charges for each buyer. The sample document lists two buyers, each of whom made multiple purchases. The page has a nested repeating structure:

- The main section is repeated for each buyer.
- Within the section for each buyer, there is a two-line structure, followed by a blank space, for each purchase transaction.

At the top of the second page, there is a page header. At the bottom, there is additional boilerplate text.


Acrobat Reader - [Invoice.pdf]
File Edit Document View Window Help
Bookmarks
Thumbnails
2 of 2
8.5 x 11 in

Account No: 12345 April 30, 2003 Page 2

Date	Our Ref	Product	Amount	Discount	Total
Purchases by: Molly					
Apr 02	22498	large eggs	30.60	1.53	29.07
		30 dozen @ 1.02 per dozen			
Apr 08	22536	large eggs	61.20	3.06	58.14
		60 dozen @ 1.02 per dozen			
Apr 08	22536	cheddar cheese	45.90	2.30	43.61
		30 lbs. @ 1.53 per lb.			
Apr 21	22798	cream cheese	28.40	1.42	26.98
		20 lbs. @ 1.42 per lb.			
Apr 29	22903	large eggs	63.00	3.15	59.85
		60 dozen @ 1.05 per dozen			
Purchases by: Jack					
Apr 12	22570	large eggs	31.50	1.58	29.93
		30 dozen @ 1.05 per dozen			
Apr 18	22734	large eggs	63.00	3.15	59.85
		60 dozen @ 1.05 per dozen			
Apr 25	22841	cheddar cheese	45.90	2.30	43.61
		30 lbs. @ 1.53 per lb.			

For your convenience:

You can pay your monthly invoice with Visa or MasterCard over the Internet
Visit our website at http://www.orshayafarms.com for details

This structure is typical of many invoices: a summary page, followed by repeating structures for different account numbers and credit card numbers. A business might store such invoices as PDF files instead of saving paper copies. It might use the PDF invoices for online billing by email or via a web site.

Your task, in designing the parser, is to retrieve the required data while ignoring the boilerplate. Since you are doing this, presumably for a system integration application, you must do it with a very high degree of reliability.

XML Output

For the purpose of this exercise, we assume that you want to retrieve the transaction data from the invoice. You need to store the data in an XML structure, which looks like this:

```
<Invoice account="12345">
  <Period_Ending>April 30, 2003</Period_Ending>
  <Current_Total>351.01</Current_Total>
  <Balance_Due>457.07</Balance_Due>
  <Buyer name="Molly" total="217.64">
    <Transaction date="Apr 02" ref="22498">
      <Product>large eggs</Product>
      <Total>29.07</Total>
    </Transaction>
    <Transaction date="Apr 08" ref="22536">
      <Product>large eggs</Product>
      <Total>58.14</Total>
    </Transaction>
    <!-- Additional transaction elements -->
  </Buyer>
  <Buyer name="Jack" total="133.37">
    <!-- Transaction elements -->
  </Buyer>
</Invoice>
```

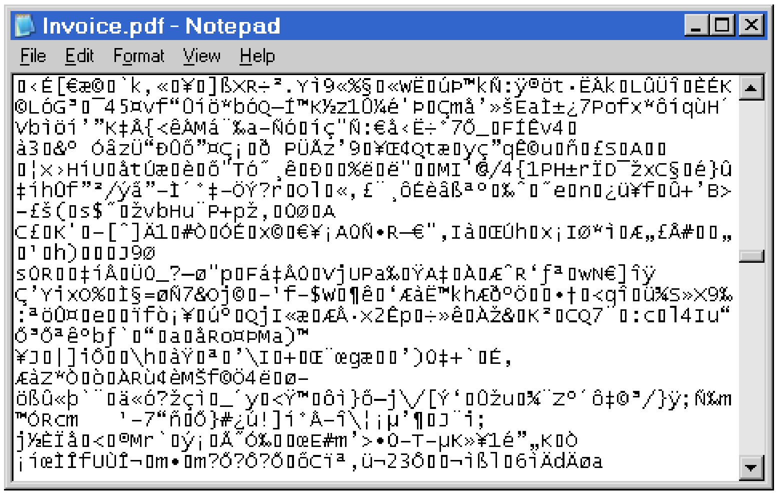
The structure contains multiple `Buyer` elements, and each `Buyer` element contains multiple `Transaction` elements. Each `Transaction` element contains selected data about a transaction: the date, reference number, product, and total price.

The structure omits other data about a transaction, which we choose to ignore. For example, the structure omits the discount, the quantity of each product, and the unit price.

Each `Buyer` element has a `total` attribute, which is the total of the buyer's purchases. The total per buyer is not recorded in the invoice. We require that Conversion Agent compute it.

The Parsing Problem

Try opening the `Invoice.pdf` file in Notepad. You will see something like this:



Parsing this binary data might be possible, but it would clearly be very difficult. We would need a detailed understanding of the internal PDF file format, and we would need to work very hard to identify the `Marker` and `Content` anchors unambiguously.

If we extract the text content of the document, the parsing problem seems more tractable:

```
Apr 08   22536     large eggs                     61.20      3.06     58.14
                     60 dozen @ 1.02 per dozen

Apr 08   22536     cheddar cheese                 45.90      2.30     43.61
                     30 lbs. @ 1.53 per lb.
```

The transaction data is aligned in columns. The position of each data field is fixed relative to the left and right margins. This is a perfect case for positional parsing—extracting the content according to its position on the page. That is why the positional format is appropriate for this exercise.

Another feature is that each transaction is recorded in a fixed pattern of lines. The first line contains the data that we wish to retrieve. The second line contains the quantity of a product and the unit price, which we do not need to retrieve. The following line is blank. We can use the repeating line structure to help parse the data.

A third feature is that the group of transactions is preceded by a heading row, such as:

```
Purchases by: Molly
```

The heading contains the buyer's name, which we need to retrieve. The heading also serves as a separator between groups of transactions.

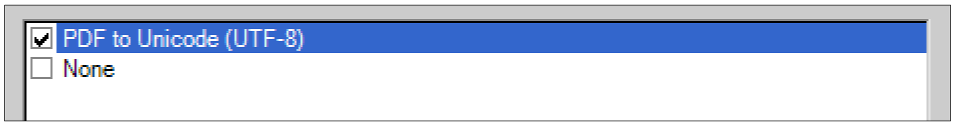
Creating the Project

Now that you understand the parsing requirements, you are ready to configure the Conversion Agent project.

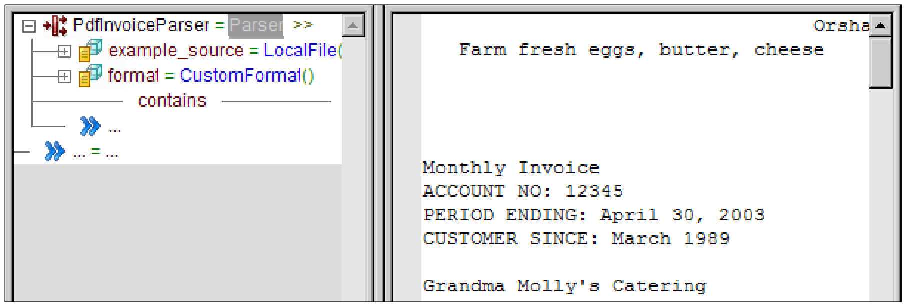
To create the project:

1. Use the File > New > Project command to create a parser project called `Tutorial_3`.
2. On the first few pages of the wizard, set the following options. The options are similar to the ones that you set in the preceding tutorial. For more information, see “Defining an HL7 Parser” on page 23.
   - Name the parser `PdfInvoiceParser`.
   - Name the script file `Pdf_ScriptFile`.
   - When prompted for the schema, browse to the file `OrshavaInvoice.xsd`, which is in following folder: `tutorials\Exercises\Files_For_Tutorial_3`.
   - Specify that the example source is a file on a local computer.
   - Browse to the example source file, which is `Invoice.pdf`.
   - Specify that the source content type is `PDF`.
3. When you reach the document processor page of the wizard, select the `PDF to Unicode (UTF-8)` processor.

   This processor converts the binary PDF format to the text format that the parser requires. The processor inserts spaces and line breaks in the text file, in an attempt to duplicate the format of the PDF file as closely as possible.

   
   [✓] PDF to Unicode (UTF-8)
   [ ] None

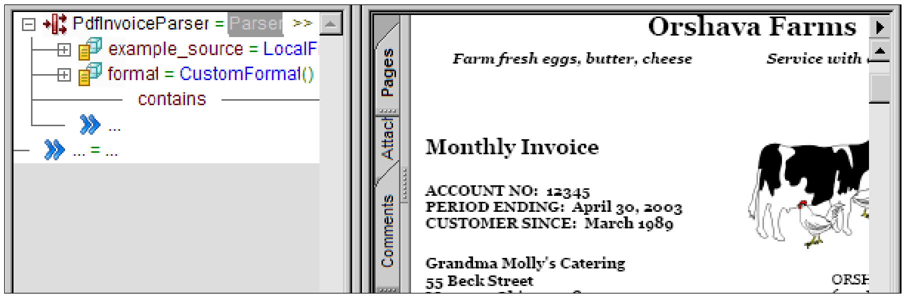
   Note: `PDF to Unicode (UTF-8)` is a description of the processor. In the IntelliScript, you can view the actual name of the processor, which is `PdfToTxt_3_01`.
4. On the next wizard page, select the document format, which is `CustomFormat`. You will change this value later, when you edit the IntelliScript.
5. On the final page, click Finish.
6. After a few seconds, the Conversion Agent Explorer displays the `Tutorial_3` project, and the script file is opened in an IntelliScript editor.
7. Notice that the example pane displays the example source in text format, which is the output of the document processor. You will configure anchors that process the text.

```
PdfInvoiceParser = Parser >>
  example_source = LocalFile(
  format = CustomFormat()
  contains
  ...
... = ...
```

```
                                          Orsha
   Farm fresh eggs, butter, cheese




Monthly Invoice
ACCOUNT NO: 12345
PERIOD ENDING: April 30, 2003
CUSTOMER SINCE: March 1989

Grandma Molly's Catering
```

8. Try clicking the Browser tab at the bottom of the example pane. This tab displays the original document in a Microsoft Internet Explorer window, which uses the Adobe Reader as a plug-in component to display the file. The Browser tab is for display only. Return to the Source tab to configure the anchors.

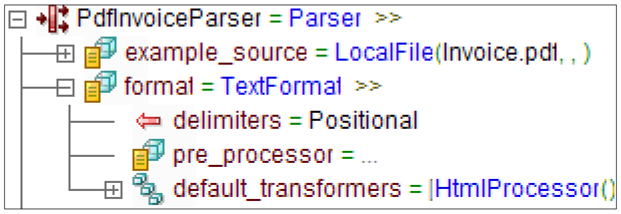


```
PdfInvoiceParser = Parser >>
    example_source = LocalF
    format = CustomFormat()
        contains
    ...
... = ...
```

9. Now that the project and parser have been created, you can do some fine tuning by editing the IntelliScript.

   Expand the `format` property of the parser, and change its value from `CustomFormat` to `TextFormat`. This is appropriate because the parser will process the text form of the document, which is the output of the document processor.

   Under the `format` property, change `delimiters` to `Positional`. This means that the parser learns the structure of the example source by counting the characters between the anchors.



```
PdfInvoiceParser = Parser >>
    example_source = LocalFile(Invoice.pdf, , )
    format = TextFormat >>
        delimiters = Positional
        pre_processor = ...
        default_transformers = HtmlProcessor()
```

Defining the Anchors

Now, you will define the anchors. You will use the parse-by-example approach to do this.

To define the anchors:

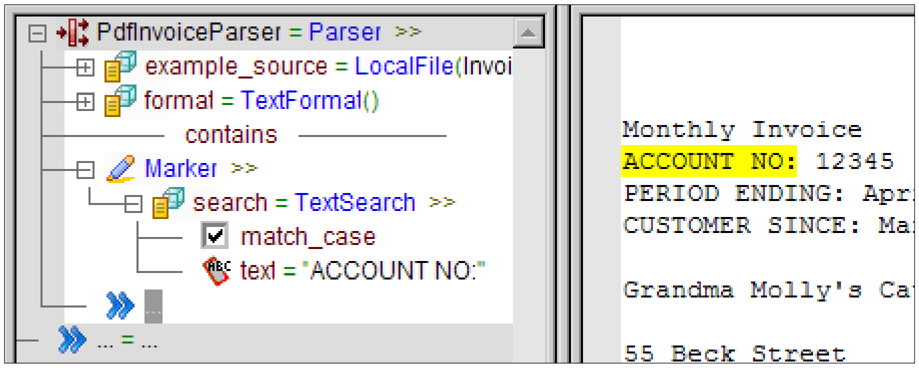
1. Near the top of the example source, find the string `ACCOUNT NO:`, which marks the beginning of the text that you need to parse. Define the string as a `Marker` anchor by selecting it, right-clicking, and choosing Insert Marker.

   The `Marker` is automatically displayed in the IntelliScript, and you are prompted to enter the `search` property setting. Select the value `TextSearch`, which is the default. Under `TextSearch`, confirm that the `text` string is `ACCOUNT NO:`.

   Click the >> symbol at the end of `TextSearch` to display its advanced properties. Select the match case option under `Marker`. The option means that the parser looks only for the string `ACCOUNT NO:`, and not `account no:`, `Account No:`, or any other spelling. This can help prevent problems if one of the other spellings happens to occur in the document. In fact, the spelling `Account No:` occurs in the header of the second page.

   Although it is not strictly necessary, we suggest that you select the match case option for all the `Marker` anchors that you define in this exercise.

For more information about the advanced properties, see "Basic and Advanced Properties" on page 49.

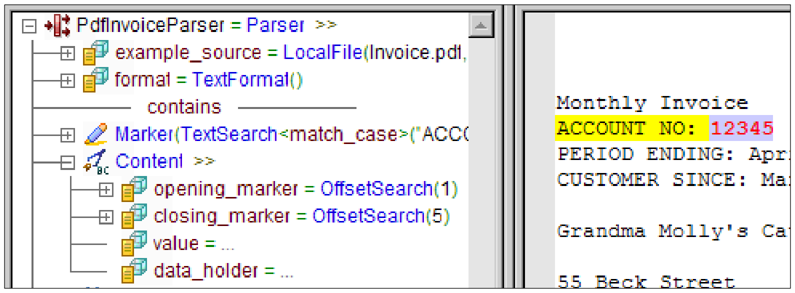

```
PdfInvoiceParser = Parser >>
  example_source = LocalFile(Invoi
  format = TextFormat()
  contains
  Marker >>
    search = TextSearch >>
      match_case
      text = "ACCOUNT NO:"
  ...
  ... = ...
```

```
Monthly Invoice
ACCOUNT NO:  12345
PERIOD ENDING: Apr
CUSTOMER SINCE: Ma

Grandma Molly's Ca

55 Beck Street
```

2. Continuing on the same line of the source document, after the `Marker`, select and right-click `12345`. On the menu, click Insert Offset Content.

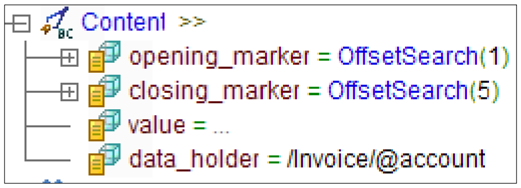
   This inserts a `Content` anchor, with property values that are appropriate for positional parsing. The properties are as follows:

Property	Description
`opening_marker = OffsetSearch(1)`	This means that the text `12345` starts 1 character after the `Marker` anchor.
`closing_marker = OffsetSearch(5)`	This means that the text `12345` ends 5 characters after its start.


```
PdfInvoiceParser = Parser >>
  example_source = LocalFile(Invoice.pdf,
  format = TextFormat()
  contains
  Marker(TextSearch<match_case>("ACC
  Content >>
    opening_marker = OffsetSearch(1)
    closing_marker = OffsetSearch(5)
    value = ...
    data_holder = ...
```

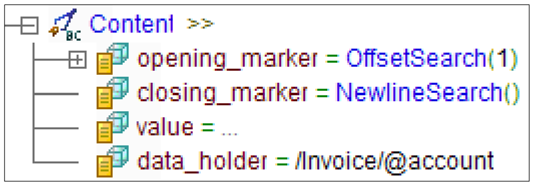
```
Monthly Invoice
ACCOUNT NO: 12345
PERIOD ENDING: Apr
CUSTOMER SINCE: Ma

Grandma Molly's Ca

55 Beck Street
```

3. Edit the `data_holder` property of the `Content` anchor. Set its value to `/Invoice/@account`. This is the data holder where the `Content` anchor will store the text that it retrieves from the source document.


```
Content >>
  opening_marker = OffsetSearch(1)
  closing_marker = OffsetSearch(5)
  value = ...
  data_holder = /Invoice/@account
```

4. There is a small problem in the definition of the `Content` anchor. What will happen if a source document has an account number such as `987654321`, which is longer than the number in the example source? According to the current definition of the `closing_marker`, the parser will retrieve only the first five digits, truncating the value to `98765`.

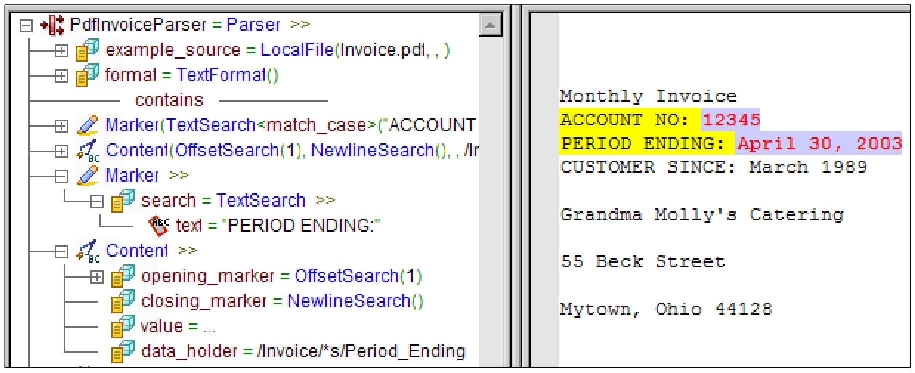
To solve this potential problem, change the `closing_marker` to retrieve not only until character 5, but until the end of the text line. Do this by changing the value of `closing_marker` from `OffsetSearch` to `NewlineSearch`.



```
Content >>
    opening_marker = OffsetSearch(1)
    closing_marker = NewlineSearch()
    value = ...
    data_holder = /Invoice/@account
```

5. Continuing to the next line of the example source, define `PERIOD ENDING:` as a `Marker` anchor.

6. Define `April 30, 2003` as a `Content` anchor with the offset properties. Map the `Content` to the `/Invoice/*s/Period_Ending` data holder. Change the `closing_marker` to `NewlineSearch`, to support source documents where the date string is longer than in the example source.



```
PdfInvoiceParser = Parser >>
    example_source = LocalFile(Invoice.pdf, , )
    format = TextFormat()
        contains
    Marker(TextSearch<match_case>("ACCOUNT
    Content(OffsetSearch(1), NewlineSearch(), , /Ir
    Marker >>
        search = TextSearch >>
            text = "PERIOD ENDING:"
    Content >>
        opening_marker = OffsetSearch(1)
        closing_marker = NewlineSearch()
        value = ...
        data_holder = /Invoice/*s/Period_Ending
```

```
Monthly Invoice
ACCOUNT NO: 12345
PERIOD ENDING: April 30, 2003
CUSTOMER SINCE: March 1989

Grandma Molly's Catering

55 Beck Street

Mytown, Ohio 44128
```

7. The next data that you need to retrieve is the current invoice amount. Scroll a few lines down, and define `CURRENT INVOICE` as a `Marker`.

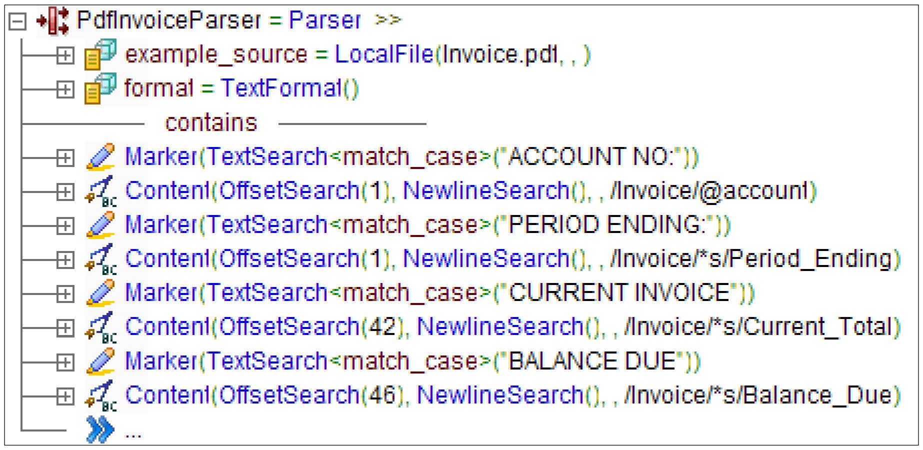
8. At the end of the same line, define `351.01` as an offset `Content` anchor. Map the `Content` to the `/Invoice/*s/Current_Total` data holder.

   When you select the `Content`, include a few space characters to the left of the string `351.01`. In positional parsing, this is important because the area to the left might contain additional characters. For example, the content might be `1351.01`.

   As above, change `closing_marker` to `NewlineSearch`, in case the number is located to the right of its position in the example source.

9. In the same way, define `BALANCE DUE` as a `Marker` and `475.07` as `Content`. Map the `Content` to `/Invoice/*s/Balance_Due`.

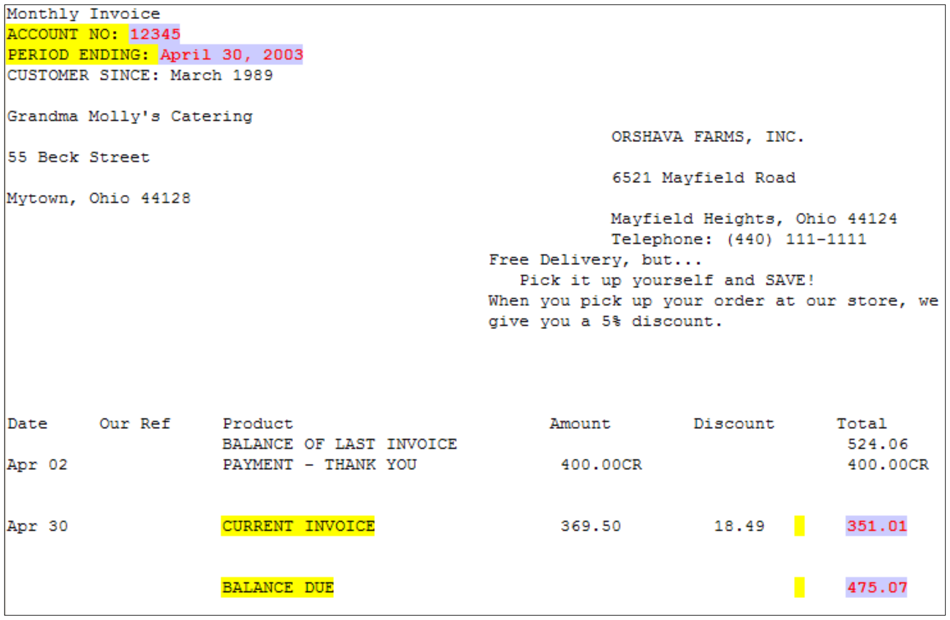
10. Examine the IntelliScript, and confirm that the sequence of `Marker` and offset-`Content` anchors is correct. It should look like this:



```
PdfInvoiceParser = Parser >>
    example_source = LocalFile(Invoice.pdf, , )
    format = TextFormat()
        contains
    Marker(TextSearch<match_case>("ACCOUNT NO:"))
    Content(OffsetSearch(1), NewlineSearch(), , /Invoice/@account)
    Marker(TextSearch<match_case>("PERIOD ENDING:"))
    Content(OffsetSearch(1), NewlineSearch(), , /Invoice/*s/Period_Ending)
    Marker(TextSearch<match_case>("CURRENT INVOICE"))
    Content(OffsetSearch(42), NewlineSearch(), , /Invoice/*s/Current_Total)
    Marker(TextSearch<match_case>("BALANCE DUE"))
    Content(OffsetSearch(46), NewlineSearch(), , /Invoice/*s/Balance_Due)
    ...
```

The precise offsets in your solution might differ from the ones in the illustration. The offsets depend on the number of spaces that you selected to the left or right of each anchor.

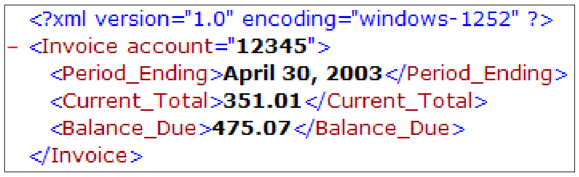
11. Examine the color coding in the example pane. It should look like this:



```
Monthly Invoice
ACCOUNT NO: 12345
PERIOD ENDING: April 30, 2003
CUSTOMER SINCE: March 1989

Grandma Molly's Catering
                                                            ORSHAVA FARMS, INC.
55 Beck Street
                                                            6521 Mayfield Road
Mytown, Ohio 44128
                                                            Mayfield Heights, Ohio 44124
                                                            Telephone: (440) 111-1111
                                                Free Delivery, but...
                                                   Pick it up yourself and SAVE!
                                                When you pick up your order at our store, we
                                                give you a 5% discount.




Date     Our Ref     Product                         Amount        Discount      Total
                     BALANCE OF LAST INVOICE                                      524.06
Apr 02               PAYMENT - THANK YOU              400.00CR                    400.00CR


Apr 30               CURRENT INVOICE                  369.50         18.49        351.01


                     BALANCE DUE                                                  475.07
```

12. Right-click and define `PdfInvoiceParser` as the startup component.

13. Run the parser and view the results file. The file should look like this:



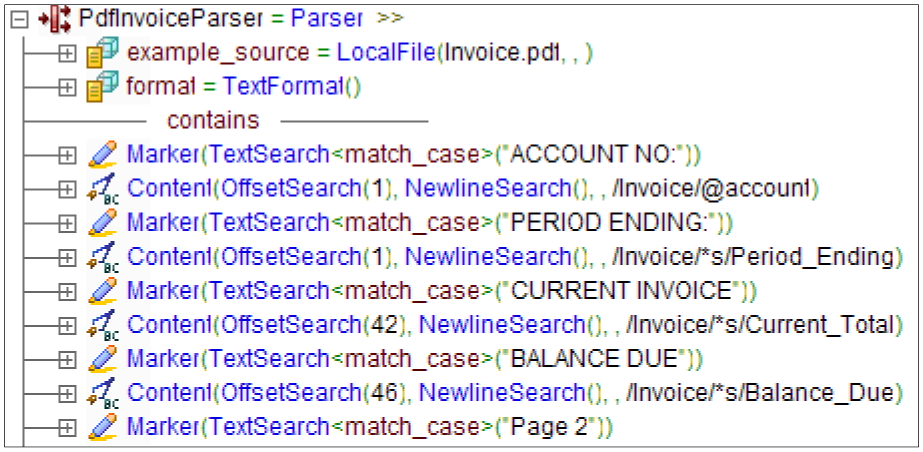
```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Invoice account="12345">
    <Period_Ending>April 30, 2003</Period_Ending>
    <Current_Total>351.01</Current_Total>
    <Balance_Due>475.07</Balance_Due>
  </Invoice>
```

14. You are now almost ready to define the repeating group, which creates the `Buyer` elements of the XML output. Afterwards, you will define a nested repeating group, which creates the `Transaction` elements.

The repeating group starts at the `Purchases by:` line, which is on page 2 of the invoice. You will define `Purchases by:` as the separator of the repeating group.

There is a catch, however. The string `Purchases by:` appears also on the bottom of page 1. To prevent this from causing a problem, you need to start the search scope for the repeating group at the beginning of page 2.

To do this, define the string `Page 2` as a `Marker` anchor. By default, the parser assumes that the anchors are defined in the order that they appear in the document. Therefore, it starts looking for the subsequent repeating group from the end of this `Marker`.



```
PdfInvoiceParser = Parser >>
  example_source = LocalFile(Invoice.pdf, , )
  format = TextFormat()
  contains
  Marker(TextSearch<match_case>("ACCOUNT NO:"))
  Content(OffsetSearch(1), NewlineSearch(), , /Invoice/@account)
  Marker(TextSearch<match_case>("PERIOD ENDING:"))
  Content(OffsetSearch(1), NewlineSearch(), , /Invoice/*s/Period_Ending)
  Marker(TextSearch<match_case>("CURRENT INVOICE"))
  Content(OffsetSearch(42), NewlineSearch(), , /Invoice/*s/Current_Total)
  Marker(TextSearch<match_case>("BALANCE DUE"))
  Content(OffsetSearch(46), NewlineSearch(), , /Invoice/*s/Balance_Due)
  Marker(TextSearch<match_case>("Page 2"))
```

More About Search Scope

The `Marker` for `Page 2` is used to establish the search scope of the subsequent repeating group. Every `Marker` anchor is to establish the search scope of other anchors. For example, suppose that a `Content` anchor is located between two `Marker` anchors. Conversion Agent finds the `Marker` anchors in an initial pass through the document, called the initial phase. In a second pass, called the main phase, the software searches for the `Content` between the `Marker` anchors. You have already seen many examples of how this works as you performed the preceding exercises.

There are many ways to refine the search scope and the way in which the software searches for anchors. For example, you can configure Conversion Agent to:

- Search for particular anchors in the initial, main, or final phase of the processing
- Search backwards for anchors
- Permit anchors to overlap one another
- Find anchors according to search criteria such as a specified text string, a pattern, which is a regular expression, or a specified data type

Adjusting the search scope, phase, and criteria is one of the most powerful features that Conversion Agent offers for parsing complex documents. For more information about anchor features, see the *Conversion Agent Studio User Guide*.

Defining the Nested Repeating Groups

Now, you need to define a `RepeatingGroup` anchor that parses the repeating `Purchases by:` structure. Within it, you will define a nested `RepeatingGroup` that parses the individual transactions.

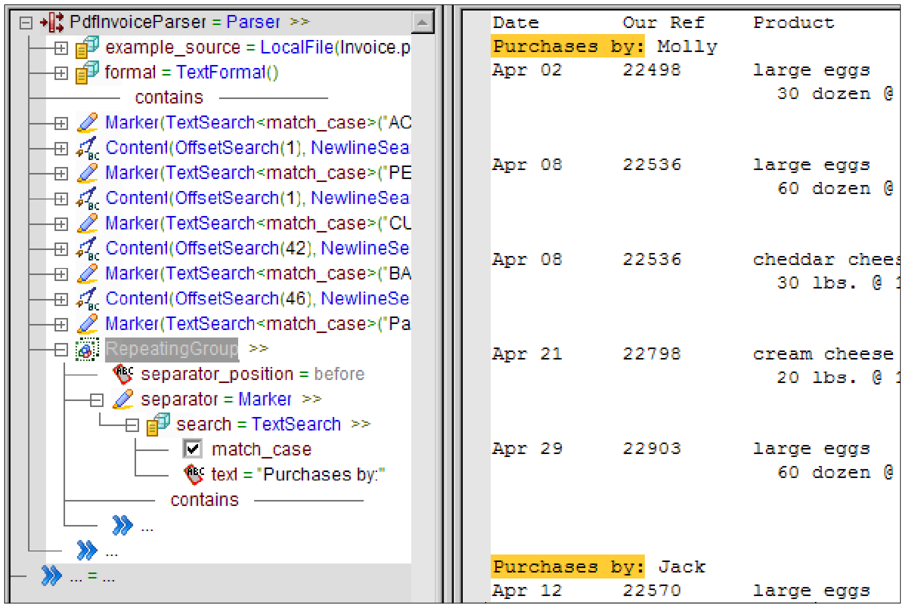
Do you remember how to define a repeating group? For more information about the procedure, see “Defining an HL7 Parser” on page 23.

To define the repeating groups:

1. At the end of the anchor list, edit the IntelliScript to insert a `RepeatingGroup`.
2. Define the `separator` property of the `RepeatingGroup` as a `Marker`, which performs a `TextSearch` for the string `Purchases by:`.

Set the separator_position to before. This is appropriate because the separator is located before each iteration of the repeating group.

Conversion Agent highlights the Purchases by: strings in the example pane.

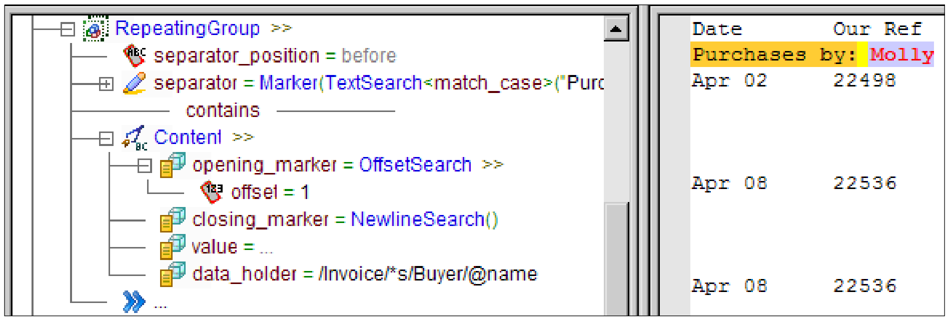


```
PdfInvoiceParser = Parser >>
  example_source = LocalFile(Invoice.p
  format = TextFormat()
  contains
  Marker(TextSearch<match_case>("AC
  Content(OffsetSearch(1), NewlineSea
  Marker(TextSearch<match_case>("PE
  Content(OffsetSearch(1), NewlineSea
  Marker(TextSearch<match_case>("CU
  Content(OffsetSearch(42), NewlineSe
  Marker(TextSearch<match_case>("BA
  Content(OffsetSearch(46), NewlineSe
  Marker(TextSearch<match_case>("Pa
  RepeatingGroup >>
    separator_position = before
    separator = Marker >>
      search = TextSearch >>
        match_case
        text = "Purchases by:"
    contains
    ...
  ...
... = ...
```

```
Date        Our Ref      Product
Purchases by: Molly
Apr 02      22498        large eggs
                           30 dozen @

Apr 08      22536        large eggs
                           60 dozen @

Apr 08      22536        cheddar chees
                           30 lbs. @ 1

Apr 21      22798        cream cheese
                           20 lbs. @ 1

Apr 29      22903        large eggs
                           60 dozen @

Purchases by: Jack
Apr 12      22570        large eggs
```

3. Within the RepeatingGroup, insert an offset Content anchor that maps the buyer's name, Molly, to the data holder /Invoice/*s/Buyer/@name.

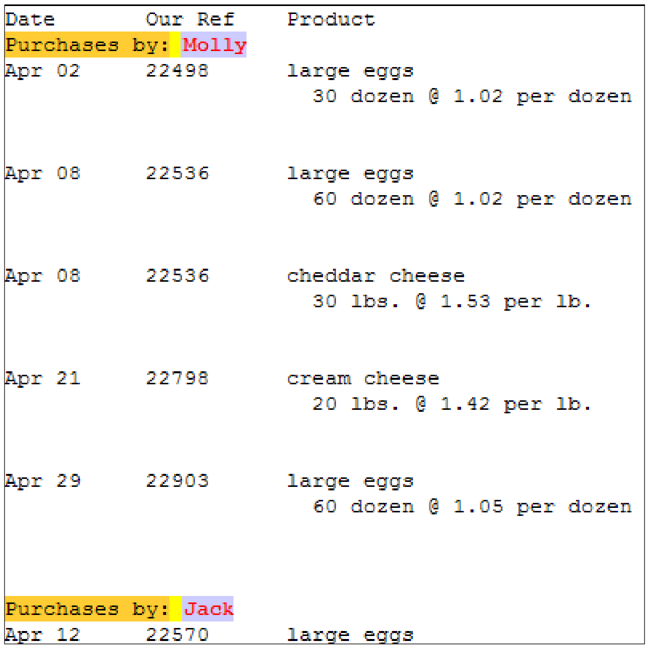
Do not forget to change the closing_marker to NewlineSearch, in case the buyer's name contains more than five letters. We won't remind you about this any more. When you define an offset Content anchor, you should confirm that it supports strings that are longer than the ones in the example source.



```
RepeatingGroup >>
  separator_position = before
  separator = Marker(TextSearch<match_case>("Purc
  contains
  Content >>
    opening_marker = OffsetSearch >>
      offset = 1
    closing_marker = NewlineSearch()
    value = ...
    data_holder = /Invoice/*s/Buyer/@name
  ...
```

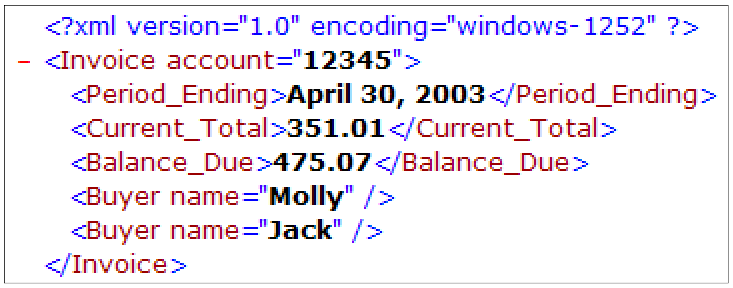
```
Date        Our Ref
Purchases by: Molly
Apr 02      22498

Apr 08      22536

Apr 08      22536
```

4. On the menu, click IntelliScript > Mark Example. Check that both buyer names, `Molly` and `Jack`, are highlighted. This confirms that the parser identifies the iterations of the `RepeatingGroup` correctly



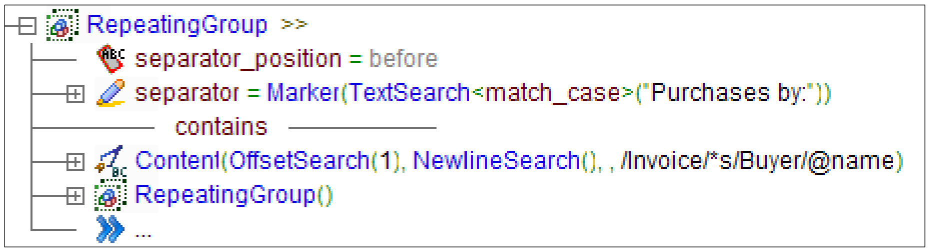
```
Date          Our Ref       Product
Purchases by: Molly
Apr 02        22498         large eggs
                              30 dozen @ 1.02 per dozen


Apr 08        22536         large eggs
                              60 dozen @ 1.02 per dozen


Apr 08        22536         cheddar cheese
                              30 lbs. @ 1.53 per lb.


Apr 21        22798         cream cheese
                              20 lbs. @ 1.42 per lb.


Apr 29        22903         large eggs
                              60 dozen @ 1.05 per dozen



Purchases by: Jack
Apr 12        22570         large eggs
```

If you want, you can also run the parser. Confirm that the results file contains two `Buyer` elements.



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Invoice account="12345">
    <Period_Ending>April 30, 2003</Period_Ending>
    <Current_Total>351.01</Current_Total>
    <Balance_Due>475.07</Balance_Due>
    <Buyer name="Molly" />
    <Buyer name="Jack" />
  </Invoice>
```

5. Find the three-dots symbol that is nested within the `RepeatingGroup`, and define another `RepeatingGroup`.



```
RepeatingGroup >>
    separator_position = before
    separator = Marker(TextSearch<match_case>("Purchases by:"))
    contains
    Content(OffsetSearch(1), NewlineSearch(), , /Invoice/*s/Buyer/@name)
    RepeatingGroup()
    ...
```

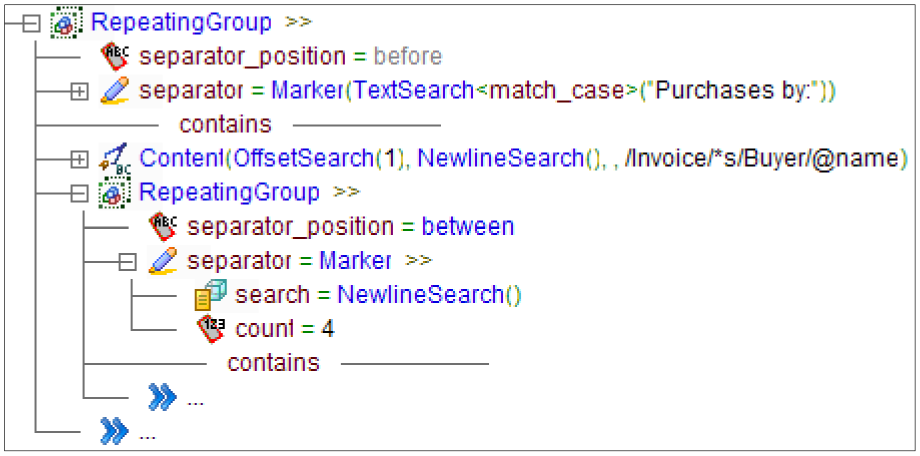
6. What can we use as the separator of the nested `RepeatingGroup`? There does not seem to be any characteristic text between the transactions.

Instead of a text separator, we can use the fact that every transaction occupies exactly four lines of the document. We can use the sequence of four newline characters as a separator.

To do this, set the `separator` property of the nested `RepeatingGroup` to a `Marker` that uses a `NewlineSearch`. Click the >> symbol at the end of the `Marker` to display its advanced properties, and set `count = 4`.

You can then click the << symbol to hide the advanced properties.

There is no separator before the first transaction or after the last transaction. To reflect this situation, set `separator_position = between`.



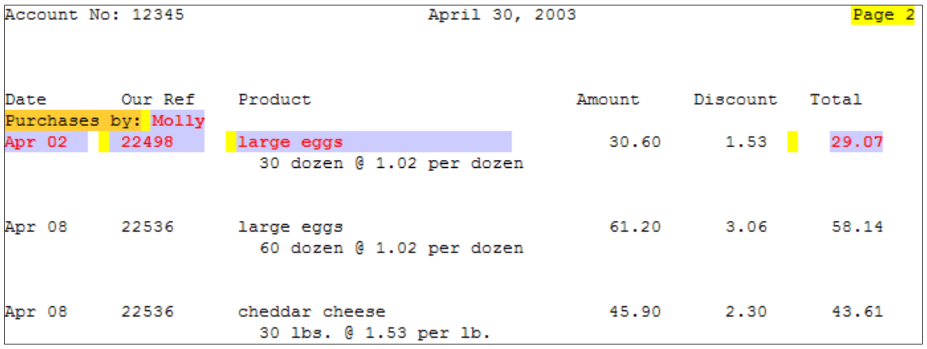
```
RepeatingGroup >>
    separator_position = before
    separator = Marker(TextSearch<match_case>("Purchases by:"))
    contains
    Content(OffsetSearch(1), NewlineSearch(), , /Invoice/*s/Buyer/@name)
    RepeatingGroup >>
        separator_position = between
        separator = Marker >>
            search = NewlineSearch()
            count = 4
        contains
        ...
    ...
```

If you examine the original PDF document carefully, you will discover that each transaction occupies only three lines. But we are parsing the output of the document processor. For some reason, undoubtedly related to the binary structure of the PDF file, the processor inserts an extra line.

7. In the first line of the first transaction, which is `Molly`'s first purchase, define the following offset `Content` anchors:

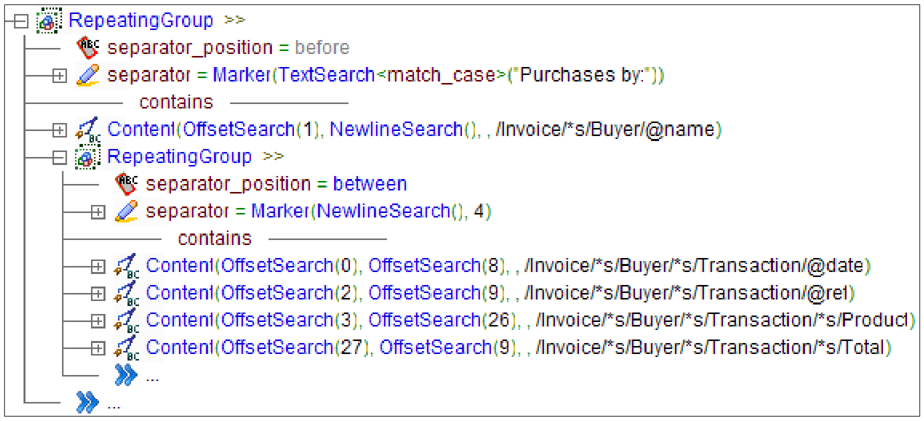
Content	Data Holder
`Apr 02`	`/Invoice/*s/Buyer/*s/Transaction/@date`
`22498`	`/Invoice/*s/Buyer/*s/Transaction/@ref`
`large eggs`	`/Invoice/*s/Buyer/*s/Transaction/*s/Product`
`29.07`	`/Invoice/*s/Buyer/*s/Transaction/*s/Total`

When you are done, the example pane should be color-coded like this. The exact position of the colors depends on the number of spaces that you selected to the left and right of each field.



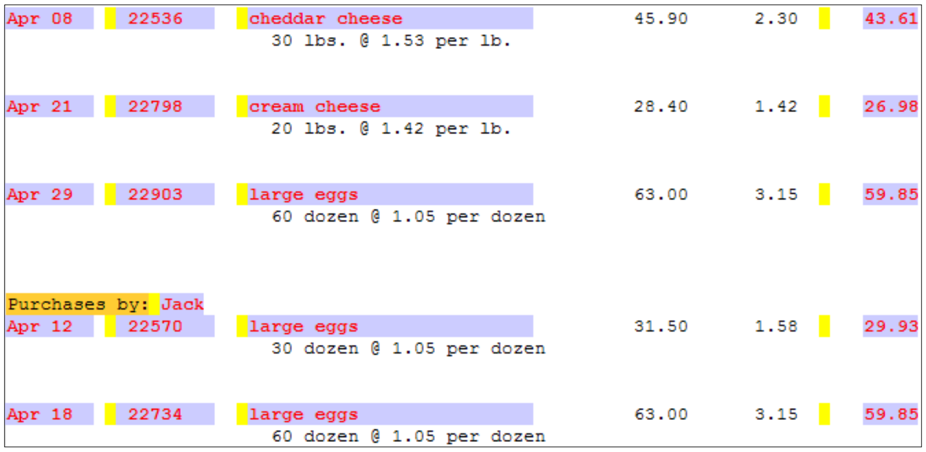
```
Account No: 12345                        April 30, 2003                        Page 2


Date          Our Ref     Product                         Amount      Discount     Total
Purchases by: Molly
Apr 02        22498       large eggs                         30.60         1.53        29.07
                            30 dozen @ 1.02 per dozen


Apr 08        22536       large eggs                         61.20         3.06        58.14
                            60 dozen @ 1.02 per dozen


Apr 08        22536       cheddar cheese                     45.90         2.30        43.61
                            30 lbs. @ 1.53 per lb.
```

The IntelliScript should display the four `Content` anchors within the nested `RepeatingGroup`.



```
RepeatingGroup >>
  separator_position = before
  separator = Marker(TextSearch<match_case>("Purchases by:"))
  contains
  Content(OffsetSearch(1), NewlineSearch(), , /Invoice/*s/Buyer/@name)
  RepeatingGroup >>
    separator_position = between
    separator = Marker(NewlineSearch(), 4)
    contains
    Content(OffsetSearch(0), OffsetSearch(8), , /Invoice/*s/Buyer/*s/Transaction/@date)
    Content(OffsetSearch(2), OffsetSearch(9), , /Invoice/*s/Buyer/*s/Transaction/@ref)
    Content(OffsetSearch(3), OffsetSearch(26), , /Invoice/*s/Buyer/*s/Transaction/*s/Product)
    Content(OffsetSearch(27), OffsetSearch(9), , /Invoice/*s/Buyer/*s/Transaction/*s/Total)
    ...
  ...
```

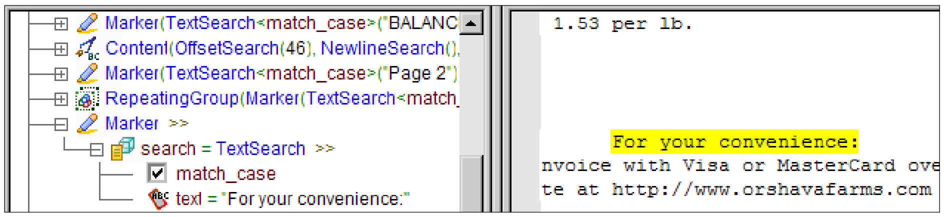
8. Click IntelliScript > Mark Example again. Confirm that the date, reference, product, and total columns are highlighted in all the transactions.



```
Apr 08   22536   cheddar cheese                   45.90     2.30     43.61
                   30 lbs. @ 1.53 per lb.


Apr 21   22798   cream cheese                     28.40     1.42     26.98
                   20 lbs. @ 1.42 per lb.


Apr 29   22903   large eggs                       63.00     3.15     59.85
                   60 dozen @ 1.05 per dozen



Purchases by: Jack
Apr 12   22570   large eggs                       31.50     1.58     29.93
                   30 dozen @ 1.05 per dozen


Apr 18   22734   large eggs                       63.00     3.15     59.85
                   60 dozen @ 1.05 per dozen
```

9. Do you remember how we restricted the search scope of the outer `RepeatingGroup` by defining `Page 2` as a `Marker`? It is a good idea to define a `Marker` at the end of the nested `RepeatingGroup`, too, to make sure that Conversion Agent does not search too far.

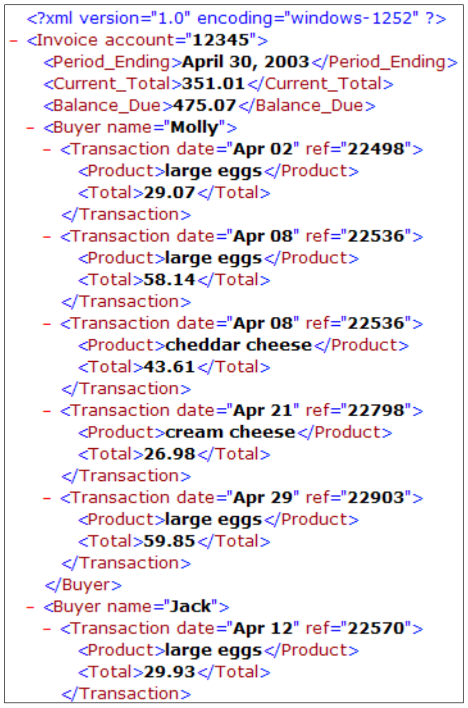
To do this, return to the top-level anchor list, which is the level at which the outer `RepeatingGroup` is defined, and define some of the boilerplate text in the footer of page 2 as a `Marker`. For example, you can define the string `For your convenience:`.



```
Marker(TextSearch<match_case>("BALANC
Content(OffsetSearch(46), NewlineSearch(),
Marker(TextSearch<match_case>("Page 2")
RepeatingGroup(Marker(TextSearch<match_
Marker >>
  search = TextSearch >>
    match_case
    text = "For your convenience:"
```

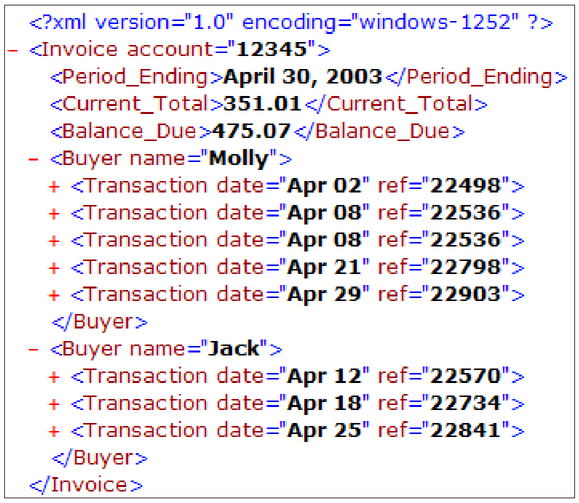
```
 1.53 per lb.



    For your convenience:
nvoice with Visa or MasterCard ove
te at http://www.orshavafarms.com
```

10. Run the parser. You should get the following output:


```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Invoice account="12345">
    <Period_Ending>April 30, 2003</Period_Ending>
    <Current_Total>351.01</Current_Total>
    <Balance_Due>475.07</Balance_Due>
  - <Buyer name="Molly">
    - <Transaction date="Apr 02" ref="22498">
        <Product>large eggs</Product>
        <Total>29.07</Total>
      </Transaction>
    - <Transaction date="Apr 08" ref="22536">
        <Product>large eggs</Product>
        <Total>58.14</Total>
      </Transaction>
    - <Transaction date="Apr 08" ref="22536">
        <Product>cheddar cheese</Product>
        <Total>43.61</Total>
      </Transaction>
    - <Transaction date="Apr 21" ref="22798">
        <Product>cream cheese</Product>
        <Total>26.98</Total>
      </Transaction>
    - <Transaction date="Apr 29" ref="22903">
        <Product>large eggs</Product>
        <Total>59.85</Total>
      </Transaction>
    </Buyer>
  - <Buyer name="Jack">
    - <Transaction date="Apr 12" ref="22570">
        <Product>large eggs</Product>
        <Total>29.93</Total>
      </Transaction>
```

11. Collapse the tree and confirm that the two `Buyer` elements contain 5 and 3 transactions, respectively. This is the number of transactions in the example source.

Note: On Windows XP SP2 or higher, Internet Explorer might display a yellow information bar notifying you that it has blocked active content in the file. If this occurs, clicking the - and + icons fails to collapse or expand the tree. You can unblock the active content, and then collapse the tree.


```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Invoice account="12345">
    <Period_Ending>April 30, 2003</Period_Ending>
    <Current_Total>351.01</Current_Total>
    <Balance_Due>475.07</Balance_Due>
  - <Buyer name="Molly">
    + <Transaction date="Apr 02" ref="22498">
    + <Transaction date="Apr 08" ref="22536">
    + <Transaction date="Apr 08" ref="22536">
    + <Transaction date="Apr 21" ref="22798">
    + <Transaction date="Apr 29" ref="22903">
    </Buyer>
  - <Buyer name="Jack">
    + <Transaction date="Apr 12" ref="22570">
    + <Transaction date="Apr 18" ref="22734">
    + <Transaction date="Apr 25" ref="22841">
    </Buyer>
  </Invoice>
```

As usual, examine the event log for any problems. Remember that events marked with the optional failure icon ( ) or the failure icon ( ) are normal at the end of a `RepeatingGroup`. For more information, see "Testing the Parser" on page 30.

Basic and Advanced Properties

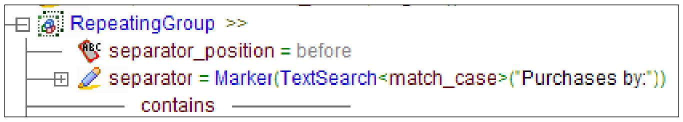
When you configured the separator of the nested `RepeatingGroup`, we told you to click the >> symbol, which displays the advanced properties. When you do this, the >> symbol changes to <<. Click the << symbol to hide the advanced properties.

Many Conversion Agent components have both basic properties and advanced properties. The basic properties are the ones that you need to use most frequently, so they are displayed by default. The advanced properties are needed less often, so Conversion Agent hides them. When you click the >> symbol, they are displayed in gray.

If you assign a non-default value to an advanced property, it turns black. The IntelliScript displays it like a basic property.

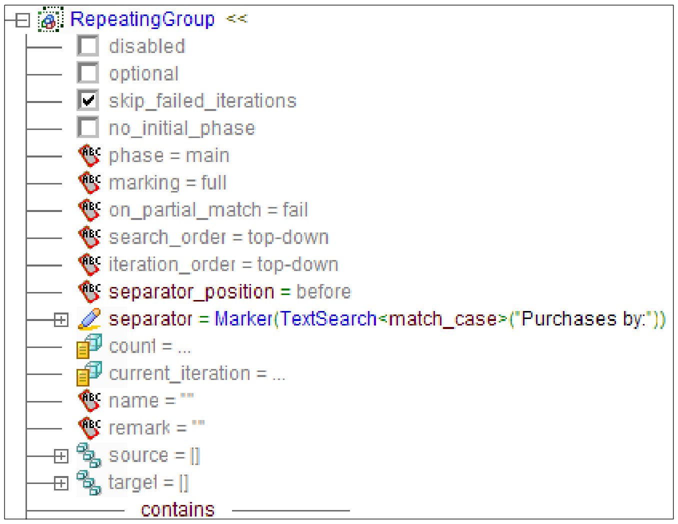
The distinction between the basic and advanced properties is only in the display. The advanced properties are not harder to understand or more difficult to use. Feel free to use them as needed.

Figure 4-1. Basic Properties



```
RepeatingGroup >>
    separator_position = before
    separator = Marker(TextSearch<match_case>("Purchases by:"))
        contains
```

Figure 4-2. Basic and Advanced Properties



```
RepeatingGroup <<
    disabled
    optional
    skip_failed_iterations
    no_initial_phase
    phase = main
    marking = full
    on_partial_match = fail
    search_order = top-down
    iteration_order = top-down
    separator_position = before
    separator = Marker(TextSearch<match_case>("Purchases by:"))
    count = ...
    current_iteration = ...
    name = ""
    remark = ""
    source = []
    target = []
        contains
```

Using an Action to Compute Subtotals

The exercise is complete except for one feature. You need to compute the subtotal for each buyer, and insert the result in the `total` attribute of the `Buyer` elements. The desired output is:

```
<Buyer name="Molly" total="217.64">
```

You can compute the subtotals in the following way:

- Before processing the transactions, initialize the `total` attribute to 0.
- After the parser processes each transaction, add the amount of the transaction to the `total`.

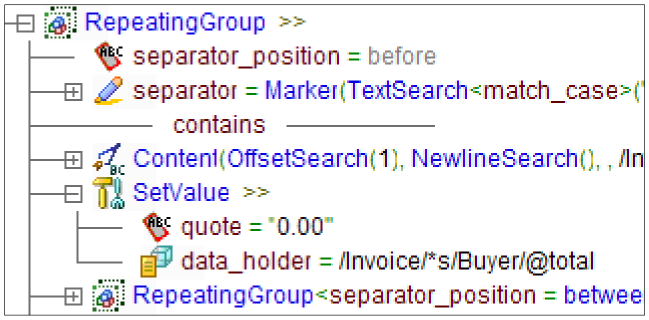
You will use a `SetValue` action to perform the initialization step. You will use a `CalculateValue` action to perform the addition.

To compute the subtotals:

1. Expand the IntelliScript to display the nested `RepeatingGroup`.
2. Right-click the nested `RepeatingGroup`. On the pop-up menu, click Insert. This lets you insert a new component above the `RepeatingGroup`.
3. Insert the `SetValue` action, and set its properties as follows:

```
quote = 0.00
data_holder = /Invoice/*s/Buyer/@total
```

The `SetValue` action assigns the `quote` value to the `data_holder`.



```
RepeatingGroup >>
    separator_position = before
    separator = Marker(TextSearch<match_case>("
    contains
    Content(OffsetSearch(1), NewlineSearch(), , /In
    SetValue >>
        quote = "0.00"
        data_holder = /Invoice/*s/Buyer/@total
    RepeatingGroup<separator_position = betwee
```

4. Now, expand the nested `RepeatingGroup`. At the three dots (...) under the fourth `Content` anchor, enter a `CalculateValue` action.

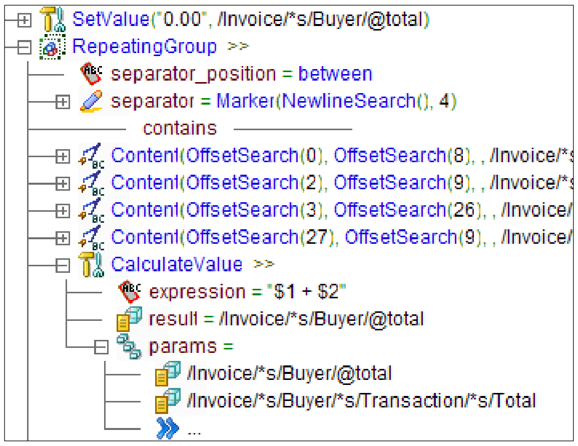
5. Set the properties of the `CalculateValue` action as follows:

```
params =
  /Invoice/*s/Buyer/@total
  /Invoice/*s/Buyer/*s/Transaction/*s/Total
result = /Invoice/*s/Buyer/@total
expression = $1 + $2
```

The `params` property defines the parameters. You can add as many parameter lines as required to the IntelliScript. In this case, there are two parameters: the current `total` attribute of the buyer and the `Total` element of the current transaction.

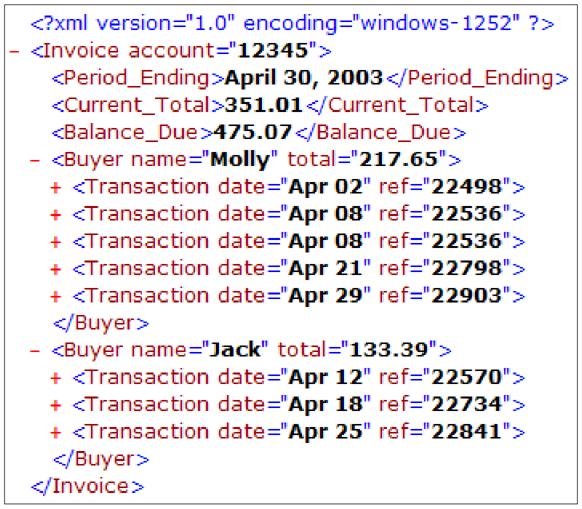
The `expression` property is a JavaScript expression that the action evaluates. `$1` and `$2` are the parameters, which you defined in the `params` property.

The `result` property means that the action assigns the result of the expression as the new value of the `total` attribute.



```
SetValue("0.00", /Invoice/*s/Buyer/@total)
RepeatingGroup >>
    separator_position = between
    separator = Marker(NewlineSearch(), 4)
    contains
    Content(OffsetSearch(0), OffsetSearch(8), , /Invoice/*s
    Content(OffsetSearch(2), OffsetSearch(9), , /Invoice/*s
    Content(OffsetSearch(3), OffsetSearch(26), , /Invoice/
    Content(OffsetSearch(27), OffsetSearch(9), , /Invoice/
    CalculateValue >>
        expression = "$1 + $2"
        result = /Invoice/*s/Buyer/@total
        params =
            /Invoice/*s/Buyer/@total
            /Invoice/*s/Buyer/*s/Transaction/*s/Total
            ...
```

6. Run the parser. In the results file, each `Buyer` element should have a `total` attribute.



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Invoice account="12345">
    <Period_Ending>April 30, 2003</Period_Ending>
    <Current_Total>351.01</Current_Total>
    <Balance_Due>475.07</Balance_Due>
  - <Buyer name="Molly" total="217.65">
    + <Transaction date="Apr 02" ref="22498">
    + <Transaction date="Apr 08" ref="22536">
    + <Transaction date="Apr 08" ref="22536">
    + <Transaction date="Apr 21" ref="22798">
    + <Transaction date="Apr 29" ref="22903">
    </Buyer>
  - <Buyer name="Jack" total="133.39">
    + <Transaction date="Apr 12" ref="22570">
    + <Transaction date="Apr 18" ref="22734">
    + <Transaction date="Apr 25" ref="22841">
    </Buyer>
  </Invoice>
```

Actions

In this exercise, you used the `SetValue` and `CalculateValue` actions to compute the subtotals.

Actions are components that perform operations on data. Conversion Agent provides numerous action components that perform operations such as:

- Computing values
- Concatenating strings
- Testing a condition for continued processing
- Running a secondary parser
- Running a database query

For more information about actions, see the *Conversion Agent Studio User Guide*.

Potential Enhancement: Handling Page Breaks

So far, we have assumed that the repeating groups have a perfectly regular structure. But what happens if the transactions run over to another page? The new page would contain a page header, which would interrupt the repeating structure.

One way to support page headers might be to redefine the separator between the transactions. For example, you might use an `Alternatives` anchor as the separator. This lets you define multiple separators that might occur between the transactions.

We won't give you an exercise to perform this enhancement, but we encourage you to experiment with the features yourself.

Points to Remember

Here are useful hints that arise out of this tutorial:

- To parse a binary format such as a PDF file, you can use a document processor that converts the source to text.
- The positional strategy is useful when the content is laid out at a fixed number of characters, called an offset, from the margins or from `Marker` anchors.

- When you define anchors, think about the possible variations that might occur in source documents. If you define the content positionally, adjust the offsets in case a source document contains longer data than the example source.
- The search scope is the segment of a document where Conversion Agent searches for an anchor. The function of `Marker` anchors is to define the search scope for other anchors.
- Repeating groups can be nested. You can use this feature to parse nested iterative structures.
- Newline characters are useful as markers and separators. You can set the count of newlines, which specifies how many lines to skip.
- In the IntelliScript, you can click the >> symbol next to a component to display its advanced properties. Advanced properties are used less frequently than basic properties, so they are hidden to avoid cluttering the display.
- You can use actions to perform operations on the data, such as computing totals.

CHAPTER 5

Parsing Word and HTML Documents

This chapter includes the following topics:

- Overview, 55
- Requirements Analysis, 56
- Creating the Project, 59
- Defining a Variable, 60
- Parsing the Name and Address, 61
- Parsing the Optional Currency Line, 63
- Parsing the Order Table, 65
- Using Transformers to Modify the Output, 70
- Testing the Parser on Another Source Document, 73
- Points to Remember, 74

Overview

Many real-life documents have less structure than the ones we have described in the previous chapters. The documents might have:

- Few labels and delimiters to help a parser identify the data fields
- Wrapped lines and flexible layout, making positional parsing difficult
- Repeated keywords, making it hard to define unambiguous `Marker` anchors

Microsoft Word documents are typical examples of these phenomena. Word documents are usually created by hand, although they can also be generated programmatically. Even if the authors create the documents from a fixed template, they might vary the layout from one document to another.

In such cases, you need to use the available markup or patterns to parse the documents. For example, you can use paragraphs, tables, and italics to identify the data fields.

For a Microsoft Word document, you can expose the formatting markup by saving the document as HTML. A parser can use the HTML tags, which are essentially format labels, to interpret the document structure.

In this chapter, you will construct a parser for a Word document. You will learn techniques such as:

- Using the `WordToHtml` document processor, which converts a Word document to an HTML document.

- Taking advantage of the opening and closing tag structure to parse an HTML document.
- Identifying anchors unambiguously, in situations where the same text is repeated frequently throughout the document.
- Using transformers to modify the output of anchors.
- Using variables to store retrieved data for later use.
- Defining a global component, which you configure once and use multiple times in a project.
- Testing a parser on source documents other than the example source.

Scope of the Exercise

This exercise is the longest and most complex in this book. It is typical of real-life parsing scenarios. You will create a full-featured parser that can be used in a production application.

We suggest that you skim the chapter from beginning to end to understand how the parser is intended to work, and then try the exercise. Feel free to experiment with the parser configuration. There is more than one possible solution to this exercise.

For example, you can modify the anchor configuration to support numerous variations of the source-document structure. You can modify the XSD schema to restrict the data types of the elements, ensuring that the parser retrieves exactly the correct data from a complex document. We have omitted most of these refinements because we want to keep the scope of the exercise within bounds.

Prerequisites

In order to do this exercise, you need Microsoft Word 97 or higher on the same computer as Conversion Agent Studio.

If you do not have Word on the same computer, you can open the source document in Word on another computer and save it as HTML. You can then transfer the HTML document back to the Conversion Agent computer and parse it. The rest of the exercise is unchanged, except that you do not need to use the `WordToHtml` processor because the document is already HTML.

We presume that you are familiar with HTML code. You can get an introduction at http://www.w3.org or from any book on web-site development.

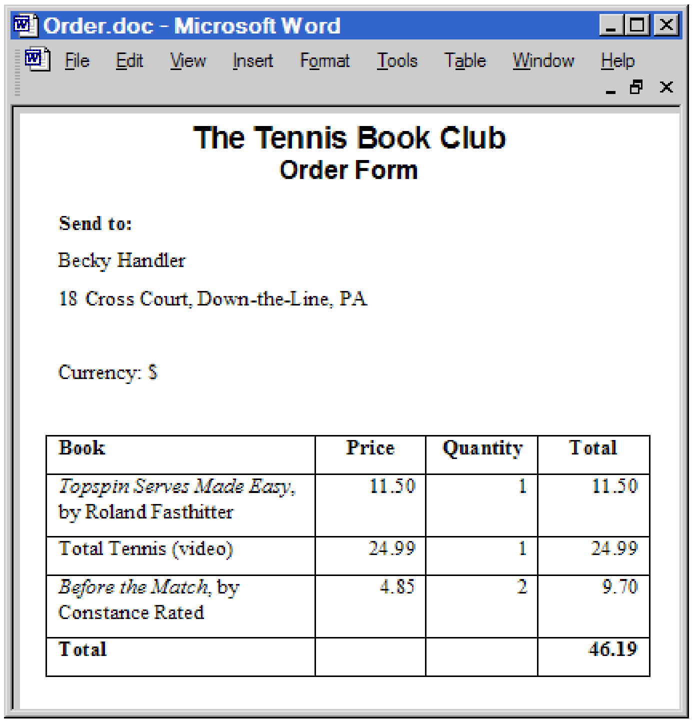
Requirements Analysis

Before you start configuring a parser, it is always a good idea to plan what the parser must do. In this section, we examine the source document structure, the required XML output, and the nature of the parsing problem.

Source Document

You can view the example source in Microsoft Word. The document is stored in:

```
tutorials\Exercises\Files_for_tutorial_4\Order.doc
```


Order.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help

The Tennis Book Club
Order Form

Send to:

Becky Handler

18 Cross Court, Down-the-Line, PA

Currency: $

Book	Price	Quantity	Total
Topspin Serves Made Easy, by Roland Fasthitter	11.50	1	11.50
Total Tennis (video)	24.99	1	24.99
Before the Match, by Constance Rated	4.85	2	9.70
Total			**46.19**

The document is an order form used by a fictitious organization called The Tennis Book Club. The form contains:

- The name of the purchaser
- A one-line address
- An optional line specifying the currency such as `$` or `£`
- A table of the items ordered

The format of the items in the table is not uniform. Although the first column heading is `Books`, some of the possible items in this column, such as a video cassette, are not books. These items might be formatted in a different font from the books.

In some of the table cells, the text wraps to a second line. Positional text parsing, using the techniques that you learned in the previous chapter, would be difficult for these cells.

The table cells do not contain any identifying labels. The parser needs to interpret the cells according to their location in the table.

The word `Total` appears three times in the table. In two of the instances, the word is in a column or row heading; we can assume that this is the case in all similar source documents. In the third instance, `Total` appears by chance in the title of the video cassette. This is significant because we want to parse the total-price row of the table, and we need an unambiguous identifier for the row.

As a further complication, the `Currency` line is optional. Some of the source documents that we plan to parse are missing this line.

XML Output

For the purpose of this exercise, we assume that the output XML format is as follows:

```
<Order>
  <To>Becky Handler</To>
  <Address>18 Cross Court, Down-the-Line, PA</Address>
  <Books>
    <Book>
      <Title>Topspin Serves Made Easy, by Roland Fasthitter</Title>
      <Price>$11.50</Price>
```

```
          <Quantity>1</Quantity>
          <Total>$11.50</Total>
        </Book>
        <!-- Additional Book elements -->
      </Books>
      <Total>$46.19</Total>
    </Order>
```

Notice that the prices and totals include the currency symbol ($). In the source document, the currency is stated on the optional currency line, and not in the table.

The Parsing Problem

Processor Selection

Conversion Agent provides several document processors that can convert a Word document to a format suitable for parsing. Among these are:

- `WordToHtml`
- `WordToRtf`
- `WordToTxt`
- `WordToXml`

We choose to use the `WordToHtml` processor because:

- The HTML code is easy to read, making it easy to configure the parser.
- The HTML tags expose a wealth of formatting information, that can help identify the data to retrieve.

The `WordToXml` and `WordToRtf` processors might also be good choices, but their output is more difficult to read than that of the `WordToHtml` processor.

The `WordToTxt` processor produces plain-text output. Because the text lacks formatting information, it would probably be much harder to parse reliably than the HTML.

HTML Structure

The `WordToHtml` processor uses the Save As HTML feature of Microsoft Word to generate the HTML. The HTML is exceedingly verbose. It can run to many pages, even though the original Word document is less than one page. This occurs because Word saves the complete document features, including the style and format definitions. Much of this information has no significant effect on the appearance of the HTML in a web browser, but Word uses the information if you later re-import the HTML to Word. Most of this code is in the `<header>` element of the HTML document.

If you wish, you can save the example source document as HTML manually in Word. You can view the resulting HTML file in a browser, or you can open the file in Notepad and examine the HTML code.

In a broad outline, the code looks like this:

```
<html>\
   <header>
    <!-- Very long header storing Word style definitions ... -->
  </header>
  <body>
    <h1>The Tennis Book Club<br>Order Form</h1>
    <p>Send to:</p>
    <p>Becky Handler</p>
    <p>18 Cross Court, Down-the-Line, PA</p>
    <table>
      <tr>
        <!-- Table column headers -->
      </tr>
      <tr>
        <td><i>Topspin Serves Made Easy</i>, by Roland Fasthitter</td>
        <td>$11.50</td>
        <td>1</td>
```

```
          <td>$11.50</td>
        </tr>
        <!-- Additional tr elements containing the other table rows -->
      </table>
    </body>
  </html>
```

You will use this basic HTML tag structure to parse the document.

The actual code might be considerably more complex than shown above. The following is a typical example, which is the first `<td>` element of the `Topspin Serves` row. The code might vary somewhat on your computer, depending on your Word version and configuration.

```
<td width=168 valign=top style='width:125.9pt;border:solid windowtext
1.0pt;border-top:none;mso-border-top-alt:solid windowtext .5pt;mso-
border-alt:solid windowtext .5pt;padding:0cm 5.4pt 0cm 5.4pt'><p
class=MsoBodyText><i style='mso-bidi-font-style:normal'>Topspin Serves
Made Easy</i>, by Roland <span class=SpellE>Fasthitter</span></p></td>
```

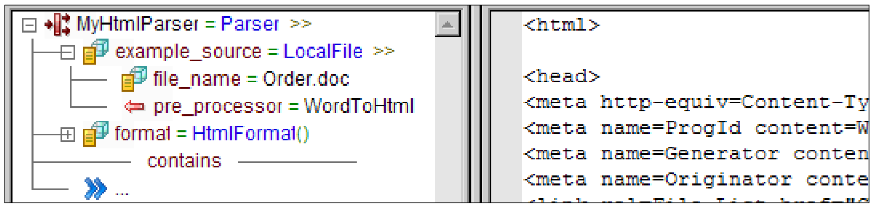
As you begin to work with the Word-generated HTML, we trust that you will quickly learn to find the important structural tags, such as the `<td>...</td>` element. You can ignore the irrelevant code such as the lengthy attributes and `<span>` tags.

Creating the Project

To create the project:

1. To start the exercise, create a project called `Tutorial_4`.
2. In the New Project wizard, select the following options:
   - Name the parser `MyHtmlParser`.
   - Name the script file `Html_ScriptFile`.
   - Select the schema `TennisBookClub.xsd`, which is in the `tutorials\Exercises\Files_For_Tutorial_4` folder.
   - Specify that the example source is a file on a local computer.
   - Browse to the example source file, which is `Order.doc`.
   - Select the `Microsoft Word` encoding.
   - Select the document processor `Microsoft Word To HTML`.
   - Select the `HTML format`. This format component is pre-configured with the appropriate delimiters and other components for parsing HTML code.
   - Click Finish to create the project.

   The resulting IntelliScript has the following appearance:



```
MyHtmlParser = Parser >>
    example_source = LocalFile >>
        file_name = Order.doc
        pre_processor = WordToHtml
    format = HtmlFormat()
        contains
    ...
```

```
<html>

<head>
<meta http-equiv=Content-Ty
<meta name=ProgId content=W
<meta name=Generator conten
<meta name=Originator conte
```

Note: When the Studio opens the example source, it might display a Microsoft Internet Explorer prompt to open or save the file. This occurs because the Studio uses Internet Explorer to display a browser view of the file. The behavior depends on your Internet Explorer security settings. Click Open.

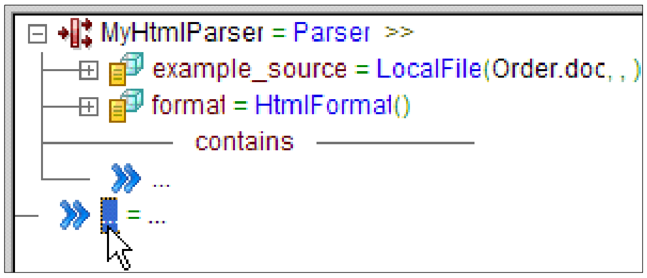
Defining a Variable

Do you remember the `Currency` line, near the top of the example source? We need to retrieve the currency symbol, but we do not want to map it to an output element or attribute. Instead, we want to store the currency symbol temporarily, and prefix it to the prices in the output.

To do this, we will define a temporary data holder called a variable. You can use a variable in exactly the same way as an element or attribute data holder, but it isn't included in the XML output.

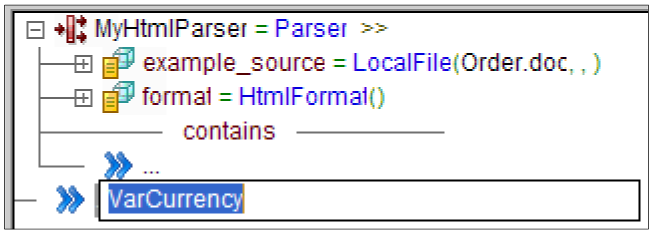
To define the variable:

1. At the bottom of the IntelliScript, at the global level and not nested within the `Parser` component, select the three dots (...) and press Enter.



```
MyHtmlParser = Parser >>
  example_source = LocalFile(Order.doc, , )
  format = HtmlFormat()
  contains
  ...
 = ...
```

2. Type the variable name, `varCurrency`, and press Enter.

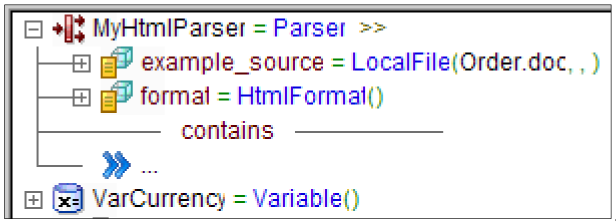


```
MyHtmlParser = Parser >>
  example_source = LocalFile(Order.doc, , )
  format = HtmlFormat()
  contains
  ...
VarCurrency
```

3. At the three dots on the right side of the equals sign (=), press Enter and select `Variable`.

   The variable appears in the IntelliScript, in the Schema view, and in the Component view.

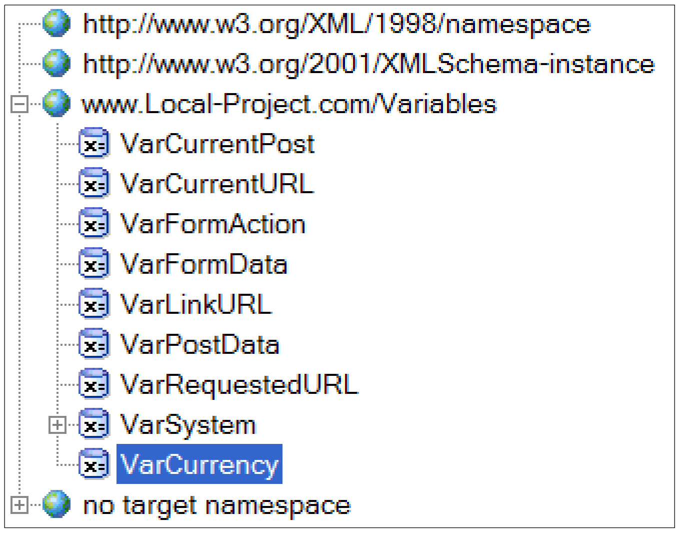
Figure 5-1. Variable Definition in the IntelliScript



```
MyHtmlParser = Parser >>
  example_source = LocalFile(Order.doc, , )
  format = HtmlFormat()
  contains
  ...
VarCurrency = Variable()
```

.

Figure 5-2. Variable Displayed in the Schema View


http://www.w3.org/XML/1998/namespace
http://www.w3.org/2001/XMLSchema-instance
www.Local-Project.com/Variables
VarCurrentPost
VarCurrentURL
VarFormAction
VarFormData
VarLinkURL
VarPostData
VarRequestedURL
VarSystem
VarCurrency
no target namespace

Notice the location of the variable in the Schema view, nested within the `www.Local-Project.com/Variables` namespace. Later, you will access the variable in the Schema view.

Parsing the Name and Address

You will now define the anchors that parse the name and address, which are located at the beginning of the example source.

To parse the name and address:

1. In the example pane, scroll past the header to the `body` tag, about 80% of the way down.

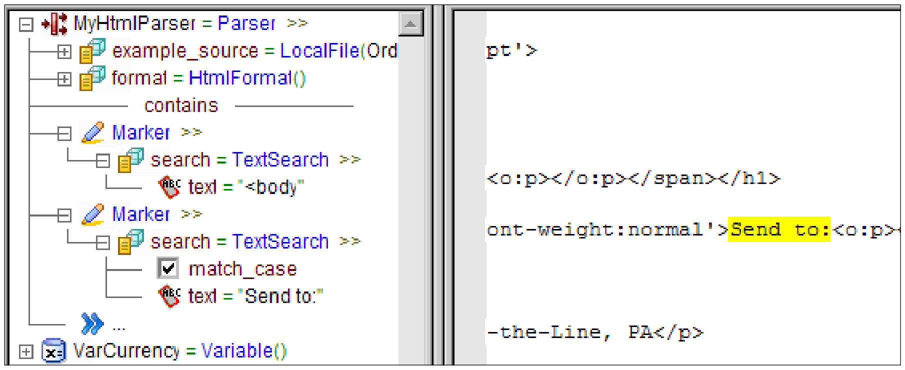
   To scroll quickly, you can right-click in the example pane, select the Find option, and find the string `<body`, with a starting < symbol but without the ending > symbol.

2. Define `<body` as a `Marker` anchor. This causes the parser to skip the header completely, just as you did when you scrolled through the document.

   Do not select the `match_case` property for this anchor, or for any other HTML tags that you define as anchors in this exercise. HTML is not case sensitive. Word 2000 and higher generate HTML tags in lower case, but Word 97 generates tags in upper case. If you select match case, the parser will not support Word 97.

3. Scroll down a few paragraphs, and define the string `Send to:` as a `Marker` anchor. The string signals the beginning of the content that you need to retrieve.

Here, you can select the match case option. This can be helpful to ensure that you match only the indicated spelling.



```
MyHtmlParser = Parser >>
  example_source = LocalFile(Ord
  format = HtmlFormat()
  contains
  Marker >>
    search = TextSearch >>
      text = "<body"
  Marker >>
    search = TextSearch >>
      match_case
      text = "Send to:"
  ...
VarCurrency = Variable()
```

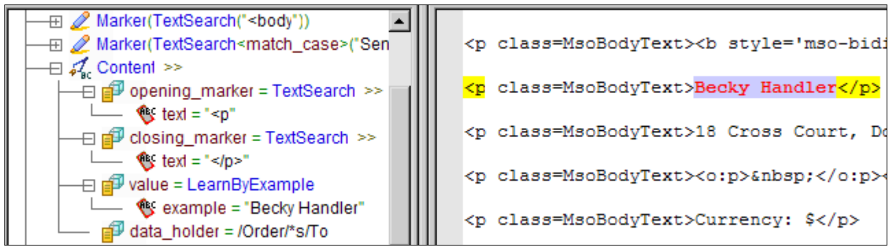
```
pt'>

<o:p></o:p></span></h1>

ont-weight:normal'>Send to:<o:p>

-the-Line, PA</p>
```

4. Now, you are ready to define the first `Content` anchor. You will configure the anchor to retrieve the data between specified starting and ending strings. A `Content` anchor that is defined in this way is similar to a `Marker Content Marker` sequence, which you could also use, if you prefer.

To do this, right-click the string `Becky Handler` and insert a `Content` anchor. Make the following assignments in the IntelliScript:

Property	Value
`opening_marker`	Select `TextSearch` and type the string `<p`.
`closing_marker`	Select `TextSearch` and type the string `</p>`.
`data_holder`	Map the anchor to `/Order/*s/To`.

When you finish these assignments, the example pane highlights `<p` and `</p>` as markers. It highlights `Becky Handler` as content.



```
Marker(TextSearch("<body"))
Marker(TextSearch<match_case>("Sen
Content >>
  opening_marker = TextSearch >>
    text = "<p"
  closing_marker = TextSearch >>
    text = "</p>"
  value = LearnByExample
    example = "Becky Handler"
  data_holder = /Order/*s/To
```

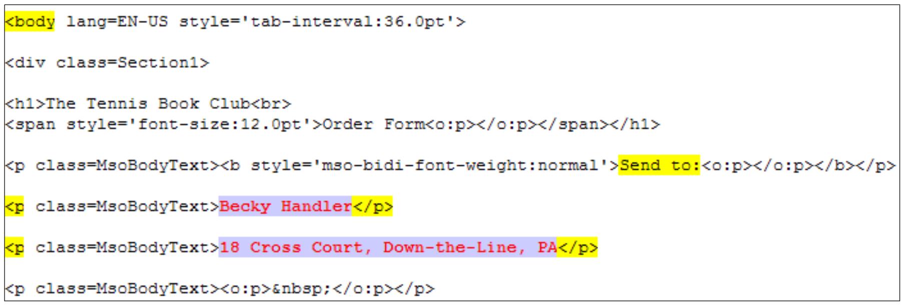
```
<p class=MsoBodyText><b style='mso-bidi
<p class=MsoBodyText>Becky Handler</p>
<p class=MsoBodyText>18 Cross Court, Do
<p class=MsoBodyText><o:p>&nbsp;</o:p>
<p class=MsoBodyText>Currency: $</p>
```

Notice that only the string `Becky Handler` is highlighted in blue. The rest of the HTML code between the `opening_marker` and the `closing_marker`, such as

```
class=MsoBodyText>
```

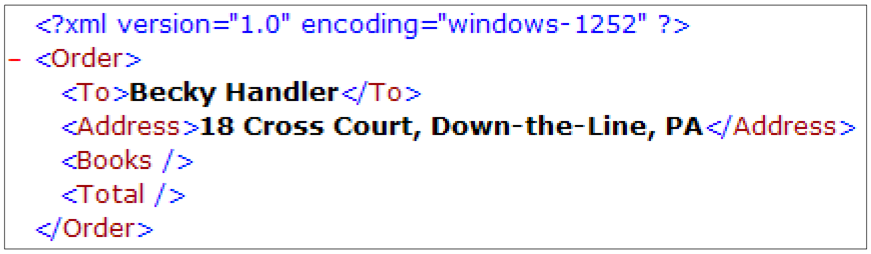
is not highlighted. This is because the `HtmlFormat` recognizes `>` as a delimiter character. It learns from the example source that the desired content is located after the `>` delimiter.

5. Move to the next paragraph of the source, and use the same technique, defining the `opening_marker` and the `closing_marker`, to map `18 Cross Court, Down-the-Line, PA` to the `/Order/*s/Address` data holder.



```
<body lang=EN-US style='tab-interval:36.0pt'>

<div class=Section1>

<h1>The Tennis Book Club<br>
<span style='font-size:12.0pt'>Order Form<o:p></o:p></span></h1>

<p class=MsoBodyText><b style='mso-bidi-font-weight:normal'>Send to:<o:p></o:p></b></p>

<p class=MsoBodyText>Becky Handler</p>

<p class=MsoBodyText>18 Cross Court, Down-the-Line, PA</p>

<p class=MsoBodyText><o:p>&nbsp;</o:p></p>
```

6. Run the parser, and confirm that you get the following result:



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Order>
    <To>Becky Handler</To>
    <Address>18 Cross Court, Down-the-Line, PA</Address>
    <Books />
    <Total />
  </Order>
```

Why the Output Contains Empty Elements

Notice that the parser inserts empty `Books` and `Total` elements. That is because the XSD schema defines `Books` and `Total` as required elements. The parser inserts the element to ensure that the XML is valid according to the schema. If necessary you can change this behavior in the project properties. The elements are empty because we have not yet parsed the table of books.

Parsing the Optional Currency Line

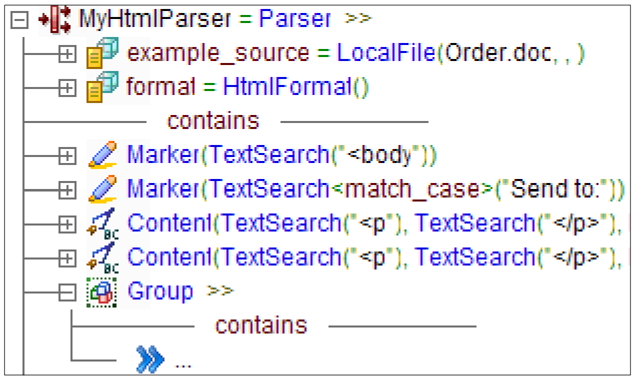
If the source document contains a `Currency` line, the parser should process the line and store the result in the `varCurrency` variable, which you created above. If the `Currency` line is missing, the parser should ignore the omission and continue parsing the rest of the document.

You can implement this in the following way:

1. Define a `Group` anchor. The purpose of this anchor is to bind a set of nested anchors together, for processing as a unit.
2. Within the group, nest `Marker` and `Content` anchors that retrieve the currency.
3. Select the `optional` property of the `Group`. This means that if the `Group` fails due to the `Marker` and/or `Content` being missing, the parser continues running.

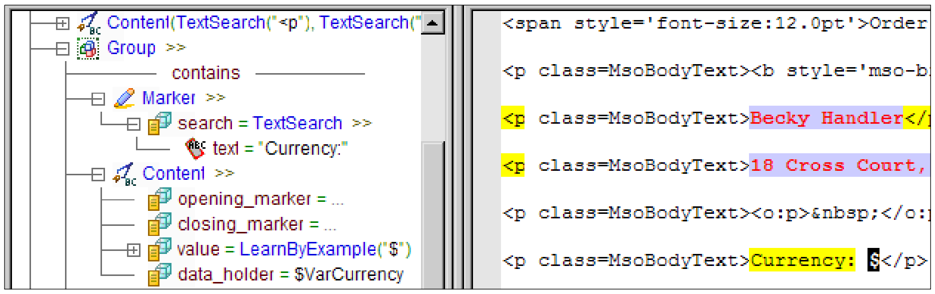
To parse the optional currency line:

1. Following the anchors that you have already defined, insert a `Group` anchor.



```
MyHtmlParser = Parser >>
  example_source = LocalFile(Order.doc, , )
  format = HtmlFormat()
  contains
  Marker(TextSearch("<body"))
  Marker(TextSearch<match_case>("Send to:"))
  Content(TextSearch("<p"), TextSearch("</p>"),
  Content(TextSearch("<p"), TextSearch("</p>"),
  Group >>
    contains
    ...
```

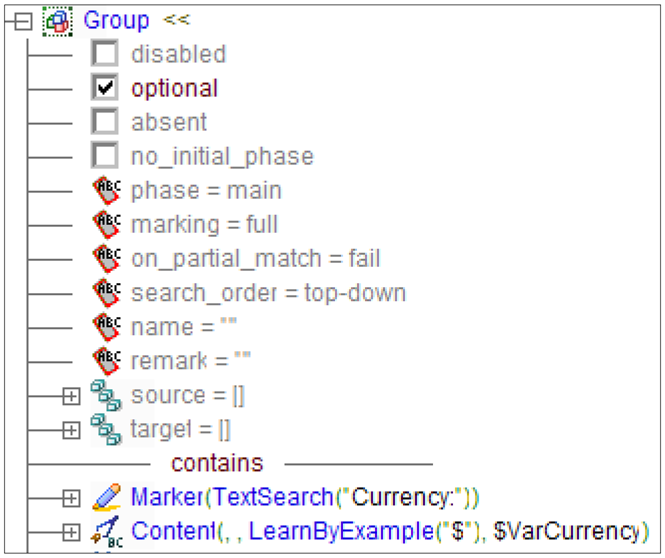
2. Select the three-dots that are nested within the `Group`. Define the text `Currency;` as a `Marker` anchor.
3. Continuing within the `Group`, define `$` as a `Content` anchor. Map the anchor to the `VarCurrency` variable, which you defined above.



```
Content(TextSearch("<p"), TextSearch("
Group >>
  contains
  Marker >>
    search = TextSearch >>
      text = "Currency:"
  Content >>
    opening_marker = ...
    closing_marker = ...
    value = LearnByExample("$")
    data_holder = $VarCurrency
```

```
<span style='font-size:12.0pt'>Order
<p class=MsoBodyText><b style='mso-b
<p class=MsoBodyText>Becky Handler</
<p class=MsoBodyText>18 Cross Court,
<p class=MsoBodyText><o:p>&nbsp;</o:
<p class=MsoBodyText>Currency: $</p>
```

4. Click the >> symbol to display the advanced properties of the `Group`, and select the `optional` property. Be sure you select the `optional` property of the `Group`, and not the `optional` property of the `Marker` or `Content`.



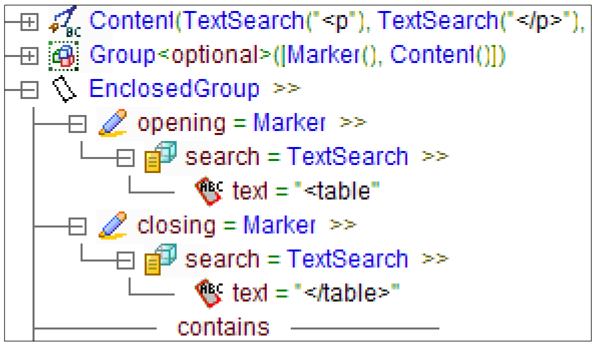
```
Group <<
  [ ] disabled
  [✓] optional
  [ ] absent
  [ ] no_initial_phase
  phase = main
  marking = full
  on_partial_match = fail
  search_order = top-down
  name = ""
  remark = ""
  source = []
  target = []
  contains
  Marker(TextSearch("Currency:"))
  Content(, , LearnByExample("$"), $VarCurrency)
```

Parsing the Order Table

As you have seen in the preceding exercises, you can parse the order table by using a `RepeatingGroup`. In this exercise, you will nest the `RepeatingGroup` in an `EnclosedGroup` anchor. The `EnclosedGroup` is useful for parsing HTML code because it recognizes the opening-and-closing tag structure, which is characteristic of the code. Specifically, you will use the `EnclosedGroup` to recognize the `<table` and `</table>` tags that surround an HTML table.

To parse the table:

1. Select the three dots at the end of the top-level anchor sequence, that is, the level where the `Group` is defined, and not the nested level within the `Group`.
2. By editing the IntelliScript, insert an `EnclosedGroup` anchor.
3. Under the `opening` property of the `EnclosedGroup`, assign `text = <table`. Under the `closing` property, assign `text = </table>`. You can type the strings, or you can drag them from the example pane.



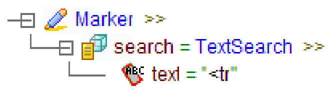
```
Content(TextSearch("<p"), TextSearch("</p>"),
Group<optional>([Marker(), Content()])
EnclosedGroup >>
    opening = Marker >>
        search = TextSearch >>
            text = "<table"
    closing = Marker >>
        search = TextSearch >>
            text = "</table>"
    contains
```

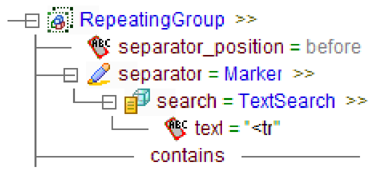
The example pane highlights the `<table` and `</table>` tags in the HTML code.

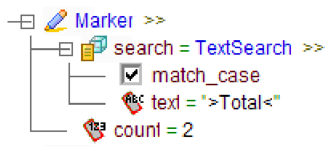
4. Now, start adding anchors within the `EnclosedGroup`. If you did the exercises in the preceding chapters, you should be able to work out a solution. We encourage you to try it yourself before you continue reading.

   One solution is to configure the anchors by editing the IntelliScript, rather than selecting and right-clicking in the example source. The HTML code of the example source is long and confusing, and several of the anchors have non-default properties, which you can configure only in the IntelliScript.

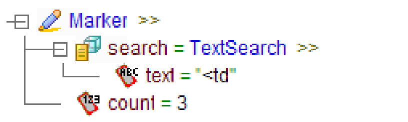
Within the EnclosedGroup Anchor, Add	Configuration	Explanation

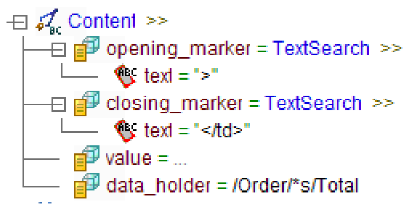
`Marker`	 Marker >>
	  search = TextSearch >>
	    text = "<tr"
		Advances to the first row of the table, which you do not need to retrieve.

`RepeatingGroup`	 RepeatingGroup >>
	  separator_position = before
	  separator = Marker >>
	    search = TextSearch >>
	      text = "<tr"
	  contains
		Starts a repeating group at the second row of the table.
		See below for the anchors to insert within the `RepeatingGroup`.

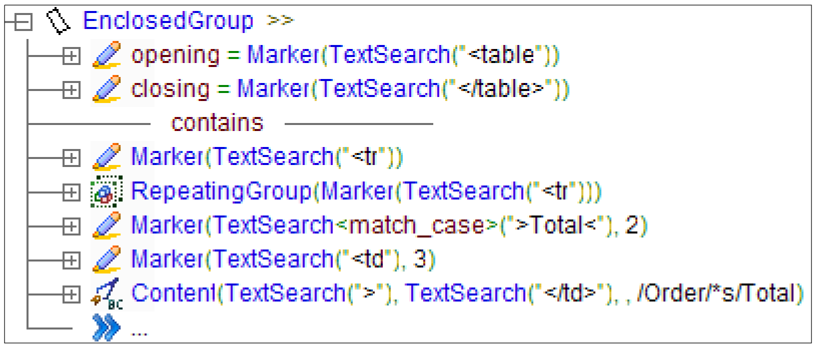
`Marker`	 Marker >>
	  search = TextSearch >>
	    ☑ match_case
	    text = ">Total<"
	  count = 2
		Terminates the repeating group and moves to the last row of the table.
		To assign `count=2`, display the advanced properties. For more information, see “Using Count to Resolve Ambiguities” on page 70.

Within the EnclosedGroup Anchor, Add	Configuration	Explanation

`Marker`	 Marker >>
	search = TextSearch >>
	text = "<td"
	count = 3
		Advances to the fourth cell in the last row.
		To assign `count=3`, display the advanced properties.

`Content`	 Content >>
	opening_marker = TextSearch >>
	text = ">"
	closing_marker = TextSearch >>
	text = "</td>"
	value = ...
	data_holder = /Order/*s/Total
		Retrieves the content of the fourth cell in the last row.

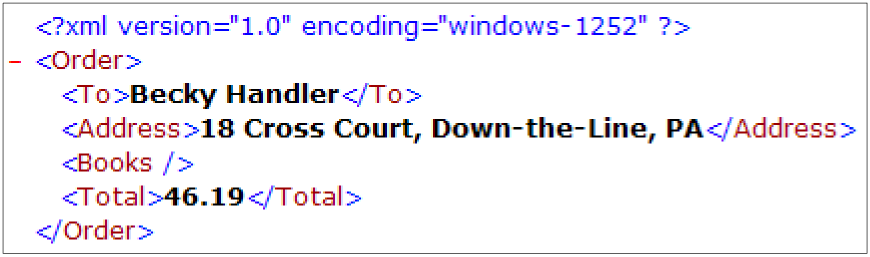
At this point, the `EnclosedGroup` looks like this:



```
EnclosedGroup >>
  opening = Marker(TextSearch("<table"))
  closing = Marker(TextSearch("</table>"))
  contains
  Marker(TextSearch("<tr"))
  RepeatingGroup(Marker(TextSearch("<tr")))
  Marker(TextSearch<match_case>(">Total<"), 2)
  Marker(TextSearch("<td"), 3)
  Content(TextSearch(">"), TextSearch("</td>"), , /Order/*s/Total)
  ...
```

5. Try the Mark Example command. If you scroll through the example source, you can view the color coding for the above anchors.

6. Run the parser. The results look like this:

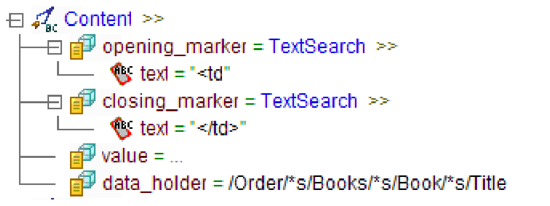


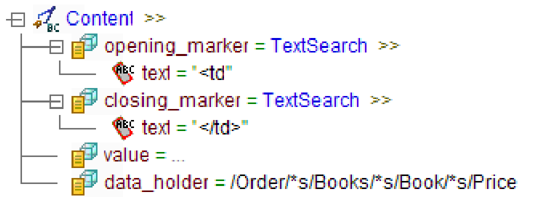
```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Order>
    <To>Becky Handler</To>
    <Address>18 Cross Court, Down-the-Line, PA</Address>
    <Books />
    <Total>46.19</Total>
  </Order>
```

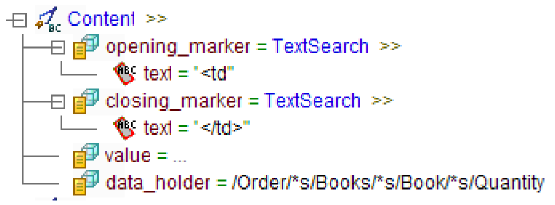
7. Now, expand the `RepeatingGroup` and insert the following anchors within it.

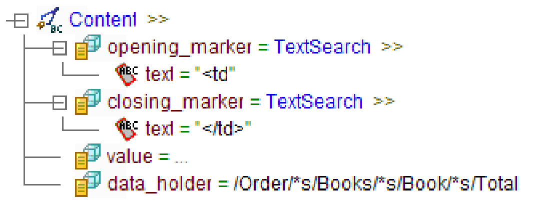
Notice that the four `Content` anchors have very similar configurations. Optionally, you can save time by copying one of the `Content` anchors, and editing the copies.

Within the RepeatingGroup Anchor, Add	Configuration	Explanation

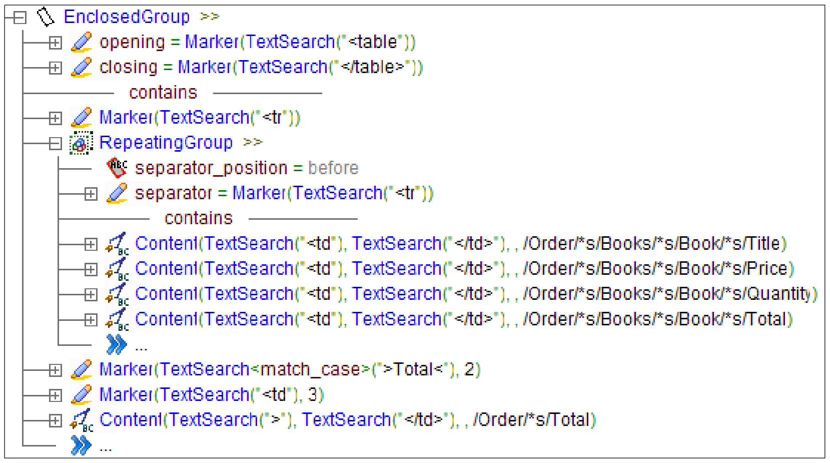
`Content`	 Content >>
	opening_marker = TextSearch >>
	text = "<td"
	closing_marker = TextSearch >>
	text = "</td>"
	value = ...
	data_holder = /Order/*s/Books/*s/Book/*s/Title
		Retrieves the content of the first cell in a row.

`Content`	 Content >>
	opening_marker = TextSearch >>
	text = "<td"
	closing_marker = TextSearch >>
	text = "</td>"
	value = ...
	data_holder = /Order/*s/Books/*s/Book/*s/Price
		Retrieves the content of the second cell.

`Content`	 Content >>
	opening_marker = TextSearch >>
	text = "<td"
	closing_marker = TextSearch >>
	text = "</td>"
	value = ...
	data_holder = /Order/*s/Books/*s/Book/*s/Quantity
		Retrieves the content of the third cell.

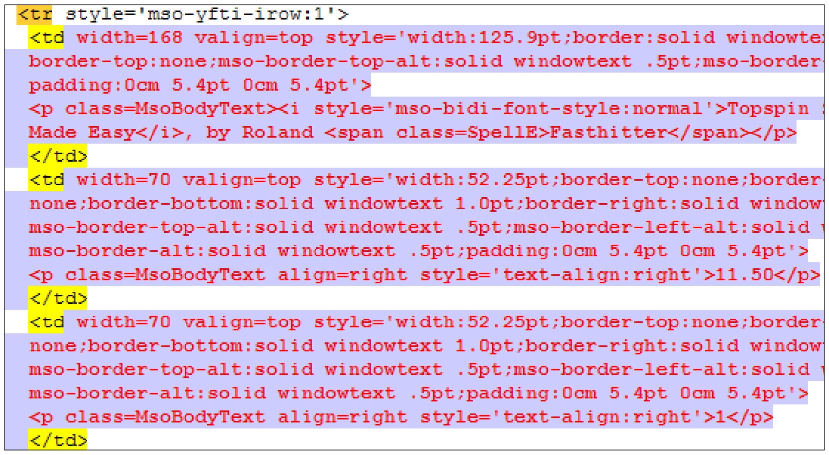
`Content`	 Content >>
	opening_marker = TextSearch >>
	text = "<td"
	closing_marker = TextSearch >>
	text = "</td>"
	value = ...
	data_holder = /Order/*s/Books/*s/Book/*s/Total
		Retrieves the content of the fourth cell.

When you have finished, the `EnclosedGroup` and `RepeatingGroup` should look like this:



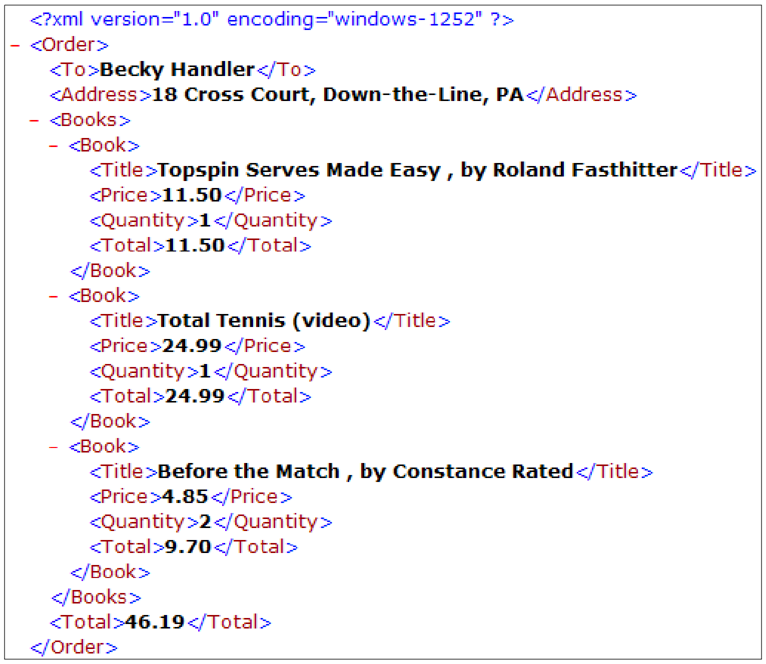
```
EnclosedGroup >>
  opening = Marker(TextSearch("<table"))
  closing = Marker(TextSearch("</table>"))
  contains
  Marker(TextSearch("<tr"))
  RepeatingGroup >>
    separator_position = before
    separator = Marker(TextSearch("<tr"))
    contains
    Content(TextSearch("<td"), TextSearch("</td>"), , /Order/*s/Books/*s/Book/*s/Title)
    Content(TextSearch("<td"), TextSearch("</td>"), , /Order/*s/Books/*s/Book/*s/Price)
    Content(TextSearch("<td"), TextSearch("</td>"), , /Order/*s/Books/*s/Book/*s/Quantity)
    Content(TextSearch("<td"), TextSearch("</td>"), , /Order/*s/Books/*s/Book/*s/Total)
    ...
  Marker(TextSearch<match_case>(">Total<"), 2)
  Marker(TextSearch("<td"), 3)
  Content(TextSearch(">"), TextSearch("</td>"), , /Order/*s/Total)
  ...
```

8. Run the Mark Example command, and check the color-coding again.



```
<tr style='mso-yfti-irow:1'>
 <td width=168 valign=top style='width:125.9pt;border:solid windowte
 border-top:none;mso-border-top-alt:solid windowtext .5pt;mso-border
 padding:0cm 5.4pt 0cm 5.4pt'>
 <p class=MsoBodyText><i style='mso-bidi-font-style:normal'>Topspin
 Made Easy</i>, by Roland <span class=SpellE>Fasthitter</span></p>
 </td>
 <td width=70 valign=top style='width:52.25pt;border-top:none;border
 none;border-bottom:solid windowtext 1.0pt;border-right:solid window
 mso-border-top-alt:solid windowtext .5pt;mso-border-left-alt:solid
 mso-border-alt:solid windowtext .5pt;padding:0cm 5.4pt 0cm 5.4pt'>
 <p class=MsoBodyText align=right style='text-align:right'>11.50</p>
 </td>
 <td width=70 valign=top style='width:52.25pt;border-top:none;border
 none;border-bottom:solid windowtext 1.0pt;border-right:solid window
 mso-border-top-alt:solid windowtext .5pt;mso-border-left-alt:solid
 mso-border-alt:solid windowtext .5pt;padding:0cm 5.4pt 0cm 5.4pt'>
 <p class=MsoBodyText align=right style='text-align:right'>1</p>
 </td>
```

9. Run the parser. The result should look like this:



```
 <?xml version="1.0" encoding="windows-1252" ?>
- <Order>
   <To>Becky Handler</To>
   <Address>18 Cross Court, Down-the-Line, PA</Address>
 - <Books>
   - <Book>
       <Title>Topspin Serves Made Easy , by Roland Fasthitter</Title>
       <Price>11.50</Price>
       <Quantity>1</Quantity>
       <Total>11.50</Total>
     </Book>
   - <Book>
       <Title>Total Tennis (video)</Title>
       <Price>24.99</Price>
       <Quantity>1</Quantity>
       <Total>24.99</Total>
     </Book>
   - <Book>
       <Title>Before the Match , by Constance Rated</Title>
       <Price>4.85</Price>
       <Quantity>2</Quantity>
       <Total>9.70</Total>
     </Book>
   </Books>
   <Total>46.19</Total>
 </Order>
```

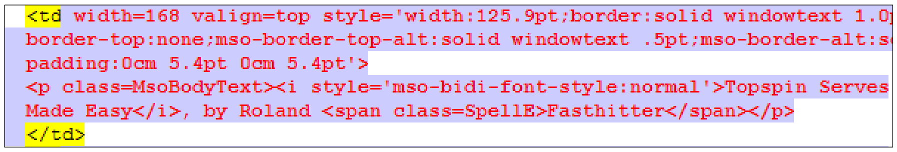
Why the Output does not Contain HTML Code

Notice that the parser removed the HTML code from the retrieved text.

For example, the first `Content` anchor within the `RepeatingGroup` is configured to retrieve all the text between the `<td` and `</td>`. You might have expected it to retrieve the text:

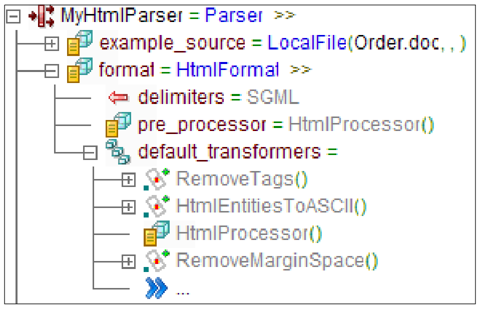
```
 width=168 valign=top style='width:125.9pt;border:solid windowtext 1.0pt;border-
top:none;mso-border-top-alt:solid windowtext .5pt;mso-border-alt:solid windowtext
.5pt;padding:0cm 5.4pt 0cm 5.4pt'><p class=MsoBodyText><i style='mso-bidi-font-
style:normal'>Topspin Serves Made Easy</i>, by Roland <span class=SpellE>Fasthitter</
span></p>
```

In fact, this is exactly the color-coding of the IntelliScript:



```
<td width=168 valign=top style='width:125.9pt;border:solid windowtext 1.0
border-top:none;mso-border-top-alt:solid windowtext .5pt;mso-border-alt:s
padding:0cm 5.4pt 0cm 5.4pt'>
<p class=MsoBodyText><i style='mso-bidi-font-style:normal'>Topspin Serves
Made Easy</i>, by Roland <span class=SpellE>Fasthitter</span></p>
</td>
```

The answer is that the Content anchor retrieves the entire text, including the HTML code. However, the HtmlFormat component of the parser is configured with a set of default transformers, which modify the output of the anchor.



```
MyHtmlParser = Parser >>
  example_source = LocalFile(Order.doc, , )
  format = HtmlFormat >>
    delimiters = SGML
    pre_processor = HtmlProcessor()
    default_transformers =
      RemoveTags()
      HtmlEntitiesToASCII()
      HtmlProcessor()
      RemoveMarginSpace()
      ...
```

One of the default transformers is RemoveTags, which removes the HTML code.

Transformers

A transformer is a component that modifies the output of an anchor. Some common uses of transformers are to add or replace strings in the output. You will use transformers for these purposes in the next stage of the exercise.

The format component of a parser is typically configured with a set of default transformers, which the parser applies to the output of all anchors. The purpose of the default transformers is to clean up the output.

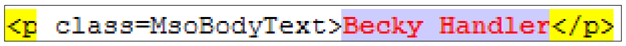
The HtmlFormat is configured with the following default transformers:

- RemoveTags, which removes HTML code from the output.
- HtmlEntitiesToASCII, which converts HTML entities, such as &gt; or &amp; to plain-text symbols, > or &.
- HtmlProcessor, which converts multiple whitespace characters, spaces, tabs, or line breaks, to single spaces.
- RemoveMarginSpace, which deletes leading and trailing space characters.

For more information about transformers, see the *Conversion Agent Studio User Guide*.

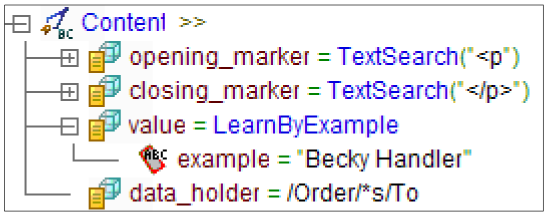
Why the Parser does not Use Delimiters to Remove the HTML Codes

Do you remember the Becky Handler anchor that you defined at the beginning of the exercise? We explained that the anchor removes HTML codes because it retrieves only the data after the > delimiter. The color-coding illustrates this:



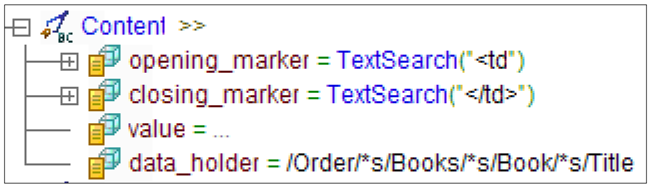
```
<p class=MsoBodyText>Becky Handler</p>
```

This works because the Becky Handler anchor is configured with the LearnByExample component, whose function is to interpret the delimiters in the example source.



```
Content >>
  opening_marker = TextSearch("<p")
  closing_marker = TextSearch("</p>")
  value = LearnByExample
    example = "Becky Handler"
  data_holder = /Order/*s/To
```

In the Content anchors of the RepeatingGroup, we deliberately omitted the LearnByExample component:



```
Content >>
  opening_marker = TextSearch("<td")
  closing_marker = TextSearch("</td>")
  value = ...
  data_holder = /Order/*s/Books/*s/Book/*s/Title
```

The `LearnByExample` component would not have worked correctly here. The problem is that the number of delimiters in the order table is not constant. In some of the table rows, the name of a book is italicized. In other rows, the name is not italicized. In some source documents, the table might contain other formatting variations. The formatting can change the HTML code, including the number and location of the delimiters.

That is why we left the `value` property empty. Instead, we relied on the default transformers to clean up the output.

Using Count to Resolve Ambiguities

Within the `EnclosedGroup`, we told you to assign the `count` property of two `Marker` anchors. The purpose was to resolve ambiguities in the example source.

In the `>Total<` anchor, we set `count = 2`. This was to avoid ending the repeating group prematurely, at the first instance of the string `>Total<`, which is in the column-heading row. The `count = 2` property means that the parser looks for the second instance of `>Total<`, which is in the last row of the table. This resolves the ambiguity.

Another interesting point is that we quoted the text `>Total<`, rather than `Total`. This resolves the ambiguity with the word `Total` that appears in the `Total Tennis (video)` row of the table. If we didn't do this, the repeating group would incorrectly end at the `Total Tennis (video)` row.

The other use of the `count` property is in the last `<td` marker of the `EnclosedGroup`, which lies within the last table row. There, we set `count = 3` to advance the parser by three table cells. The skipped cells in this row contain no data, so we do not need to parse them.

Using Transformers to Modify the Output

At this point, the XML output still has two small problems:

- Some of the book titles are punctuated incorrectly. The output is

  ```
  <Title>Topspin Serves Made Easy , by Roland Fasthitter</Title>
  ```

  instead of

  ```
  <Title>Topspin Serves Made Easy, by Roland Fasthitter</Title>
  ```

  The difference is an extra space character, located before the comma.
- The prices and totals do not have a currency symbol. The output is

  ```
  <Price>11.40</Price>
  <Total>46.19</Total>
  ```

  instead of

  ```
  <Price>$11.40</Price>
  <Total>$46.19</Total>
  ```

You will apply transformers to particular `Content` anchors. The transformers will modify the output of the anchors and correct these problems.

These transformers are in addition to the default transformers, which the parser applies to all the anchors.

To define the transformers:

1. In the IntelliScript, expand the first `Content` anchor of the `RepeatingGroup`.
2. Display the advanced properties of the `Content` anchor, and expand the `transformers` property.
3. At the three dots within the `transformers` property, nest a `Replace` transformer.
4. Set the properties of the `Replace` transformer as follows:

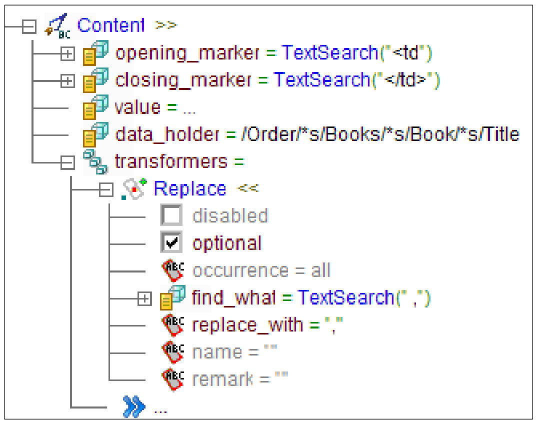
   ```
   find_what = TextSearch(" ,")
   ```

```
replace_with = ","
```

In other words, the transformer replaces a space followed by a comma with a comma.

5. Click the >> symbol to display the advanced properties of the `Replace` transformer. Select the `optional` property.

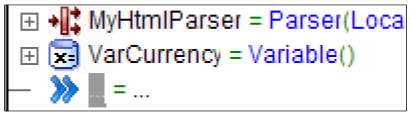
   This is important because the second row of the order table, "`Total Tennis (Video)`", does not contain a space followed by a comma. The transformer cannot perform the replacement on this row. If you do not select `optional`, the `Content` anchor fails and the output omits the second row of the order.

   ```
   Content >>
     opening_marker = TextSearch("<td")
     closing_marker = TextSearch("</td>")
     value = ...
     data_holder = /Order/*s/Books/*s/Book/*s/Title
     transformers =
       Replace <<
         disabled
         optional
         occurrence = all
         find_what = TextSearch(" ,")
         replace_with = ","
         name = ""
         remark = ""
       ...
   ```

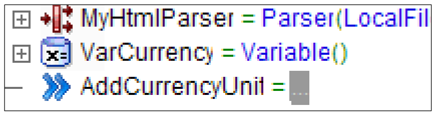
6. You will use an `AddString` transformer to prefix the prices and totals with the currency symbol. You could independently configure an independent `AddString` transformer in each appropriate `Content` anchor, but there is an easier way. You can configure a single `AddString` transformer as a global component. You can then use the global component wherever it is needed.

   To configure the global component, select the three dots to the left of the equal sign, at the top level of the IntelliScript, the level where `MyHtmlParser` and `VarCurrency` are defined. This is called the global level.

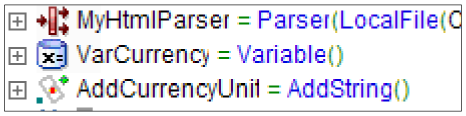
   ```
   MyHtmlParser = Parser(Loca
   VarCurrency = Variable()
   ... = ...
   ```

   Press Enter, and type `AddCurrencyUnit`. This is the identifier of the global component.

   ```
   MyHtmlParser = Parser(LocalFil
   VarCurrency = Variable()
   AddCurrencyUnit = ...
   ```

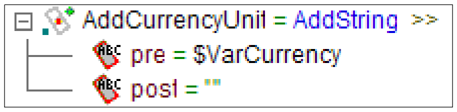
7. Press Enter at the second three-dots symbol on the same line, following the equals sign. Insert an `AddString` transformer.

   ```
   MyHtmlParser = Parser(LocalFile(C
   VarCurrency = Variable()
   AddCurrencyUnit = AddString()
   ```

8. Select the `pre` property of the `AddString` transformer, and press Enter. At the right of the text box, click the browse button. This displays a Schema view, where you can select a data holder.

   In the Schema view, select `VarCurrency`.

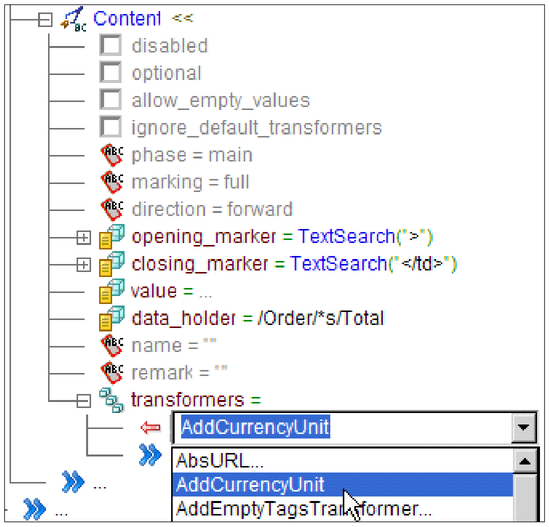
   

   ```
   AddCurrencyUnit = AddString >>
     pre = $VarCurrency
     post = ""
   ```

   This means that the `AddString` transformer prefixes its input with the value of `VarCurrency`, which is the currency symbol.

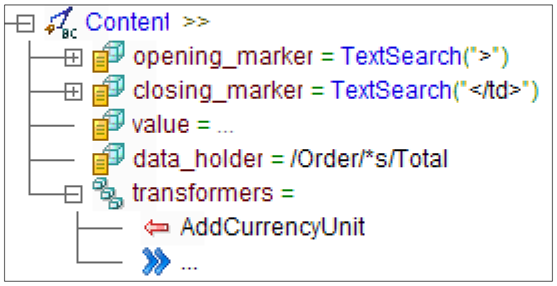
9. Now, you can insert the `AddCurrencyUnit` component in the appropriate locations of `MyHtmlParser`.

For example, display the advanced properties of the last `Content` anchor in the `EnclosedGroup`, which is the one that is mapped to `/Order/*s/Total`. Under its `transformers` property, select the three dots and press Enter. The drop-down list displays the `AddCurrencyUnit` global component, which you have just defined. Select the global component. This applies the transformer to the anchor.



```
Content <<
    disabled
    optional
    allow_empty_values
    ignore_default_transformers
    phase = main
    marking = full
    direction = forward
    opening_marker = TextSearch(">")
    closing_marker = TextSearch("</td>")
    value = ...
    data_holder = /Order/*s/Total
    name = ""
    remark = ""
    transformers =
        AddCurrencyUnit
        AbsURL...
        AddCurrencyUnit
        AddEmptyTagsTransformer...
  ...
...
```

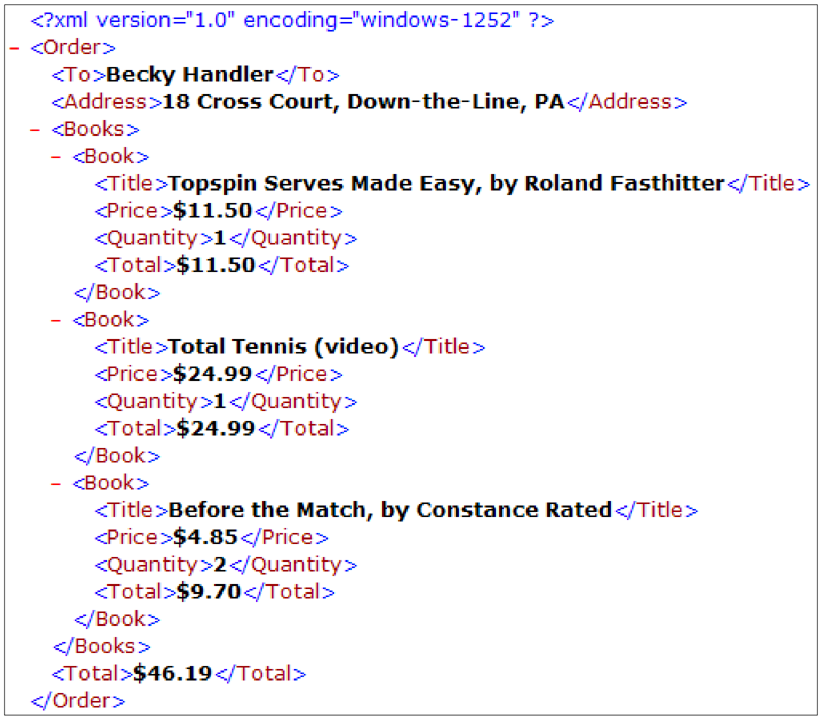
The result should look like this:



```
Content >>
    opening_marker = TextSearch(">")
    closing_marker = TextSearch("</td>")
    value = ...
    data_holder = /Order/*s/Total
    transformers =
        AddCurrencyUnit
        ...
```

In the same way, insert `AddCurrencyUnit` in the `transformers` property of the second and fourth `Content` anchors in the `RepeatingGroup`, which are mapped to `Price` and `Total`, respectively.

10. Run the parser. The result should be:



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Order>
    <To>Becky Handler</To>
    <Address>18 Cross Court, Down-the-Line, PA</Address>
  - <Books>
    - <Book>
        <Title>Topspin Serves Made Easy, by Roland Fasthitter</Title>
        <Price>$11.50</Price>
        <Quantity>1</Quantity>
        <Total>$11.50</Total>
      </Book>
    - <Book>
        <Title>Total Tennis (video)</Title>
        <Price>$24.99</Price>
        <Quantity>1</Quantity>
        <Total>$24.99</Total>
      </Book>
    - <Book>
        <Title>Before the Match, by Constance Rated</Title>
        <Price>$4.85</Price>
        <Quantity>2</Quantity>
        <Total>$9.70</Total>
      </Book>
    </Books>
    <Total>$46.19</Total>
  </Order>
```

Notice that the `Title` element does not contain an extra space before the comma. The `Price` and `Total` elements contain the `$` sign. These are the effects of the transformers that you defined.

Global Components

Global components are useful when you need to use the same component configuration repeatedly in a project.

You can define any Conversion Agent component as a global component, for example, a transformer, an action, or an anchor. The identifier of the global component appears in the drop-down list at the appropriate locations of the IntelliScript. You can use the global component just like any other component.

In this exercise, you defined three global components:

- The parser, called `MyHtmlParser`, which you defined by using the wizard.
- The `VarCurrency` variable that you defined at the beginning of the exercise.
- The `AddCurrencyUnit` transformer that you added at the end of the exercise.

Testing the Parser on Another Source Document

As a final step, test the parser on the source document `OrderWithoutCurrency.doc`. This document is identical to the example source (`Order.doc`), except that it is missing the optional currency line.

The purpose of this test is to confirm that the parser correctly processes a document when the currency line is missing.

To perform the test:

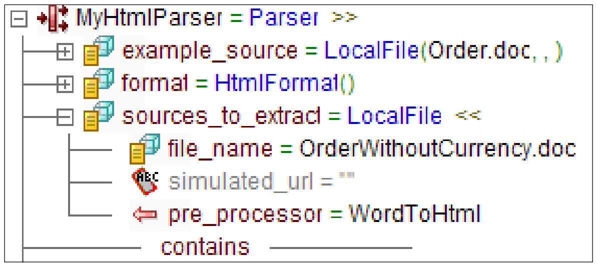
1. Display the advanced properties of `MyHtmlParser`.

2. Assign the `sources_to_extract` property a value of `LocalFile`.

This means that the parser processes a file on the local computer, instead of the example source.

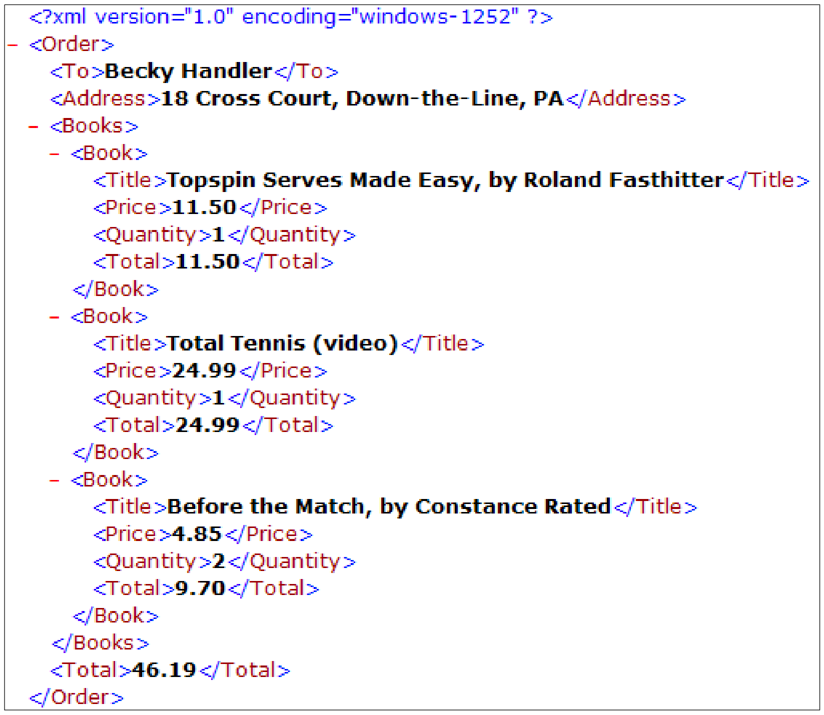
3. Edit the `file_name` property, and browse to `OrderWithoutCurrency.doc`.
4. Display the advanced properties of `LocalFile`, and assign `pre_processor = WordToHtml`.

   This means that Conversion Agent runs the `WordToHtml` processor on the document before it runs the parser.

```
⊟ MyHtmlParser = Parser  >>
  ├⊞ example_source = LocalFile(Order.doc, , )
  ├⊞ format = HtmlFormat()
  ├⊟ sources_to_extract = LocalFile  <<
  │   ├── file_name = OrderWithoutCurrency.doc
  │   ├── simulated_url = ""
  │   └── pre_processor = WordToHtml
  └──── contains ────
```

5. Run the parser. The output should be identical to the above, except that the prices and totals are not prefixed with a `$` sign.

```
  <?xml version="1.0" encoding="windows-1252" ?>
- <Order>
    <To>Becky Handler</To>
    <Address>18 Cross Court, Down-the-Line, PA</Address>
  - <Books>
    - <Book>
        <Title>Topspin Serves Made Easy, by Roland Fasthitter</Title>
        <Price>11.50</Price>
        <Quantity>1</Quantity>
        <Total>11.50</Total>
      </Book>
    - <Book>
        <Title>Total Tennis (video)</Title>
        <Price>24.99</Price>
        <Quantity>1</Quantity>
        <Total>24.99</Total>
      </Book>
    - <Book>
        <Title>Before the Match, by Constance Rated</Title>
        <Price>4.85</Price>
        <Quantity>2</Quantity>
        <Total>9.70</Total>
      </Book>
    </Books>
    <Total>46.19</Total>
  </Order>
```

6. When you finish, you can delete the `sources_to_extract` value. This was a temporary setting that you needed only for testing the parser.

Points to Remember

To parse a Microsoft Word document, you can use a document processor such as `WordToHtml`, which converts the document to a parser-friendly HTML format.

To parse HTML code, you can use the `EnclosedGroup` anchor or the `opening_marker` and `closing_marker` properties of the `Content` anchor. These approaches take advantage of the characteristic opening-and-closing tag structure of HTML. If you define an HTML tag as a `Marker`, it is usually best not to select the `match_case` property because HTML is not case sensitive.

To resolve ambiguities between multiple instances of the same text, you can use the count property of a `Marker`.

Use variables to store retrieved data, which you do not want to display in the XML output of a parser. The variables are useful as input to other components.

Use transformers to modify the output of anchors. You can apply default transformers to all the anchors, or you can apply transformers to specific anchors.

To exclude HTML tags from the parsing output, you can either define the anchors using delimiters, or you can rely on the default transformers to remove the code.

Global components are useful when you need to use the same component configuration repeatedly.

To test a parser on additional documents other than the example source, assign the `sources_to_extract` property. Assign a document processor to the document if it needs one.

CHAPTER 6

Defining a Serializer

This chapter includes the following topics:

- Overview, 77
- Creating the Project, 78
- Configuring the Serializer, 80
- Calling the Serializer Recursively, 81
- Points to Remember, 84

Overview

In the preceding exercises, you have defined parsers, which convert documents in various formats to XML. In this exercise, you will create a serializer that works in the opposite direction, converting XML to another format.

Usually, it is easier to define a serializer than a parser because the input is a fully structured, unambiguous XML document.

The serializer that you will define is very simple. It contains only four serialization anchors, which are the opposite of the anchors that you use in parsing. Nonetheless, the serializer has some interesting points:

- The serializer is recursive. That is, it calls itself repetitively to serialize the nested sections of an XML document.
- The output of the serializer is a worksheet that you can open in Excel.

You will define the serializer by editing the IntelliScript. It is also possible to generate a serializer automatically by inverting the operation of a parser. For more information about defining serializers, see the *Conversion Agent Studio User Guide*.

Prerequisite

The output of this exercise is a `*.csv` (comma separated values) file. You can view the output in Notepad, but for the most meaningful display, you need Microsoft Excel.

You do not need Excel to run the serializer. It is recommended only to view the output.

Requirements Analysis

The input XML document is:

```
tutorials\Exercises\Files_For_Tutorial_5\FamilyTree.xml
```

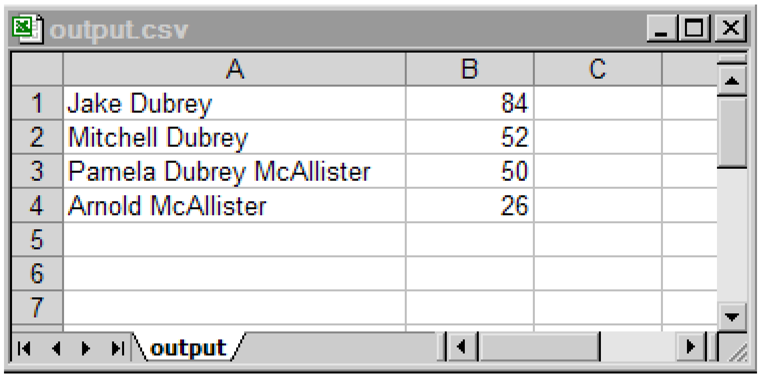
The document is an XML representation of a family tree. Notice the inherently recursive structure: each `Person` element can contain a `Children` element, which contains additional `Person` elements.

```
<Person>
  <Name>Jake Dubrey</Name>
  <Age>84</Age>
  <Children>
    <Person>
      <Name>Mitchell Dubrey</Name>
      <Age>52</Age>
    </Person>
    <Person>
      <Name>Pamela Dubrey McAllister</Name>
      <Age>50</Age>
      <Children>
        <Person>
          <Name>Arnold McAllister</Name>
          <Age>26</Age>
        </Person>
      </Children>
    </Person>
  </Children>
</Person>
```

Our goal is to output the names and ages of the `Person` elements as a `*.csv` file, which has the following structure:

```
Jake Dubrey,84
Mitchell Dubrey,52
Pamela Dubrey McAllister,50
Arnold McAllister,26
```

Excel displays the output as a worksheet looking like this:



	A	B	C
1	Jake Dubrey	84	
2	Mitchell Dubrey	52	
3	Pamela Dubrey McAllister	50	
4	Arnold McAllister	26	
5			
6			
7			

Creating the Project

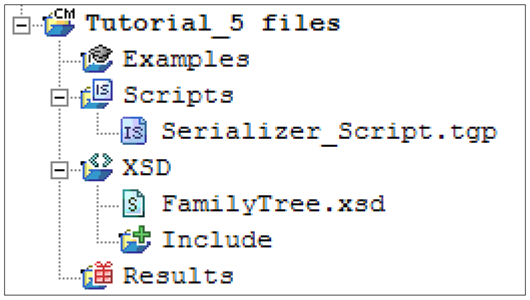
To create the project:

1. On the Conversion Agent Studio menu, click File > New > Project.
2. Under the Conversion Agent node, select a Serializer Project.
3. On the following wizard pages, specify the following options:
   - Name the project `Tutorial_5`.
   - Name the serializer `FamilyTreeSerializer`.
   - Name the script file `Serializer_Script`.

4. When you reach the Schema page, browse to the schema `FamilyTree.xsd`, which is in the `tutorials\Exercises\Files_For_Tutorial_5` folder.

   The schema defines the structure of the input XML document. If you are new to XSD, you can open the schema in an XSD editor or in Notepad, and examine how the recursive data structure is defined.

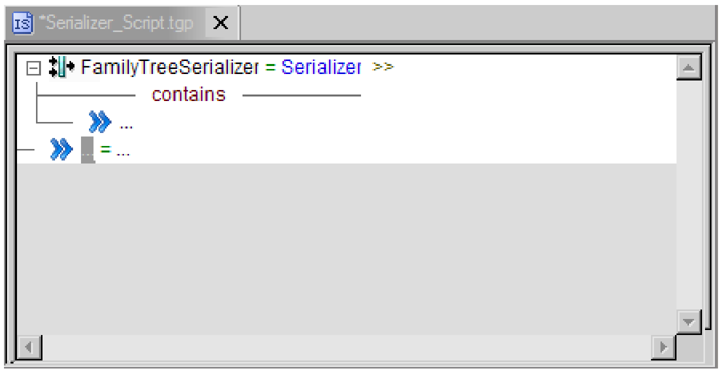
5. When you finish the wizard, the Conversion Agent Explorer displays the new project. Double click the `Serializer_Script.tgp` file to edit it.



```
Tutorial_5 files
    Examples
    Scripts
        Serializer_Script.tgp
    XSD
        FamilyTree.xsd
        Include
    Results
```

6. Unlike a parser, a serializer does not have an example source file. Optionally, you can hide the empty example pane of the IntelliScript editor.

   To do this, click IntelliScript > IntelliScript or click the toolbar button that is labeled Show IntelliScript Pane Only. To restore the example pane, click IntelliScript > Both on click the button that is labeled Show Both IntelliScript and Example Panes.



```
*Serializer_Script.tgp
FamilyTreeSerializer = Serializer >>
    contains
        ...
    = ...
```

7. To design the serializer, you will use the `FamilyTree.xml` input document, whose content is presented above.

   Although not required, it is a good idea to organize all the files that you use to design and test a project in the project folder, which is located in your Eclipse workspace. We suggest that you copy the `FamilyTree.xml` file from

   ```
   tutorials\Exercises\Files_For_Tutorial_5
   ```

   to the project folder, which is by default

   ```
   My Documents\SAP\ConversionAgent\4.0\workspace\Tutorial_5
   ```

Determining the Project Folder Location

Your project folder might be in a non-default location for either of the following reasons:

- In the New Project wizard, you selected a non-default location.
- Your copy of Eclipse is configured to use a non-default workspace location.

To determine the location of the project folder, select the project in the Conversion Agent Explorer, and click the File > Properties command. Alternatively, click anywhere in the IntelliScript editor, and then click the Project > Properties command. The Info tab of the properties window displays the location.

Project Properties

The properties window displays many useful options, such as the input and output encoding that are used in your documents and the XML validation options. For more information the project properties, see the *Conversion Agent Studio User Guide*.

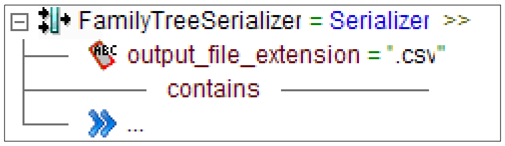
Configuring the Serializer

You are now ready to configure the serializer properties and add the serialization anchors. You must do this by editing the IntelliScript. There is no analogy to the select-and-click approach that you used to configure parsers.

To configure the serializer:

1. Display the advanced properties of the serializer, and set `output_file_extension =.csv`, with a leading period.

   When you run the serializer in the Studio, this causes the output file to have the name `output.csv`. By default, `*.csv` files open in Microsoft Excel.

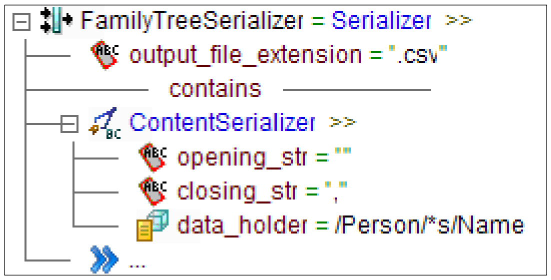


```
FamilyTreeSerializer = Serializer >>
    output_file_extension = ".csv"
    contains
    ...
```

2. Under the `contains` line of the serializer, insert a `Content Serializer` serialization anchor and configure its properties as follows:

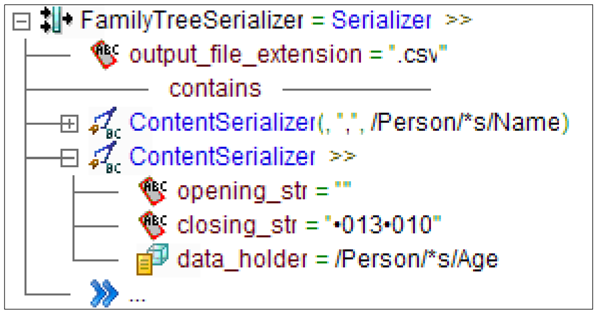
```
data_holder = /Person/*s/Name
closing_str = ","
```

   This means that the serialization anchor writes the content of the `/Person/*s/Name` data holder to the output file. It appends the closing string `","` (a comma).



```
FamilyTreeSerializer = Serializer >>
    output_file_extension = ".csv"
    contains
    ContentSerializer >>
        opening_str = ""
        closing_str = ","
        data_holder = /Person/*s/Name
    ...
```

3. Define a second `ContentSerializer` as illustrated:



```
FamilyTreeSerializer = Serializer >>
    output_file_extension = ".csv"
    contains
    ContentSerializer(, ",", /Person/*s/Name)
    ContentSerializer >>
        opening_str = ""
        closing_str = "•013•010"
        data_holder = /Person/*s/Age
    ...
```

   This `ContentSerializer` writes the `/Person/*s/Age` data holder to the output. It appends a carriage return (ASCII code `013`) and a linefeed (ASCII `010`) to the output.

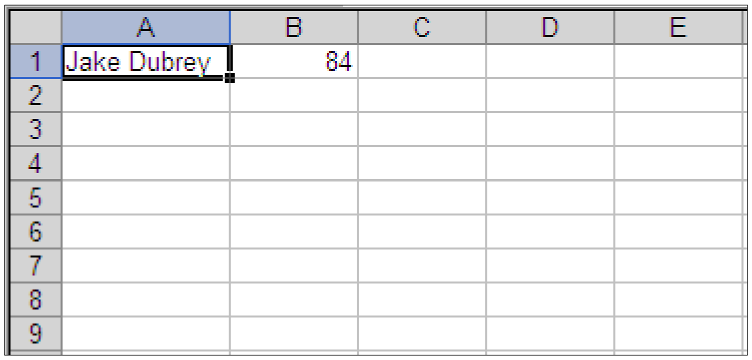
   To type the ASCII codes:

- Select the `closing_str` property and press Enter.
- On the keyboard, press Ctrl+a. This displays a small dot in the text box.
- Type `013`.
- Press Ctrl+a again.
- Type `010`.
- Press Enter to complete the property assignment.

4. Run the serializer. To do this:

- Set the serializer as the startup component.
- On the menu, click Run > Run.
- At the prompt, browse to the test input file, `FamilyTree.xml`.
- When the serializer has completed, examine the Events view for errors.
- In the Conversion Agent Explorer view, under `Results`, double-click `output.csv` to view the output.

Assuming that Excel is installed on the computer, Conversion Agent displays an Excel window, like this:



	A	B	C	D	E
1	Jake Dubrey	84			
2					
3					
4					
5					
6					
7					
8					
9					

If Excel is not installed on the computer, you can view the output file in Notepad. Alternatively, you can copy the file to another computer where Excel is installed, and open it there.

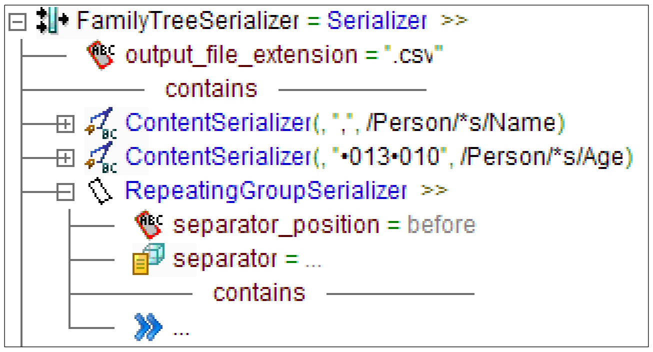
Calling the Serializer Recursively

So far, the results contain only the top-level `Person` element. You need to configure the serializer to move deeper in the XML tree and process the child `Person` elements.

To call the serializer recursively:

1. Insert a `RepeatingGroupSerializer`.

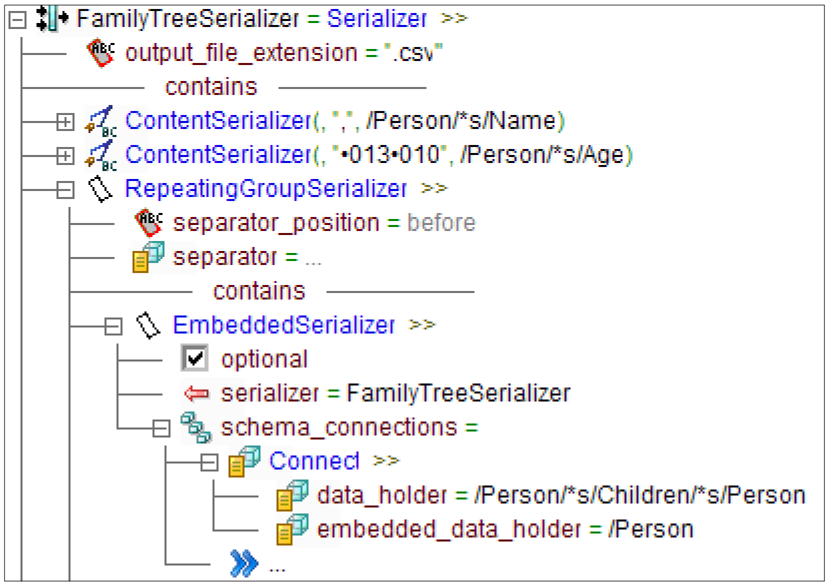
   You can use this serialization anchor to iterate over a repetitive structure in the input, and to generate a repetitive structure in the output. In this case, the `RepeatingGroupSerializer` will iterate over all the `Person` elements at a given level of nesting.



```
FamilyTreeSerializer = Serializer >>
    output_file_extension = ".csv"
    contains
    ContentSerializer(, ",", /Person/*s/Name)
    ContentSerializer(, "•013•010", /Person/*s/Age)
    RepeatingGroupSerializer >>
        separator_position = before
        separator = ...
        contains
        ...
```

2. Within the `RepeatingGroupSerializer`, nest an `EmbeddedSerializer`. The purpose of this serialization anchor is to call a secondary serializer.

3. Assign the properties of the `EmbeddedSerializer` as illustrated.

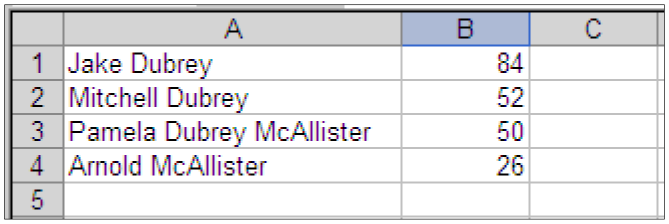


```
FamilyTreeSerializer = Serializer >>
    output_file_extension = ".csv"
    contains
    ContentSerializer(, ",", /Person/*s/Name)
    ContentSerializer(, "•013•010", /Person/*s/Age)
    RepeatingGroupSerializer >>
        separator_position = before
        separator = ...
        contains
        EmbeddedSerializer >>
            ✓ optional
            serializer = FamilyTreeSerializer
            schema_connections =
                Connect >>
                    data_holder = /Person/*s/Children/*s/Person
                    embedded_data_holder = /Person
                ...
```

The properties have the following meanings:

- The assignment `serializer = FamilyTreeSerializer` means that the secondary serializer is the same as the main serializer. In other words, the serializer calls itself recursively.
- The `schema_connections` property means that the secondary serializer should process `/Person/*s/Children/*s/Person` as though it were a top-level `/Person` element. This is what lets the serializer move down through the generations of the family tree.
- The `optional` property means that the secondary serializer does not cause the main serializer to fail when it runs out of data.

4. Run the serializer again. The result should be:



	A	B	C
1	Jake Dubrey	84	
2	Mitchell Dubrey	52	
3	Pamela Dubrey McAllister	50	
4	Arnold McAllister	26	
5			

Defining Multiple Components in a Project

None of the exercises in this book contain multiple components. In the exercises throughout this book, each project contains a single parser, serializer, or mapper. We did this for simplicity.

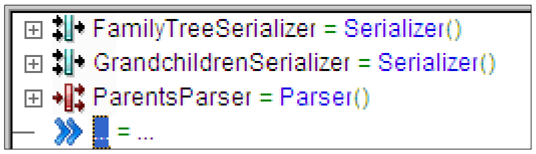
It is quite possible for a single project to contain multiple components, for example:

- Multiple parsers, serializers, or mappers
- Multiple script (TGP) files
- Multiple XSD schemas

The following paragraphs are a brief summary of some possible project configurations involving multiple components. For more information, see the *Conversion Agent Studio User Guide*.

Multiple Transformation Components

To define multiple transformation components such as parsers, serializers, or mappers, insert them at the global level of the IntelliScript. Here is an example:



```
FamilyTreeSerializer = Serializer()
GrandchildrenSerializer = Serializer()
ParentsParser = Parser()
 = ...
```

To run one of the components, set it as the startup component and use the commands on the Run menu.

The startup component can call secondary parsers, serializers, or mappers to process portions of a document. That is what you did in this exercise, with the interesting twist that the main and secondary serializers were the same—a recursive call.

Multiple Script Files

You can use multiple script files to organize your work.

To create a script file,

1. Right-click the `Scripts` node in the Conversion Agent Explorer, and click New > Script.

To add a script that you created in another project, click Add File.

2. To open a script file for editing, double-click the file in the Conversion Agent Explorer.

Multiple XSD Schemas

To add multiple schemas to a project, right-click the `XSD` node in the Conversion Agent Explorer and click Add File. To create an empty XSD schema that you can edit in any editor, click New > XSD.

Points to Remember

A serializer is the opposite of a parser: it converts XML to other formats. You can design a serializer that outputs to any data format, for example, Microsoft Excel.

You can create a serializer either by generating it from an existing parser or by editing the IntelliScript. A serializer contains serialization anchors that are analogous to the anchors that you use in a parser, but work in the opposite direction.

In an IntelliScript editor, you can hide or display the panes by choosing the commands on the IntelliScript menu or on the toolbar.

We recommend that you store all files associated with a project in the project folder, located in your Eclipse workspace. You can determine the folder location by clicking File > Properties or Project > Properties.

A single project can contain multiple transformation components such as parsers, serializers, and mappers. The startup component can call secondary serializers, parsers, or mappers, which are defined in the same project. Secondary components can process portions of a document. Recursion, a component calling itself, is supported.

CHAPTER 7

Defining a Mapper

This chapter includes the following topics:

- Overview, 85
- Creating the Project, 86
- Configuring the Mapper, 87
- Points to Remember, 88

Overview

So far, you have worked with two major components. With parsers, you converted documents of any format to XML. With serializers, you changed XML documents to other formats.

In this chapter, you will work with a mapper, which performs XML to XML conversions. The purpose is to change the XML structure or vocabulary of the data.

The mapper design is similar to that of parsers and serializers. Within the main `Mapper` component, you can nest mapper anchors, `Map` actions, and other components that perform the mapping operations.

The exercise presents a simple mapper that generates an XML summary report. Among other features, the exercise demonstrates how to configure an initially empty project, which you create by using the Blank Project wizard. You can use a blank project to configure any kind of transformation, in addition to mappers.

Requirements Analysis

The goal of this exercise is to use an existing XML file to generate a new XML file that has a modified data structure. The files are provided in the following folder:

```
tutorials\Exercises\Tutorial_6
```

Input XML

The input of the mapper is an XML file (`Input.xml`) that records the identification numbers and names of several persons. The input conforms to the schema `Input.xsd`.

```
<Persons>
  <Person ID="10">Bob</Person>
  <Person ID="17">Larissa</Person>
  <Person ID="13">Marie</Person>
</Persons>
```

Output XML

The expected output XML (`ExpectedOutput.xml`) is a summary report listing the names and IDs separately. The output conforms to the schema `Output.xsd`:

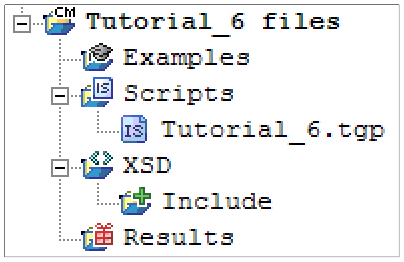
```
<SummaryData>
  <Names>
    <Name>Bob</Name>
    <Name>Larissa</Name>
    <Name>Marie</Name>
  </Names>
  <IDs>
    <ID>10</ID>
    <ID>17</ID>
    <ID>13</ID>
  </IDs>
</SummaryData>
```

Creating the Project

To create the mapper, you will start with a blank project.

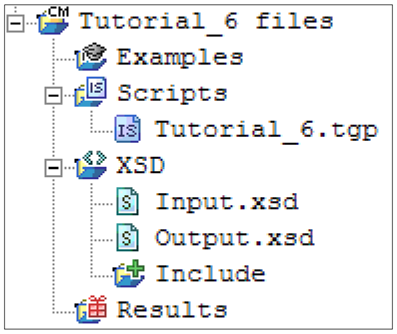
To create the project:

1. On the Conversion Agent Studio menu, click File > New > Project.
2. Under the Conversion Agent node, select a Blank Project.
3. In the wizard, name the project `Tutorial_6`, and click Finish.
4. In the Conversion Agent Explorer, expand the project. Notice that it contains a default TGP script file, but it contains no XSD schemas or other components.



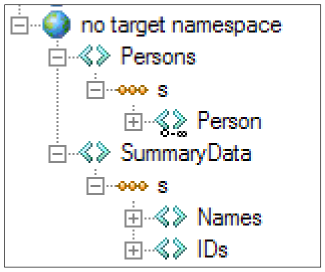
```
Tutorial_6 files
  Examples
  Scripts
    Tutorial_6.tgp
  XSD
    Include
  Results
```

5. Right-click the `XSD` node and click Add File. Add the schemas `Input.xsd` and `Output.xsd`. Both schemas are necessary because you must define the structure of both the input and output XML.



```
Tutorial_6 files
  Examples
  Scripts
    Tutorial_6.tgp
  XSD
    Input.xsd
    Output.xsd
    Include
  Results
```

6. The Schema view displays the elements that are defined in both schemas. You will work with the `Persons` branch of the tree for the input, and with the `SummaryData` branch for the output.



```
no target namespace
  Persons
    s
      Person
  SummaryData
    s
      Names
      IDs
```

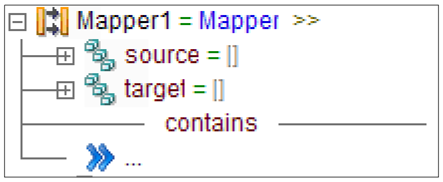
7. We recommend that you copy the test documents, `Input.xml` and `ExpectedOutput.xml`, into the project folder. By default, the folder is:

```
My Documents\SAP\ConversionAgent\4.0\workspace\Tutorial_6
```

Configuring the Mapper

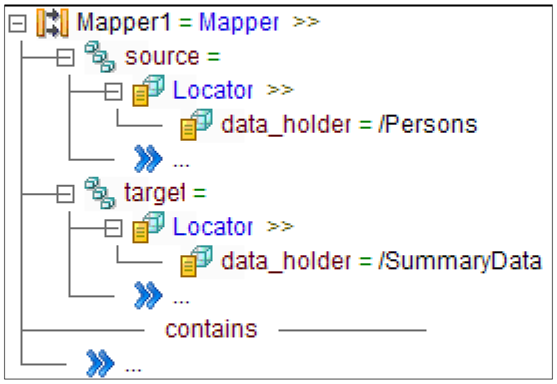
To configure the mapper:

1. Open the script file, which has the default name `Tutorial_6.tgp`, in an IntelliScript editor.
2. Optionally, use the commands on the IntelliScript menu or the toolbar to display the IntelliScript pane only. A mapper does not use an example source, so you do not need the example pane.
3. Define a global component called `Mapper1`, and give it a type of `Mapper`. For more information, see "Global Components" on page 73.



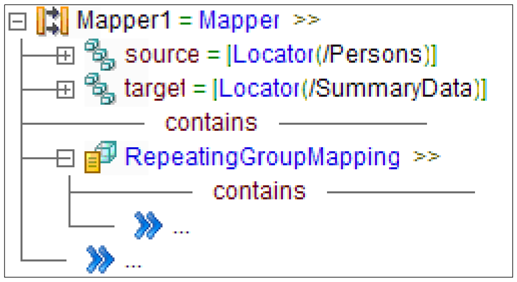
```
Mapper1 = Mapper >>
  source = []
  target = []
  contains
  ...
```

4. Assign the `source` and `target` properties of the mapper as illustrated. The properties define the schema branches where the mapper will retrieve its input and store its output.



```
Mapper1 = Mapper >>
  source =
    Locator >>
      data_holder = /Persons
    ...
  target =
    Locator >>
      data_holder = /SummaryData
    ...
  contains
  ...
```

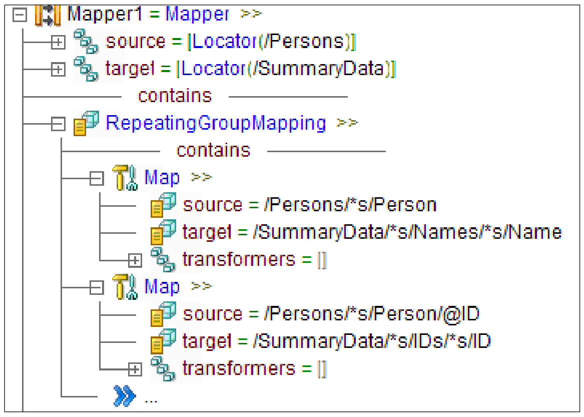
5. Under the `contains` line of the mapper, insert a `RepeatingGroupMapping` component. You will use this mapper anchor to iterate over the repetitive XML structures in the input and output.



```
Mapper1 = Mapper >>
  source = [Locator(/Persons)]
  target = [Locator(/SummaryData)]
  contains
  RepeatingGroupMapping >>
    contains
    ...
  ...
```

6. Within the `RepeatingGroupMapping`, insert two `Map` actions. The purpose of these actions is to copy the data from the input to the output.

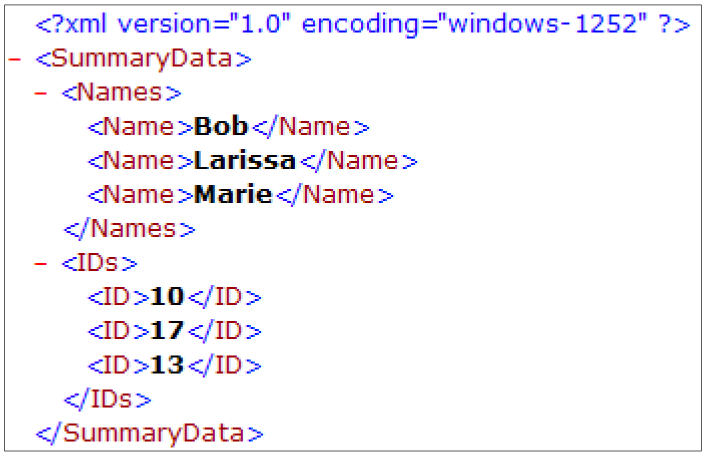
   Configure the `source` property of each `Map` action to retrieve a data holder from the input. Configure the `target` property to write the corresponding data holder in the output.



```
Mapper1 = Mapper >>
  source = [Locator(/Persons)]
  target = [Locator(/SummaryData)]
  contains
  RepeatingGroupMapping >>
    contains
    Map >>
      source = /Persons/*s/Person
      target = /SummaryData/*s/Names/*s/Name
      transformers = []
    Map >>
      source = /Persons/*s/Person/@ID
      target = /SummaryData/*s/IDs/*s/ID
      transformers = []
    ...
```

7. Set the mapper as the startup component and run it. Check the Events view for errors.

8. Compare the results file, which is called `output.xml`, with `ExpectedOutput.xml`. If you have configured the mapper correctly, the files should be identical, except perhaps for the `<?xml?>` processing declaration, which depends on options in the project properties.



```
  <?xml version="1.0" encoding="windows-1252" ?>
- <SummaryData>
  - <Names>
      <Name>Bob</Name>
      <Name>Larissa</Name>
      <Name>Marie</Name>
    </Names>
  - <IDs>
      <ID>10</ID>
      <ID>17</ID>
      <ID>13</ID>
    </IDs>
  </SummaryData>
```

Points to Remember

A mapper converts an XML source document to an XML output document conforming to a different schema.

You can create a mapper by editing a blank project in the IntelliScript. A mapper contains mapper anchors, which are analogous to the anchors that you use in a parser or to the serialization anchors in a serializer. It uses `Map` actions to copy the data from the input to the output.

CHAPTER 8

Running Conversion Agent Engine

This chapter includes the following topics:

- Overview, 91
- Deploying a Transformation as a Service, 92
- COM API Application, 92
- Running the COM API Application, 94
- Points to Remember, 95

Overview

After you have created and tested a transformation, you need to move it from the development stage to production. There are two main steps to do this:

1. In Conversion Agent Studio, deploy the transformation as a service. This makes it available to run in Conversion Agent Engine.
2. Launch an application that activates the Engine and runs the service.

In this tutorial, you will perform these steps on the HL7 parser that you already created. For more information about the parser, see “Defining an HL7 Parser” on page 23. If you prefer, you can perform the exercise on any other transformation that you have prepared.

The application that activates the Engine will be a Microsoft Visual Basic 6 program that calls the Conversion Agent COM API. For the benefit of Conversion Agent users who do not have Visual Basic, we provide both the source code and the compiled program.

As an alternative to the approach that you will learn in this tutorial, you can run Conversion Agent services:

- From the command line.
- By using the Conversion Agent Java, C, C++, .NET, or Web Service APIs.
- By using the CGI web interface.

For more information, see the *Conversion Agent Engine Developer Guide*.

In addition, you can run services by using the Conversion Agent process module for SAP PI. For more information, see *Deploying and Using Conversion Agent*.

Deploying a Transformation as a Service

Deploying a Conversion Agent service means making a transformation available to Conversion Agent Engine. By default, the repository location is:

```
c:\Program Files\SAP\ConversionAgent\ServiceDB
```

Note: There is no relation between Conversion Agent services and Windows services. You cannot view or administer Conversion Agent services in the Windows Control Panel.

To deploy a transformation as a service:

1. In the Conversion Agent Explorer, select the project. We suggest that you use the `Tutorial_2` project that you already prepared. For more information about `Tutorial_2`, see “Defining an HL7 Parser” on page 23.
2. Confirm that the startup component is selected and that the transformation runs correctly.
3. On the Conversion Agent menu, click Project > Deploy.
4. The Deploy Service window displays the service details. Edit the information as required.
5. Click the Deploy button.
6. At the lower right of the Conversion Agent Studio window, display the Repository view. The view lists the service that you have deployed, along with any other Conversion Agent services that have been deployed on the computer.

COM API Application

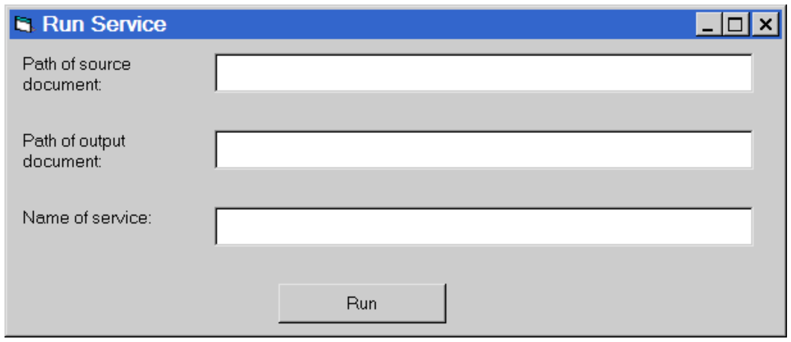
To illustrate how to run a Conversion Agent service from an API application, we have supplied a sample application that calls the Conversion Agent COM API. The application is programmed in Microsoft Visual Basic 6.0. The source code and the compiled executable file are stored in the folder:

```
tutorials\Exercises\CMComApiTutorial
```

The application is described in the following paragraphs.

Source Code

The Visual Basic application displays the following form:



Enter the source document path, an output document path, and the name of a Conversion Agent service. When you click the Run button, the application executes the following code:

```
Private Sub cmdRun_Click()

    Dim objCMEngine As CM_COM3Lib.CMEngine    'Engine
    Dim objCMRequest As CM_COM3Lib.CMRequest  'Request generation
    Dim objCMStatus As CM_COM3Lib.CMStatus    'Status display
    Dim strRequest As String  'Request string
    Dim strOutput As String   'Output string (not used in this sample)
    Dim strStatus As String   'Status string

    'Check for the required input
    If txtSource = "" Then
        MsgBox "Enter the path of the source document"
        Exit Sub
    End If

    If txtOutput = "" Then
        MsgBox "Enter the path of the output document"
        Exit Sub
    End If

    If txtService = "" Then
        MsgBox "Enter the name of a deployed service"
        Exit Sub
    End If

    Me.MousePointer = vbHourglass

    'Initialize Engine
    Set objCMEngine = New CM_COM3Lib.CMEngine
    objCMEngine.InitEngine

    'Generate a request string
    Set objCMRequest = New CM_COM3Lib.CMRequest
    strRequest = objCMRequest.Generate( _
                    txtService.Text, _
                    objCMRequest.FileInput(txtSource.Text), _
                    objCMRequest.FileOutput(txtOutput.Text), _
                    "", "", "")

    'Execute the request
    strStatus = objCMEngine.Exec(strRequest, "", "", strOutput)

    'Display the status
    Set objCMStatus = New CM_COM3Lib.CMStatus
    Call MsgBox( _
            "Return code = " & objCMStatus.IsGood(strStatus) & vbCr & _
            objCMStatus.GetDescription(strStatus), _
            vbOKOnly, "Service Status")

    Me.MousePointer = vbDefault

    Set objCMEngine = Nothing
    Set objCMRequest = Nothing
    Set objCMStatus = Nothing

End Sub
```

Explanation of the API Calls

The Visual Basic sample uses three Conversion Agent COM API objects:

COM API Objects	Description
`CMRequest`	Generates a request string, which specifies the service name, the input, the output, and other details of the operations that Conversion Agent Engine should perform.
`CMEngine`	Executes the request and generates the output.
`CMStatus`	Displays information about the results of the Conversion Agent Engine operation, such as a return code and error messages.

The steps for using these objects, demonstrated in the sample, are as follows:

1. Define the request parameters, such as the input location, output location, and Conversion Agent service name.
2. Use `CMRequest` to generate a request string.
3. Use `CMEngine` to execute the request.
4. Use `CMStatus` to confirm that the request ran successfully.

For more information, see the *Conversion Agent COM API Reference*.

Running the COM API Application

If you have the Visual Basic 6.0 run-time library on your computer, which is true on most Windows computers, you can run the sample Visual Basic application. Of course, if this were a production application, we would package it in a setup file that includes the required run-time library.

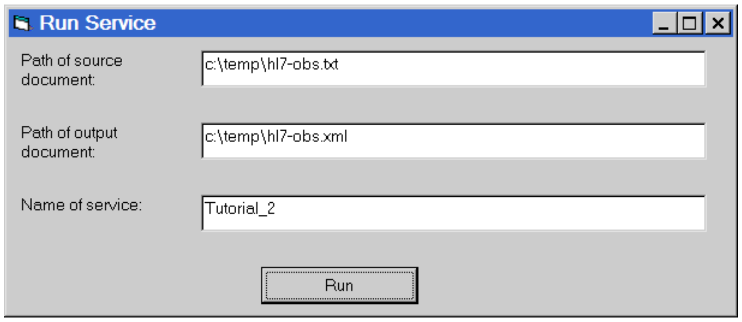
To run the COM API application:

1. In your working copy of the `Tutorials` folder, execute the following file:

   ```
   tutorials\Exercises\CMComApiTutorial\CMComApiTutorial.exe
   ```

2. Type the full path of the source document and the output document. Type the name of the Conversion Agent service that you deployed. By default, it is the name of the project.

   For example, to run the sample HL7 parser, you might copy the source document `hl7-obs.txt` to the `c:\temp` directory, and enter the options as illustrated below:

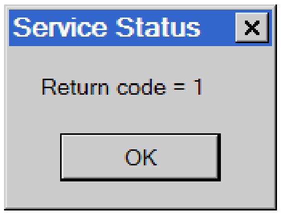
   Run Service

   Path of source document: c:\temp\hl7-obs.txt

   Path of output document: c:\temp\hl7-obs.xml

   Name of service: Tutorial_2

3. Click the Run button. After a moment, the application displays a status message:

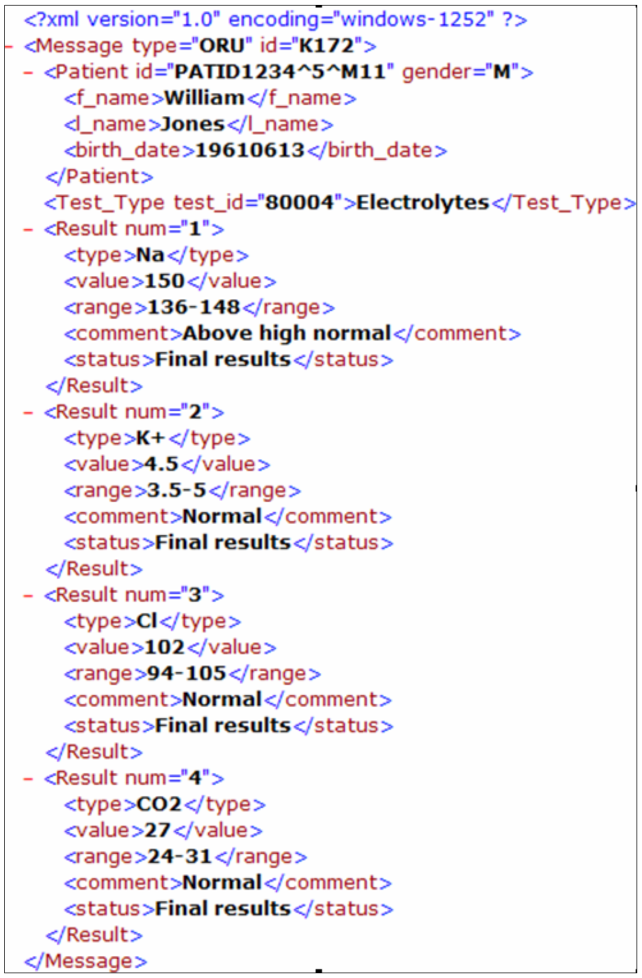
   Service Status

   Return code = 1

   A return code of 1 means success. If an error occurred during the service operation, the application displays an error message.

4. Open the output file in Notepad or Internet Explorer.

The output should be identical to the output that you generated when you ran the application in the Studio.



```
<?xml version="1.0" encoding="windows-1252" ?>
- <Message type="ORU" id="K172">
  - <Patient id="PATID1234^5^M11" gender="M">
      <f_name>William</f_name>
      <l_name>Jones</l_name>
      <birth_date>19610613</birth_date>
    </Patient>
    <Test_Type test_id="80004">Electrolytes</Test_Type>
  - <Result num="1">
      <type>Na</type>
      <value>150</value>
      <range>136-148</range>
      <comment>Above high normal</comment>
      <status>Final results</status>
    </Result>
  - <Result num="2">
      <type>K+</type>
      <value>4.5</value>
      <range>3.5-5</range>
      <comment>Normal</comment>
      <status>Final results</status>
    </Result>
  - <Result num="3">
      <type>Cl</type>
      <value>102</value>
      <range>94-105</range>
      <comment>Normal</comment>
      <status>Final results</status>
    </Result>
  - <Result num="4">
      <type>CO2</type>
      <value>27</value>
      <range>24-31</range>
      <comment>Normal</comment>
      <status>Final results</status>
    </Result>
  </Message>
```

If an error occurred, the Engine stores an event log in the Conversion Agent reports location, by default

```
c:\Documents and Settings\<USER>\Application Data\SAP\ConversionAgent\CMReports
```

where `<USER>` is your user name. To view a log, you can drag the `*.cme` file to the Events view of the Studio.

For more information about events logs, see the *Conversion Agent Engine Developer Guide*.

Points to Remember

To move a transformation from development to production, deploy it as a Conversion Agent service. The services are stored in the Conversion Agent repository.

You can run a service in Conversion Agent Engine by using the command-line interface, by API programming, by using the CGI interface, or by using integration tools.

INDEX

A

Acrobat
  parsing tutorial 35
  transformations 3
Acrobat files
  parsing 35
Acrobat Reader
  PDF files 36
actions
  computing totals 35
  tutorial 50
Adobe Reader
  PDF files 36
advanced properties 49
  viewing and editing 35
Alternatives
  anchor 52
ambiguities
  resolving 56
ambiguous source text
  resolving 70
anchor
  Group 63
anchors
  Alternatives 52
  Content 13, 62
  defining 17
  defining positional 40
  EnclosedGroup 65
  Marker 13
  RepeatingGroup 28
API
  COM example 92
  Engine 3
  tutorial 91
attributes
  XML 2

B

basic properties 49
  editing 35
binary data
  parsing 38
Binary Source
  view 11
blank project
  editing 86
boilerplate
  ignoring 35

C

child elements
  XML 2
CMW files
  project 10
COM API
  Visual Basic example 92
  Visual Basic source code example 92
Component
  view 11
components
  global 71, 73
Content
  anchor 13, 62
ContentSerializer
  serialization anchor 80
count
  of newlines 46
  using to resolve ambiguities 70

D

data holder
  meaning of 15
debugging
  parser 30
default transformers
  purpose 68
delimiters
  LearnByExample 69
  positional 35
  tab 13
deploying services
  tutorial 91
document processors
  example 39
  for Word documents 58
documents
  defined 2

E

editors
  Studio 10

elements
XML 2
EmbeddedSerializer
serialization anchor 82
empty elements
in XML output 63
EnclosedGroup
anchor 65
Engine 3
API tutorial 91
error
events 32
errors
correcting in parser configuration 17
event log
interpreting 31
Events
view 11
example
pane 11
example source
document 10
Excel
serialization tutorial 77
Explorer
view 11

F

failure
events 31
fatal error
events 32
folder
project 79
format
positional 35
tab-delimited 18
formatting
using for parsing 55

G

global components 73
IntelliScript 71
Group
anchor 63

H

hello, world
tutorial 7
Help
view 11
HL7
background 23
message structure 24
parsing tutorial 23
transformations 3
HTML
background information 56
converting Word documents to 55
generated by Word 58
removing code from output 68
HTML documents
transformations 3

I

information
events 31
installation
procedure 4
IntelliScript
correcting errors in 17
pane 11
IntelliScript Assistant
view 11
IntelliScript editor 11
opening 12
Internet Explorer
XML editor 5
invoices
parsing 36

L

LearnByExample
interpreting delimiters 69

M

Map
action 88
mappers
creating 86
definition 1, 85
tutorial 85
Marker
anchor 13
Microsoft Word
parsing documents 55
parsing tutorial 55

N

newlines
as separator of RepeatingGroup 46
searching up to 42

O

offset
Content anchor 41
optional property
of Group 63

P

parent elements
XML 2

parser
 running in Studio 18
parsers
 definition 1
 multiple in project 83
 testing and debugging 30
parsing
 beginner tutorial 7
 by example 1
 HL7 tutorial 23
PDF
 parsing tutorial 35
PDF files
 parsing 35
 transformations 3
 viewing 36
PDF to Unicode
 document processor example 39
PdfToTxt_3_00
 document processor example 39
perspective
 resetting 8
positional
 format 35
positional parsing
 tutorial 35
processors
 document 39
 for Word documents 58
 WordToHtml 55
project
 determining folder location 79
project properties
 in Studio 80
projects
 creating 25
 importing 9
properties
 project 80

R

recursion
 serializer calls 77
repeating groups
 nested 35, 44
RepeatingGroup
 anchor 28
RepeatingGroupMapping
 mapper anchor 88
RepeatingGroupSerializer
 serialization anchor 82
Repository view 11
requirements analysis
 for parsing 36
result files
 output 10

S

Schema
 view 11
schema
 XML 2
schema files
 XSD 10
schemas
 XSD 27
script files
 TGP 10
search scope 43, 44
 controlling 35
separator
 of repeating group 29
serialization anchors 77
 ContentSerializer 80
 defining 80
 EmbeddedSerializer 82
 RepeatingGroupSerializer 82
serializers
 creating 78
 definition 1
 multiple in project 83
 tutorial 77
service
 deploying 92
services 3
solutions
 to Getting Started exercises 4
source
 of Mapper 87
source documents
 testing additional 19
 testing parser on 73
startup component
 setting 18
Studio 2
 opening 8
system requirements
 installation 4

T

tab delimiters
 between text 13
tab-delimited
 format 18
target
 of Mapper 87
testing
 parser 30
 source documents 73
text files
 parsing tutorial 7
TGP
 script files 10
totals
 computing 50
transformation
 deploying as a service 92
transformers 69
 configuring 70
 default 68

Tutorial_1
    basic parsing techniques 7
Tutorial_2
    defining a parser 23
Tutorial_3
    positional parsing 35
Tutorial_4
    loosely structured documents 55
Tutorial_5
    defining a serializer 77
Tutorial_6
    defining a mapper 85
tutorials
    basic parsing techniques 7
    HL7 parser 23
    parsing HTML document 55
    parsing Word document 55
    positional PDF parsing 35
    running service in the Engine 91
Tutorials folder
    copying 4

V

valid XML
    schema 2
variables
    defining 60
views
    Studio 10
Visual Basic
    COM API example 92
vocabulary
    XML 2

W

warning
    events 32
web pages
    transformations 3
welcome page
    displaying in Eclipse 8
well-formed XML
    rules for 2
whitespace
    in XML 2
windows
    Studio 10
Word
    parsing documents 55
    parsing tutorial 55
WordToHtml
    processor 55, 58
WordToRtf
    processor 58
WordToTxt
    processor 58

X

XML
    overview 1
    standards 2
    tutorial 2
XML editor
    Internet Explorer 5
XPath
    defining anchor mappings 15
    extensions 16
XSD
    reference information 27
    schemas 27