Mercator®

Design Guide

Release 2.0

Mercator  nline Library

tsi soft

TSI International Software Ltd.

TSI Software Web Site

www.tsisoft.com

Corporate Headquarters
45 Danbury Road
Wilton, CT 06897-0840
Voice: 203.761.8600
Fax: 203.762.9677

Customer Service Centers
US/Canada 800.215.9633
847.444.0740

UK +44 (0) 171 393 8000
Australia +61 (0) 3 9820 2077
Singapore +65 225 0700
Hong Kong +852 2786 9991

Sales and Support
Bannockburn Lake Office Plaza
2345 Waukegan Road, Suite E100
Bannockburn, IL 60015
Voice: 847.317.9000
Fax: 847.317.9019

275 Madison Avenue, 24th Floor
New York, NY 10016
Voice: 212.683.0050
Fax: 212.683.0111

200 East State Street, Suite 202
Media, PA 19063
Voice: 610.892.7100
Fax: 610.892.7105

492 St Kilda Road
Melbourne, Victoria 3004
Australia
Voice: +61 (0) 3 9593 9399
Fax: +61 (0) 3 9593 9310

100 Walker Street
Level 12
North Sydney, NSW2060
Australia
Voice: +61 (0) 3 9593 9399
Fax: +61 (0) 3 9593 9310

Coveham House
Downside Bridge Road
Cobham
Surrey, England KT11 3EP
Voice: +44 (0) 193 257 6800
Fax: +44 (0) 193 257 6899

62 Queen Street
London, England EC4R 1AF
Voice: +44 (0) 171 314 9600
Fax: +44 (0) 171 314 9601

114 Rochester Row
London, England SW1P IJQ
Voice: +44 (0) 171 233 7144
Fax: +44 (0) 171 233 6931

33, rue Galilee
75116 Paris, France
Voice: +33 1 44 43 52 88
Fax: +33 1 44 43 52 99

Willy Brandt Platz 6
68161 Mannheim
Germany
Voice: +49 621 1594 164
Fax: +49 621 1594 200

Graadt van Roggenweg 328
P.O. Box 19127, 3531 AH
Utrecht, The Netherlands
Voice: +31 (0) 30 298 2269
Fax: +31 (0) 30 298 2178

12-04 Keck Seng Tower
133 Cecil Street
Singapore 06535
Voice: +65 220 1126
Fax: +65 220 6809

Research & Development
Peninsula Plaza, Suite 250
2424 North Federal Highway
Boca Raton, FL 33431
Voice: 561.394.3400
Fax: 561.394.3470

Publication Number 79507

Contents

Using the Design Guide

This *Design Guide* was created to help you learn Mercator through hands-on work. It includes a tutorial for learning the basics, and a number of examples, which you can duplicate on your own. Use this guide as a practical tool for learning Mercator.

Other Mercator Documentation

In addition, you should read the other Mercator Authoring System documentation, which includes the following:

Getting Started

Type Editor Reference Guide

Map Editor Reference Guide

Functions and Expressions Reference Guide

Using a Command Execution Engine

Execution Commands Reference Guide

Building and Using an Application Adapter

Type Tree Maker Reference Guide

The *Getting Started* manual covers installation of the Mercator Authoring System, and introduces the concepts of data objects and object-oriented mapping. The *Type Editor Reference Guide* contains detailed information on using the Type Editor, and the *Map Editor Reference Guide* has similar detail about the Map Editor. The *Functions and Expressions Reference* explains how expressions are evaluated, and lists each Mercator function with its syntax and examples. *Using a Command Execution Engine* discusses how to run a map on your system platform using a command Execution Engine. The *Execution Commands Reference Guide* explains execution commands and their options. *Building and Using an Application Adapter* explains how to build an application adapter and how to use it as a source or destination for a map. The *Type Tree Maker Reference Guide* covers how to use the Type Tree Maker to create a type tree. You may want to refer to one or more of these manuals as you work through the *Design Guide*.

Using Mercator

There are three basic steps in using Mercator:

1. The first thing you do is define your data to Mercator, in the Type Editor.
2. Next, in the Map Editor, you tell Mercator how to map your data.
3. Then, you use an Execution Engine for your system platform to actually map the data.

"Here's the definition of my data" **"Here's how to map my data"** **"Now, map my data"**

Mercator Examples

Mercator comes with examples, that are installed when you run **Setup**. Each example includes data file(s), type tree(s), and a map source file.

The examples are located in the "*Examples*" directory (folder in Windows 95), under the directory where you installed Mercator. Within the *Examples* directory, is the directory "*general*." Some of the examples in the *general* directory are explained in this guide.

Each example is located in a directory with an appropriate name. For example, the map that uses the EXIT function is in the *Examples\general\exit* directory.

Other Examples in This Guide

In addition to explaining some of the examples that come with Mercator, this guide documents other examples. These examples show you how to use some of the Mercator functions. They also explain common mapping methods that you may want to use.

Chapter 1 – Mercator Tutorial

This chapter guides you through a simple example that teaches the basics of using Mercator. Using a data file supplied with Mercator, you follow step-by-step instructions to create your own type tree and map. Before you begin working through this example, you should understand the basic concepts of data objects and mapping, which are explained in the Mercator *Getting Started* manual.

What You Want to Do

Suppose you have a simple file of just one record. This record contains information about one of your customers. The record includes the name of the contact person at the company, the company name, the address, and the phone number.

Input Data:

```
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,60525,708,3525555
```

From this file, you want to generate a mailing label for that customer.

Output Data:

```
James P Adams
ABC Co.
29 Frankford Rd
Bloomington, IL 60525
```

How to Do It

First you need to define the contact data and the label data in a type tree. Next, define how you want to transform the contact information into a label. To do this, create a map in the Map Editor. Then, build, or compile, the map. Finally, run the map to generate the output data.

Files Used in This Chapter

The following table contains the input file and the files to create when working through the tutorial.

File	Use
contact.txt	Use as an input data file. It is located in your *mercator\examples* directory (folder in Windows xx).
address.mtt	Create this type tree file.

mail.mms Create this map source file.

label.txt Running the completed map creates this output file.

Creating a Type Tree

Define the data in a type tree. Create two type trees–one for the input, and one for the output. Or, create one type tree that defines both the input and output. It does not matter whether you create one or two type trees. Mercator's performance is not affected. One advantage of creating a single tree is that the data objects that appear in both the input and the output are in one place. If you plan to create mailing labels from a variety of different input sources, you might decide to create one tree for input, and a separate tree for output.

Thinking about the Input

If you think about the input data, and describe it, you might say, "The file is made up of just one contact record. The contact record is made up of certain fields."

You need to consider which data objects will be defined as Items, and which ones will be defined as Groups. Simple data objects will be Items. Complex data objects will be Groups.

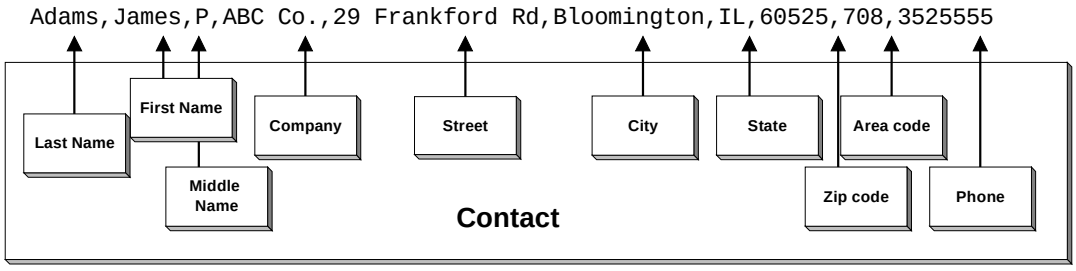
Which objects are simple?

All of the fields are simple data objects. They do not have other data objects inside of them. Therefore, each field will be defined as an Item.

Which objects are complex?

The contact record is complex–it is made up of fields. It is a Group. In this case, contact records comprise the entire file.

If you look at the input data again, you see many data objects. Each field is a data object. The contact record itself is a data object. You need to name each data object.



Notice that a comma is used to separate the fields, and although you cannot see it here, there is a carriage return/linefeed (CR/LF) at the end of the record. The comma and CR/LF are syntactical objects. They are used to separate one field from another, and tell when a record ends.

Thinking about the Output

If you think about the output data, and describe it, you might say, “The file is made up of just one mailing label. The label is made up of four fields, each on a separate line.”

What objects are simple?

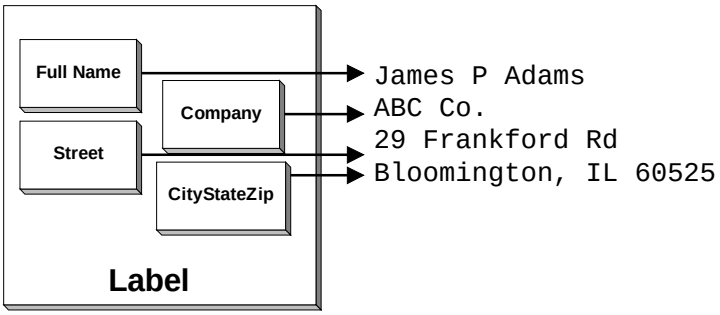
Each field in the label is considered a simple text object. Simple objects are defined as Items.

Even the last line in the address—city, state, and zip code—is considered one text Item.

What objects are complex?

The label is made up of multiple fields. Therefore, it is considered complex. Complex objects are Groups. Therefore, the label is a Group.

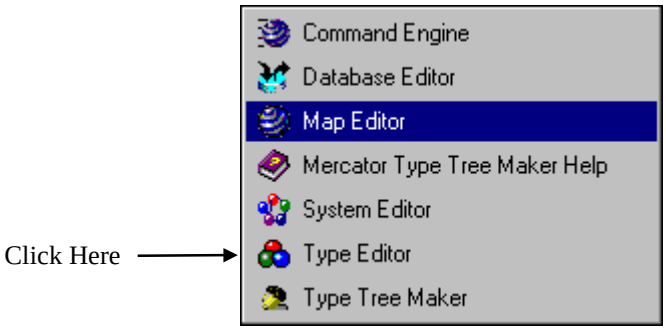
Look at the output data, and name the data objects.



Using the Type Editor

It's time to do some hands-on work. Start the Mercator Type Editor:

Click **Start**, then point to **Programs**, point to the Mercator group, and click the Type Editor icon.

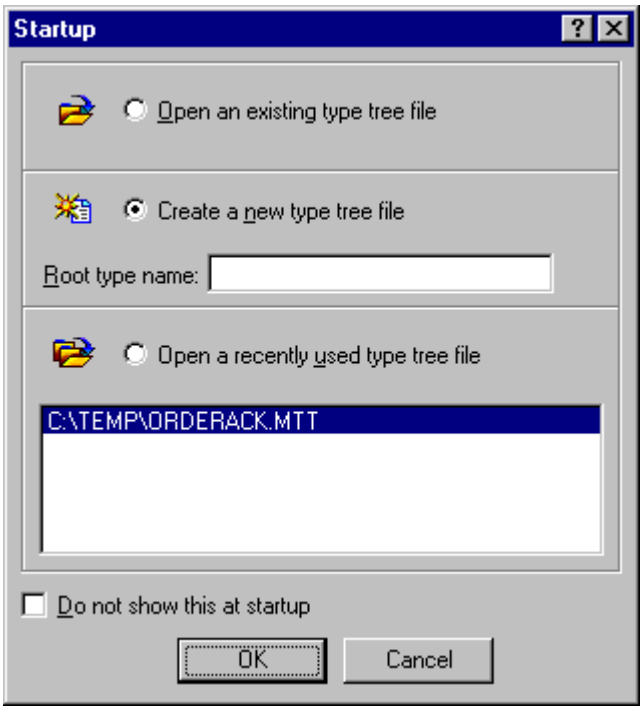


The first thing you need to do is to create a new type tree. The type tree's purpose is to describe the data to be used as input, and the data you want to create—the output. You are going to describe both the input and output data in the same type tree.

To Create a New Type Tree

1 When you open up the Type Editor, as shown above, you will be given the option of opening up an existing tree or creating a new one.

The Startup dialog is displayed.

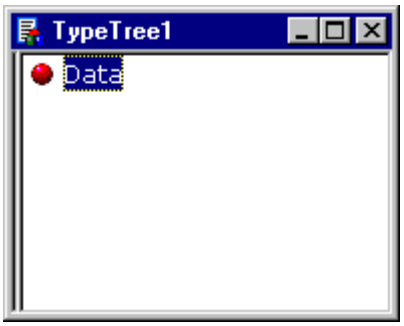


Now, enter the name of the root type. Name the root type *Data*.

2 In the Root type name box, enter *Data.*

3 Click **OK**.

Now, a type tree window is displayed. The root type is *Data*. The root type has a red icon. The red icon means it is a Category.



The next thing you should do is save the type tree file.

To Save the Type Tree

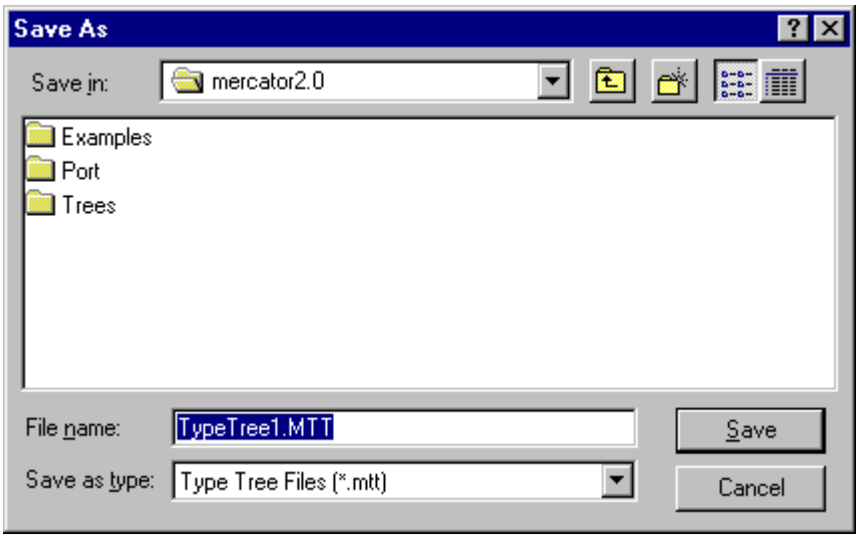
1 From the **File** menu, select **Save**.

Or, click the **File Save** tool. 

The Save As dialog is displayed.

Save the type tree file in the *Examples* directory (folder in Windows 95 or 98), because this is where the data file is located. This makes things easier later on.

2 Double-click the *Examples* directory.



The contents of the *Examples* directory is displayed.

3 In the **File Name** field, type over *TypeTree1.MMT*, and enter *address*.

Mercator automatically adds the extension (.mtt) to the type tree file name.

4 Click **OK**.

The name of the type tree file, *address.mtt*, is displayed in the title bar of the type tree.

Next, create the types.

The order in which you create types does not matter, in general. However, for this tutorial, follow along step by step.

Creating Group Types

The Group types are Contact and Label. The *Contact* Group represents the input file. The *Label* Group represents the output file.

So, the Group types to create are:

- Contact
- Label

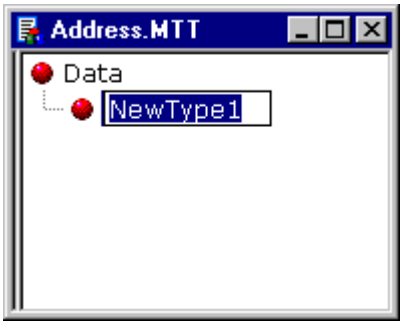
To Create the Contact Type

1 From the **Type** menu, select **Add**.

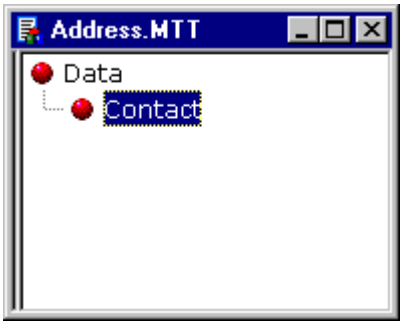
Or, click the **Add Type** tool. 

Or, press the INSERT key.

The New Type is displayed in your tree. The default name is NewType1.



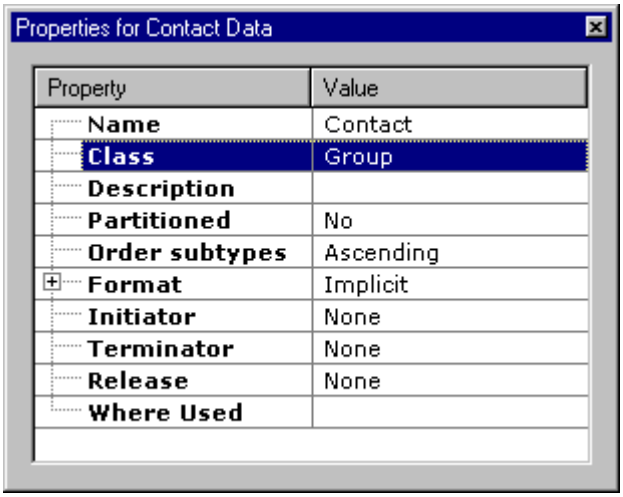
2 Type over NewType1 and enter *Contact*.



Now, specify that *Contact* is a Group. To do that, bring up the Properties grid for this new type. Do this with a right-mouse click on the new type and then select Properties.

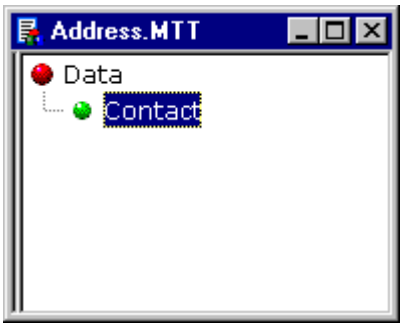
Or, click the **Properties** tool. 

3 In the Class section, click on Category. From the drop-down list, select Group.



4 Click **OK**.

Contact should now have a green icon, indicating that it is a Group type.



To Create the Label Type

When you create a type, it is displayed beneath the type that is highlighted. Make sure *Data* is highlighted when you create *Label*. This ensures that *Label* is placed underneath the root type, *Data*.

1 Select the type *Data*.

2 From the **Type** menu, select **Add**.

Or, click the **Add Type** tool. 

Or, press the INSERT key.

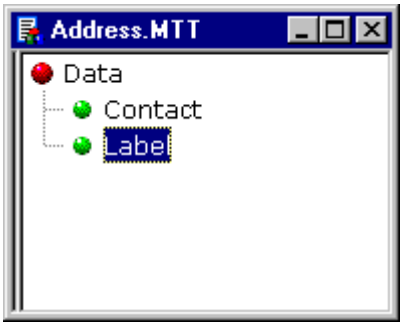
The New Type is displayed in your tree. The default name is NewType1.

3 Type over NewType1 and enter *Label*.

4 Bring up the properties grid and in the Class section, select Group.

5 Click **OK**.

The type *Label*, with a green icon, stems down from the root.



Now, create the Item types.

Creating Item Types

The *Item* types are all of the fields in Contact and Label. Some of the fields that appear in Contact are also in Label. The fields in both are the Company and Street fields. Define these just once, and re-use them.

Here is a list, in alphabetical order, of the Item types to create:

- AreaCode
- City
- CityStateZip
- Company
- First Name
- Full Name
- Last Name
- Middle Name
- Phone
- State
- Street
- ZipCode

Organizing Types

Sometimes, you may want to organize similar types in the same area of the type tree. Create a type and put related types underneath it.

Create an Item named *Field*, and put all of the field types underneath it. In addition to organizing similar types, another advantage of creating a *Field* Item is that all of the types created beneath it are automatically created as Items. When you do this, there is no need to select Item from the Class section when you create each field type.

To Create the Field Type

1 Select the root type, *Data*.

2 From the **Type** menu, select **Add**.

Or, click the **Add Type** tool. 

Or, press the INSERT key.

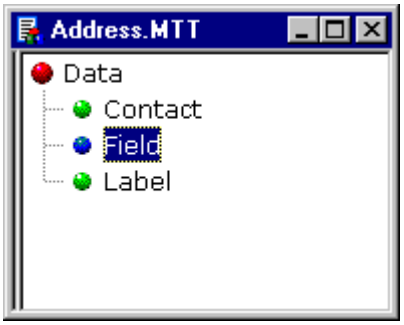
The New Type is displayed in your tree. The default name is NewType1.

3 Type over NewType1 and enter *Field*.

4 In the Class section of the Properties grid, select Item.

5 Click **OK**.

The Field type, with a blue icon, now stems off the root.



Note: All types are displayed in alphabetical order, from top to bottom.

Using the Type Tree Inheritance

When types have common properties, take advantage of the type tree's Inheritance feature. When a type tree is created, it inherits the properties from the type above it. If some types have common properties, create a type and define the common properties for it. Then create all the types that have those properties underneath it. That way, there is no need to define the same properties multiple times.

The following table contains the Items' properties. These properties include Interpretation, Minimum size, Maximum size, Justification, and Pad Character.

Item	Interpretation	Min	Max	Justification	Pad Character
AreaCode	Text	3	3	Left	<space>
City	Text	2	<none>	Left	<space>
CityStateZip	Text	10	<none>	Left	<space>
Company	Text	3	<none>	Right	<space>
First Name	Text	1	<none>	Left	<space>
Full Name	Text	1	<none>	Left	<space>
Last Name	Text	1	<none>	Left	<space>
Middle Name	Text	1	<none>	Left	<space>
Phone	Text	7	7	Left	<space>
State	Text	2	2	Left	<space>
Street	Text	2	<none>	Left	<space>
ZipCode	Text	5	5	Left	<space>

Notice that the name fields have the same properties. The common properties of the name fields are:

- Interpretation – text
- Minimum size – 1
- Maximum size – none
- Justification – left
- Pad Character – space

Create a type called *Name*, and assign to it the properties that all the name types have. Then, when the name types are created, they automatically inherit these properties. There is no need to assign them for each one.

To Create the Name Type

1 Select the *Field* type.

From the **Type** menu, select **Add**.

Or, click the **Add Type** tool. 

Or, press the INSERT key.

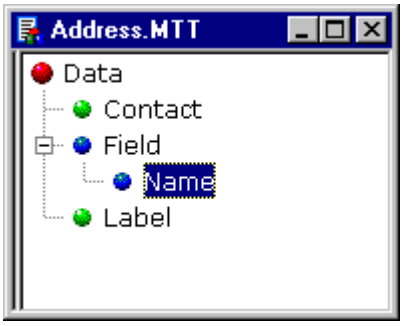
The New Type is displayed in your tree. The default name is NewType1.

2 Type over NewType1 and enter *Field*.

If you pull up the Properties grid for the *Field* type, you will see that Item is already chosen in the Class section. This is because a type beneath an Item must be an Item. Mercator does not allow you to select any other class for this type.

3 Click **OK**.

The *Name* type is displayed beneath the *Field* type:



To Define Properties of the Name Type

Now, define the properties of the *Name* type.

1 Select the *Name* type.

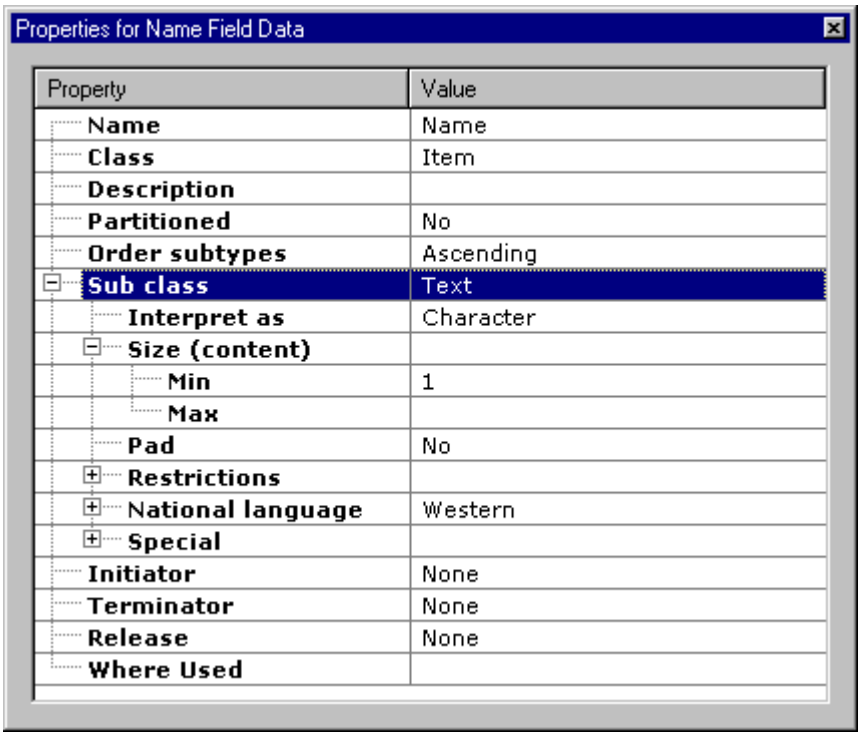
2 Bring up the Properties grid for the *Name* type.

Right-mouse click the new type and then select **Properties**.

Or, click the **Properties** tool. 

Or, hold down the ALT key and press the ENTER key.

The Properties grid is displayed.



Most of the properties that were passed down from the root to *Field* to *Name* type should be kept. You do not have to change the settings on Interpret as (text), or Pad.

The only change to make is the Minimum size.

1 In the Size section, enter *1* in the **Min** box.

2 Press the ENTER key.

Now, each type you create under the Name type inherits the Name's properties. There are four name data objects—First Name, Last Name, Middle Name, and Full Name. Create four types under the Name type, and call them First, Last, Middle, and Full. You created *subtypes* of Name. A subtype is a type that stems beneath a given type.

To Create the Name Subtypes

1 Select the *Name* type.

From the **Type** menu, select **Add**.

Or, click the **Add Type** tool. 

Or, press the INSERT key.

The New Type is displayed in your tree. The default name is NewType1.

2 Type over NewType1 and enter *First*.

Notice that Item is already selected in the Class section of the Properties grid.

The type First inherited the properties of the type Name. There is no need to define the properties for First. It is already done! To look at the properties, select First and then bring up the Properties grid.

To Look at Properties of First

1 Select the *First* type.

2 Bring up the Properties grid for the *First* type.

Right-mouse click the new type and then select Properties.

Or, click the **Properties** tool. 

Or, hold down the ALT key and press the ENTER key.

The Properties grid is displayed. Note that the properties for the *First* type are already defined.

Now that you have created one of the Name subtypes, define the rest.

Create the Remaining Name Subtypes

To create the rest of the Name subtypes, follow the instructions under "To Create the Name Subtypes,". In step 3, enter a different type name.

Follow the instructions three more times—once for each remaining subtype. The remaining subtypes are:

- Last
- Middle
- Full

Now, create the remaining Items under the Field type.

Create the Remaining Field Subtypes

Create the remaining subtypes of Field:

- AreaCode
- City
- CityStateZip
- Company
- Phone
- State
- Street
- ZipCode

To Create Other Field Subtypes

Here are instructions for creating the remaining Field subtypes. In step 3, the word **typename** stands for the particular name of the type, as they are listed above. For example, the first time you follow these instructions, enter *AreaCode* as the **typename**. The next time, enter *City*.

1 Select the *Field* type.

From the **Type** menu, select **Add**.

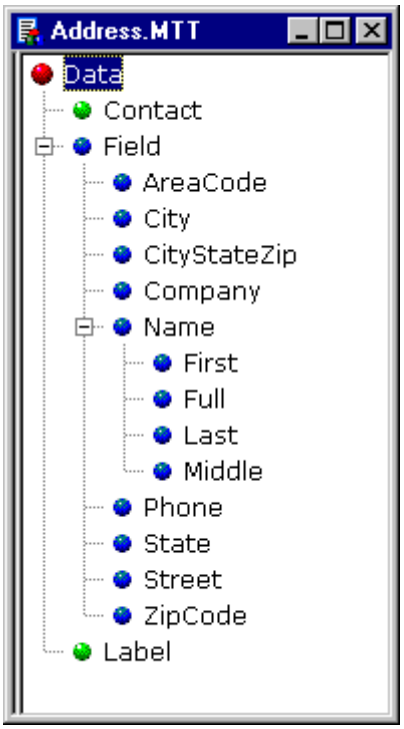
Or, click the **Add Type** tool. 

Or, press the INSERT key.

The New Type is displayed in your tree. The default name is NewType1.

2 Type over NewType1 and enter *typename*.

When finished creating the Field subtypes, the tree should look like this:



All Field types in the Contact and Label data are defined. The Contact and Label types are defined. There are no other types to define.

The input and output types are now all defined in the type tree.

Defining Components

Now, define the *components* of the Group types.

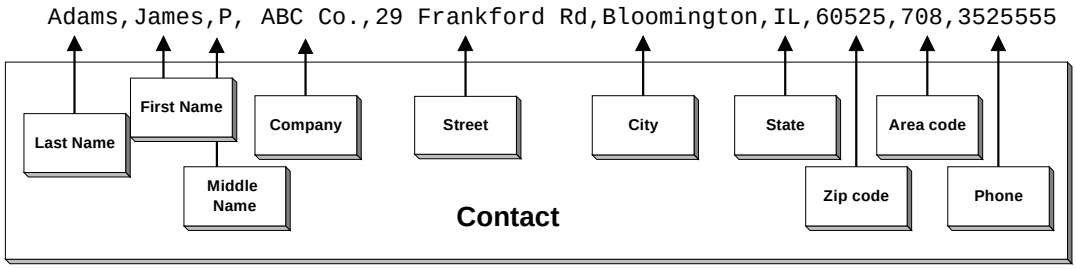
A *component* is a data object that is part of a complex data object. For example, the data object First Name is part of the data object Contact, so First Name is a component of Contact. The data object CityStateZip is part of the data object Label, so CityStateZip is a component of Label.

Remember that Group types are complex—they are made up of other objects. Therefore, they have components. In contrast, Item types are simple. They are *not* made up of other objects. So, Item types *do not* have components.

Components of Contact

When defining components of a Group, tell Mercator what data objects make up the Group. Mercator also needs to know the order of data objects in the data stream.

To determine the order of Contact's components, look at the data again.



This diagram shows the components of Contact. Each field is a component of Contact. By reading the diagram left to right, the order of the fields is clearly seen.

Now, make a list of the components of Contact, in their data stream order—their order in the data.

Components of Contact

- Last Name
- First Name
- Middle Name (0:1)
- Company
- Street

- City
- State
- ZipCode
- AreaCode
- Phone

The first component in the list is Last Name Field. The next component is First Name Field, and so on.

The Middle Name component has a range—(0:1). In a range, the first number in the parentheses is the minimum number of consecutive occurrences of this component. The second number is the maximum number of consecutive occurrences of this component. There may be between zero and one occurrence of Middle Name in the data. Middle Name does not have to appear at all; it is optional.

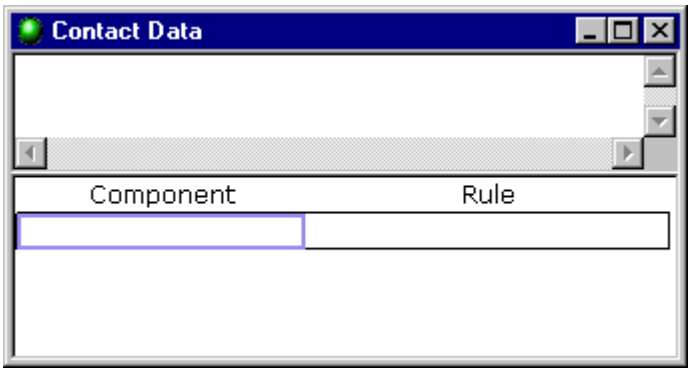
To Define the Components of Contact

1 Double-click the type *Contact*.

Or, select the type *Contact*, and press the ENTER key.

The component window of Contact is displayed. The complete name of the type is displayed in the title bar of the component window. The complete name begins at the type and includes the names of the types on the path up to the root type. The complete name of this type is:

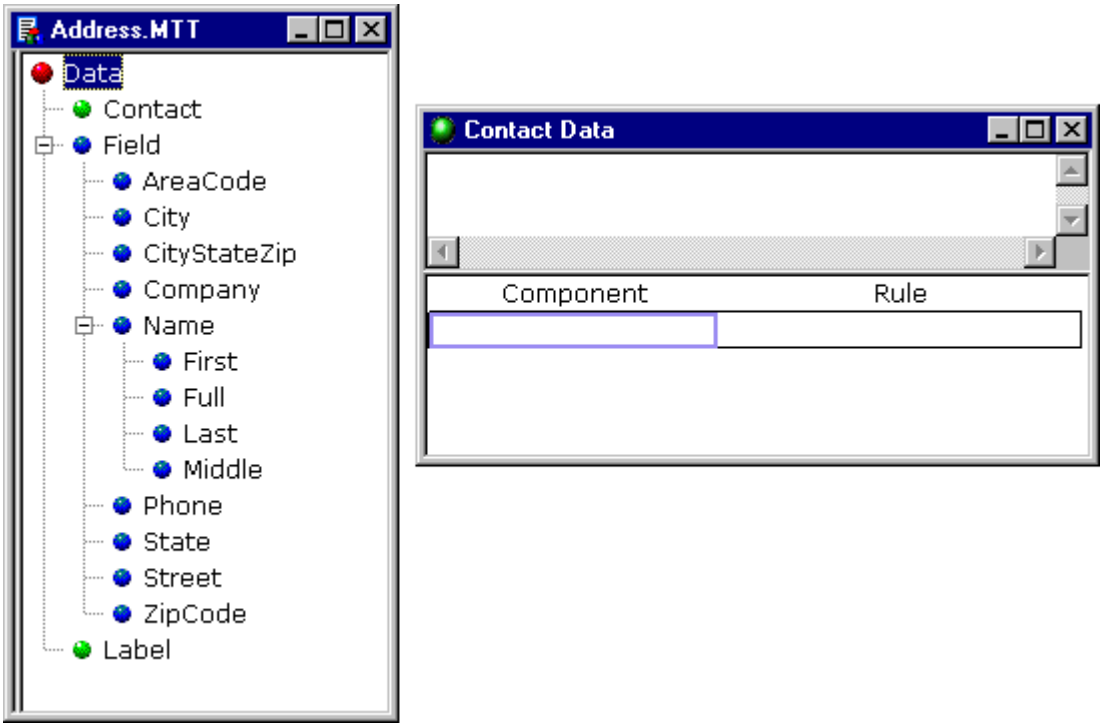
Contact Data



Note To define components, drag and drop types from the type tree into the component window. To do this properly, make sure the two windows—the type tree window and the component window—do not overlap.

2 Use the Tile command, in the Window menu to arrange the type tree window and the component window so that they do not overlap.

You might arrange your windows to look like this:

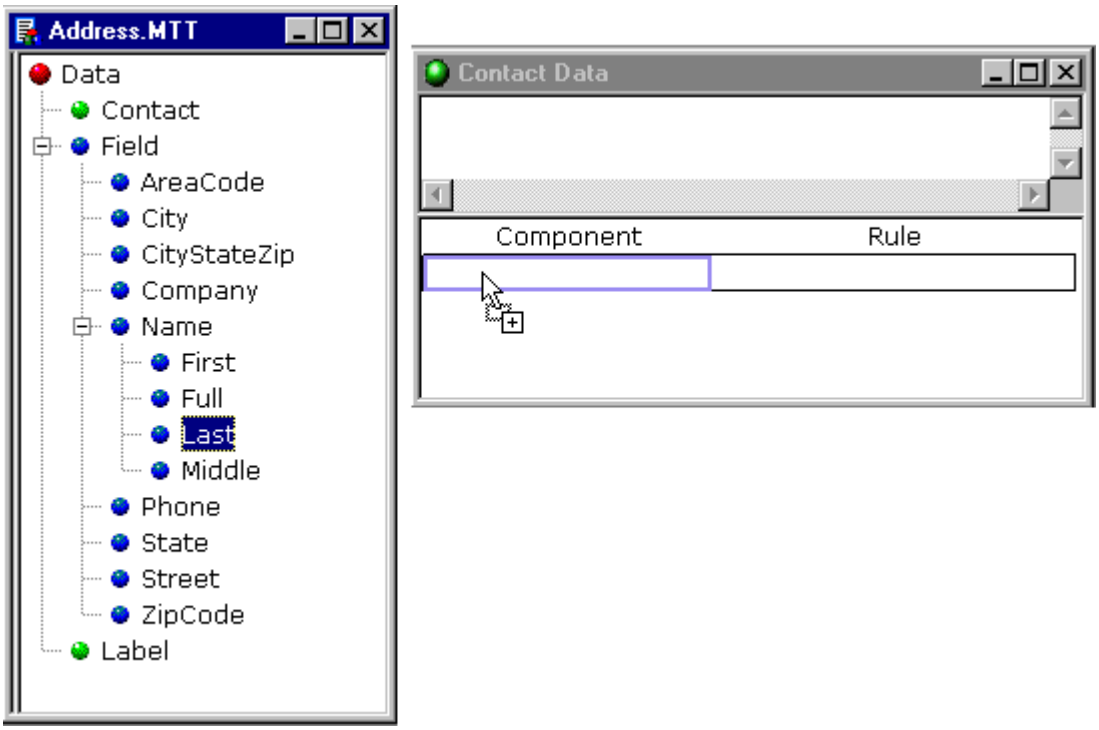


In the component window, there are two columns—the Component column, and the Rule column. Move the line that separates the two columns. Right now, work with the Component column. Move the line so that only the Component column is in sight. To do this, select it with the mouse, and drag it over.

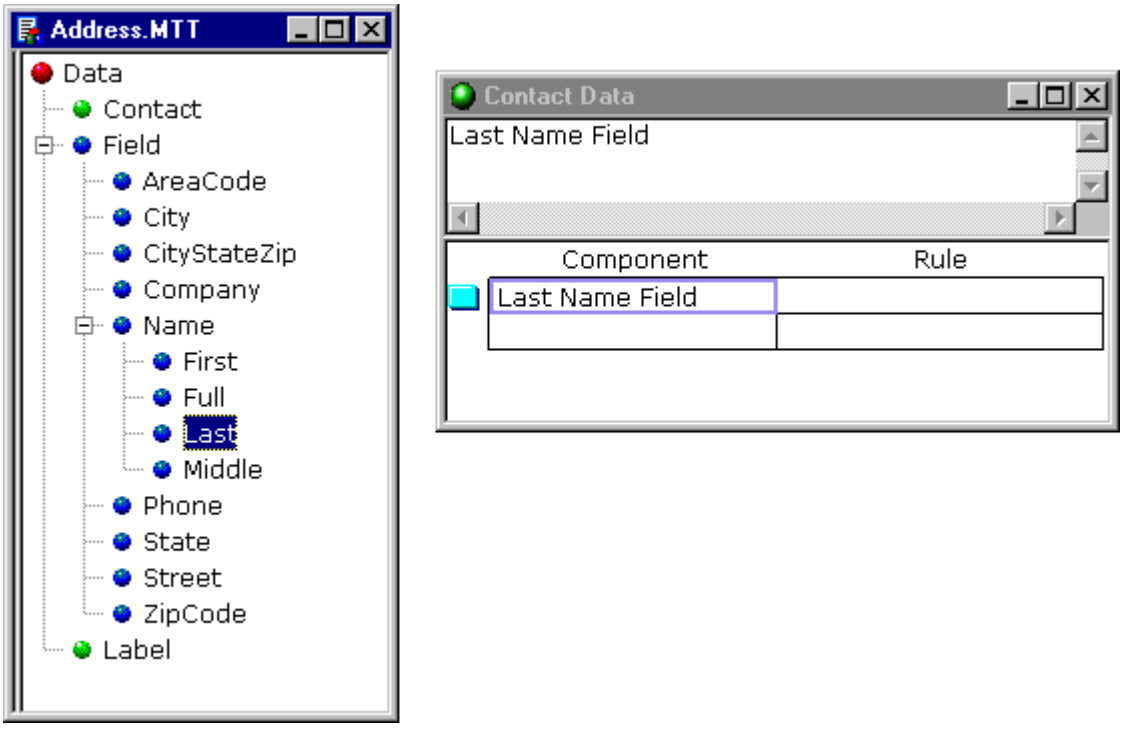
3 Move the column separator to the right, to increase the size of the Component column.

4 Drag and drop the type Last from the tree into the component cell.

To select the type, select any part of its name, or its icon.



After you drag and drop the type, you see the component name, Last Name Field, in the component cell. This name is a relative type name. It reflects the location of the component type, with respect to the type being defined.



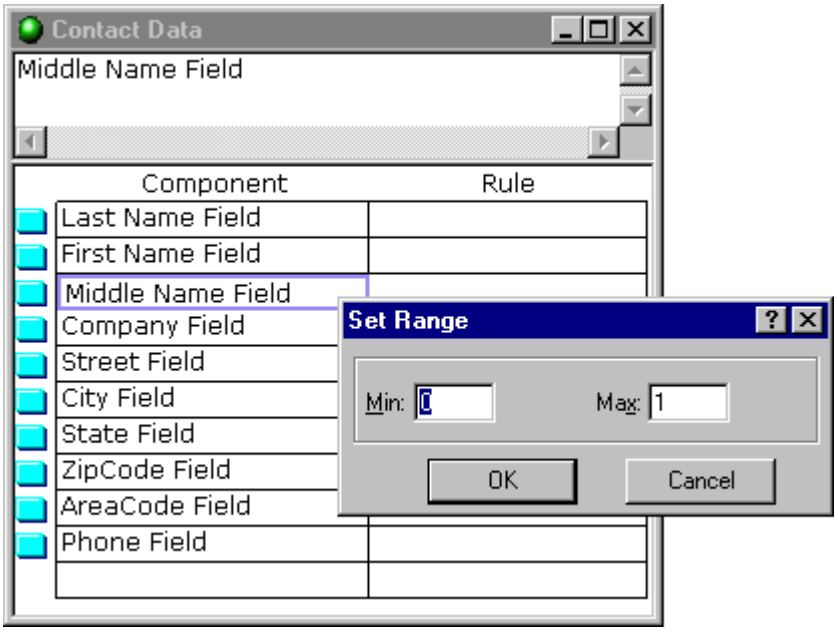
Now, define the rest of the components by dragging and dropping the component types into the component window.

Look at the list of Contact's components. The component after Last Name is First Name. So, First Name is the next component to define. Drag and drop the First item into the component cell beneath Last Name Field. Continue to define the rest of the components this way, until all of the components of Contact are defined.

Each time a component is added to the window, a new cell is created. The cells may scroll up, and seem to disappear. To see these components, use the scroll bar on the right side of the component window, or make the component window bigger.

To Add Component Range for Middle Name Field

1 Select the component Middle Name Field. Notice that the selected component appears in the rule bar, as shown below.



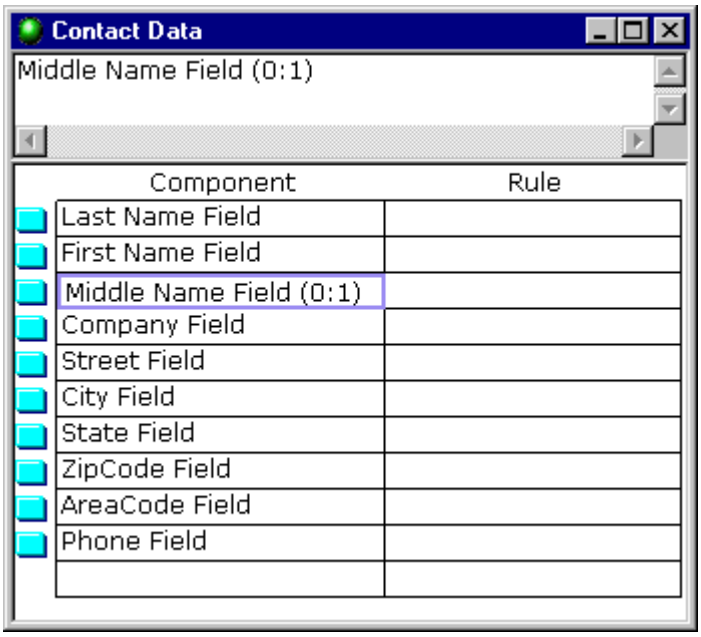
The component name is displayed in the rule bar.

2 With the mouse, click in the rule bar to the right of the component name.

3 Type a space, and then **(0:1)**.

4 Press the ENTER key.

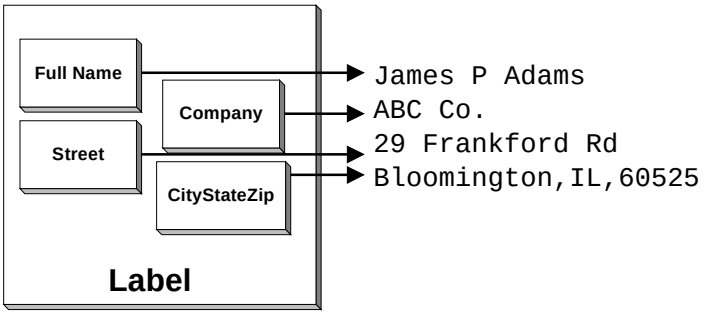
When the components of Contact are defined, the component window should look like this:



When you close the component window (by clicking on the X in the upper right corner), you will be prompted to save the changes you have made.

Components of Label

Look at the diagram of Label.



Make a list of its components, in their order of appearance in the data:

- Full Name
- Company

- Street
- CityStateZip

Defining the Components of Label

1 Double-click the type *Label*.

Or, select the type *Label*, and press the ENTER key.

The component window of Label is displayed.

2 Arrange the type tree window and the component window so that they do not overlap.

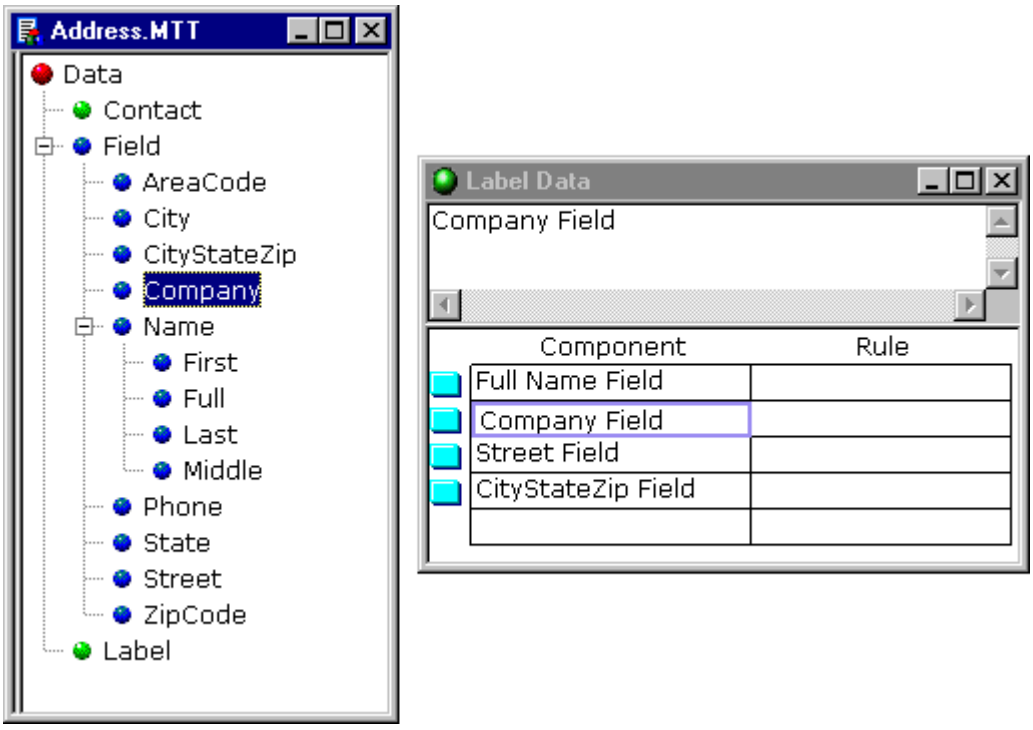
3 Move the column separator to the right, to increase the size of the Component column.

4 Drag and drop the type Full from the tree, into the component window.

The Full Name Field, is displayed in the component window.

Define the rest of Label's components by dragging and dropping the component types from the tree.

When finished, the component window of Label should look like this:



Next, define properties of the types.

Defining Item Properties

To define properties of the Items, look at the table of properties in the Using the Type Tree Inheritance section. The table gives you the values to enter for each property. Go down the list of types and define the properties for each one—except for the Name types, because you already defined these.

Note: If the Max size is <none>, leave the Max box blank.

The hex value **20** is a space.

Many of the properties of the Items in this data are the default properties. When the Properties dialog is displayed, you see that these values are already specified, and you do not have to change them.

To Define Properties of Each Item

1. Select the Item whose properties you want to define.
2. Bring up the Properties grid for the *Name* type.

Right-mouse click the new type and then select Properties.

Or, click the **Properties** tool. 

Or, hold down the ALT key and press the ENTER key.

The Properties grid is displayed.

3. In the Interpret as list, select the Item's interpretation.
4. In the Size section, enter the minimum size in the Min box.
5. Enter the maximum size in the Max box.
6. In the Pad section, enter the pad character in the box. If it is a non-printable character, select Hex from the View As list, and enter the hex value in the box.
7. Save changes.

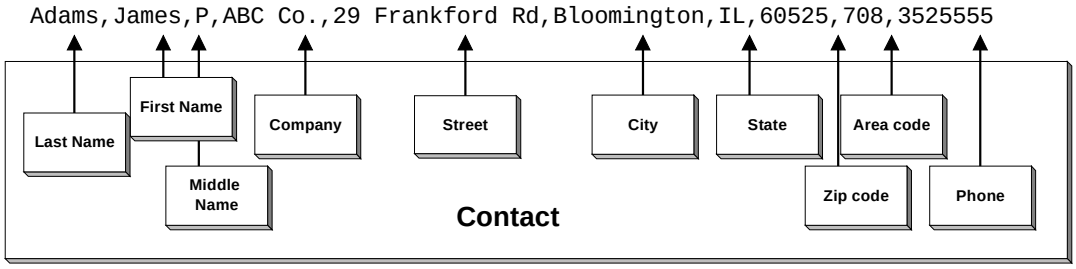
Defining Group Properties

Each Group has a certain format. Some Groups are made up of a fixed number of components, and each component has a fixed size. These Groups have a *Fixed* format. Some Groups have a delimiter

separating the components. These Groups have a *Delimited* format. Finally, some Groups are neither *Fixed* nor *Delimited*, but their components appear in a certain pattern. These Groups have an *Implied* format.

Properties of Contact

Look at the diagram for Contact. The comma is used as a delimiter. So, Contact has a Delimited format.



When a Group is delimited, specify where the delimiter is displayed. Sometimes the delimiter is displayed before each component—this is called *Prefix*. Sometimes the delimiter is displayed after each component—this is called *Postfix*. When the delimiter is displayed between components, but not before the first or after the last component, this is called I*nfix*.

Because the comma is displayed between components, but not before the Last Name field, and not after the Phone field, the location of the delimiter, in this example, is *Infix*.

In addition to the comma delimiter, Contact has another syntax object—the CR/LF, which is displayed at the end of Contact. It cannot be seen because it is a combination of non-printable characters. The CR/LF is defined as the *Terminator* of Contact. A *Terminator* is a syntax object that is displayed at the end of a type.

To Define Properties of Contact

1 Select the type *Contact*.

2 Bring up the Properties grid for the *Contact* type.

Right-mouse click the new type and then select Properties.

Or, click the **Properties** tool. 

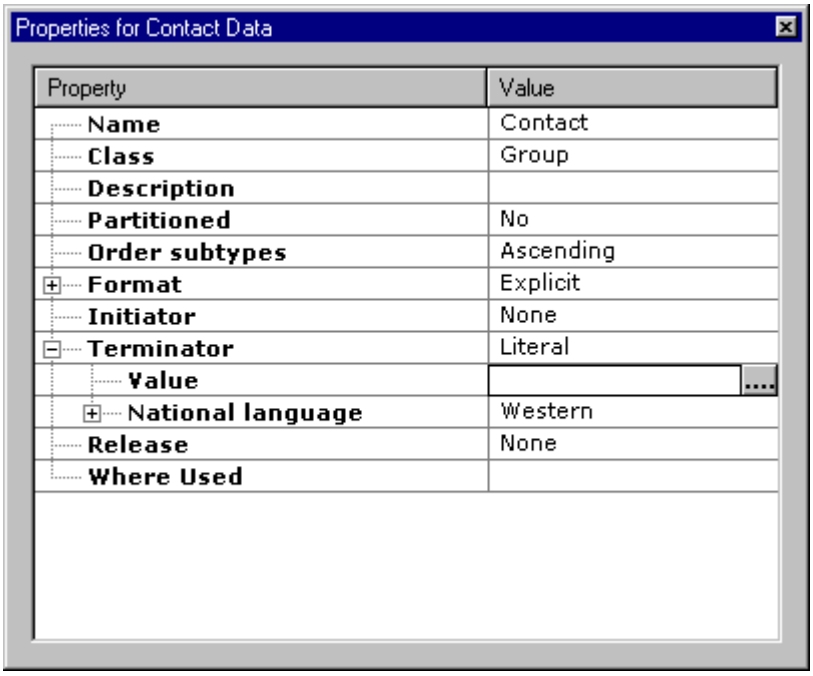
Or, hold down the ALT key and press the ENTER key.

The Properties grid is displayed.

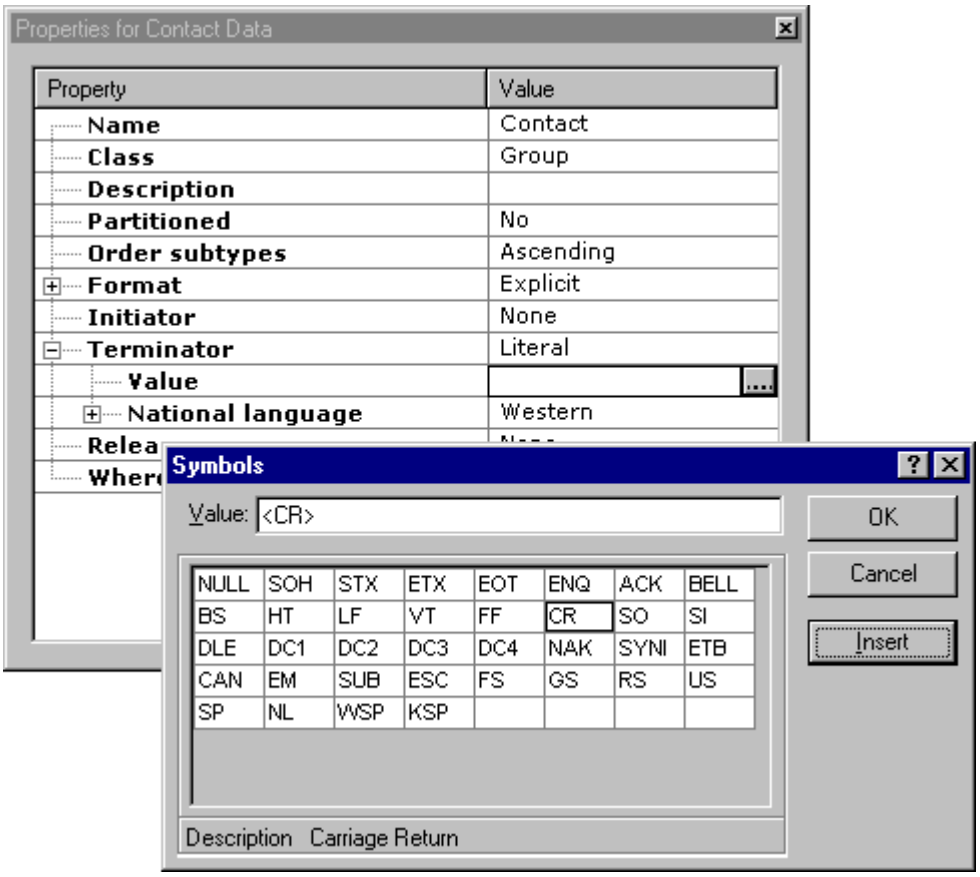
3 In the Group Format section, select the value Explicit.

4 Click the Terminator field and select *Literal* from the drop-down list.

5 Click on the Value field, and you will see a Browse button. ... Click on this Browse button, and the Symbols window is displayed.



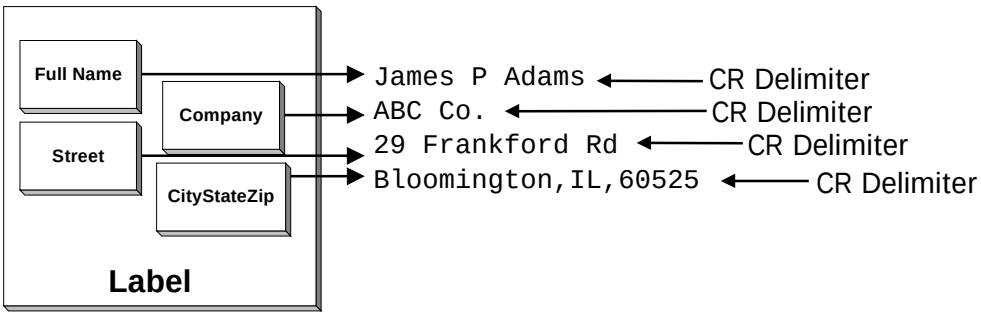
6 In the Symbols window, select *CR*. This is the value for Carriage Return.



7 Click **OK**.

Properties of Label

Look at the diagram of Label. Each field is on a separate line. To make this happen, you define a carriage return to appear at the end of each field. The CR is the syntax object that separates the components of Label. Therefore, CR is defined as the delimiter of Label.



Because the CR delimiter is displayed after each component, the delimiter location is *Postfix*.

Later, you will produce a file of many labels. A blank line separates the labels. To do this you want Mercator to put a new line at the end of each label. You will do this by inserting the NL function, for new line.

To Define Properties of Label

1 Select the type *Label*.

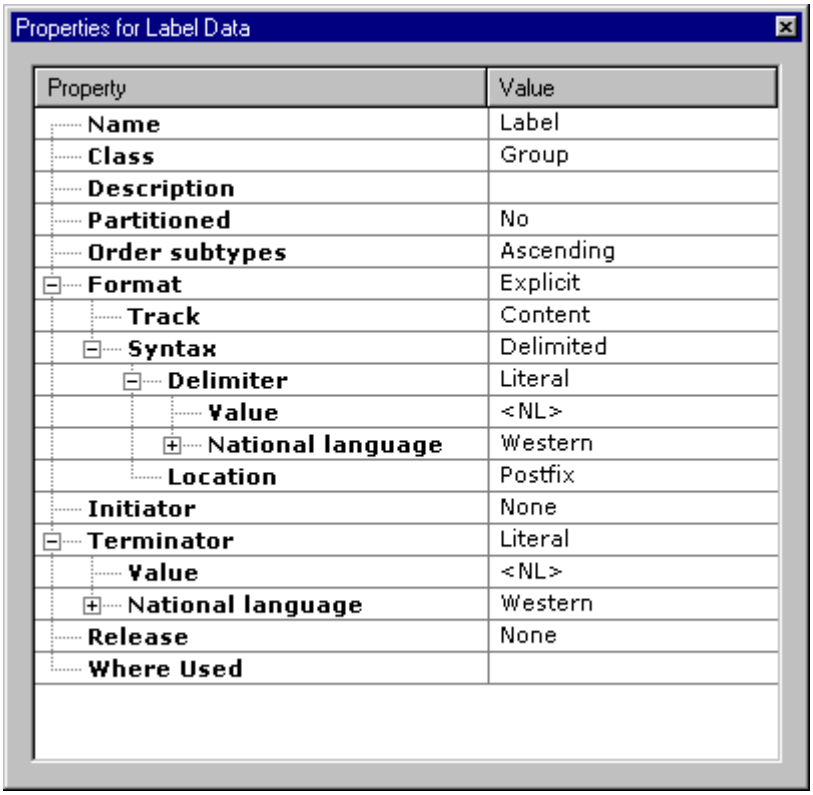
2 Bring up the Properties grid for the *Label* type.

Right-mouse click on the new type and then select Properties.

Or, click the **Properties** tool. 

Or, hold down the ALT key and press the ENTER key.

The Properties grid is displayed.



3 In the Format field, select the value Explicit.

4 In the Syntax field, select the value Delimited.

5 In the Value field, click on the Browse button and enter <NL>. This is the value for a new line.

6 From the Location field, select the value Postfix.

7 In the Terminator field, select the value Literal.

The type tree is now finished. You created all the types, defined components, and defined properties. Next, double-check that you have not made any mistakes.

Analyze the Type Tree

When you finish creating a type tree and defining all the types, always analyze the type tree. The Type Tree Analyzer reveals if you defined types inconsistently, or failed to define the aspects of the types that allow Mercator to recognize data objects.

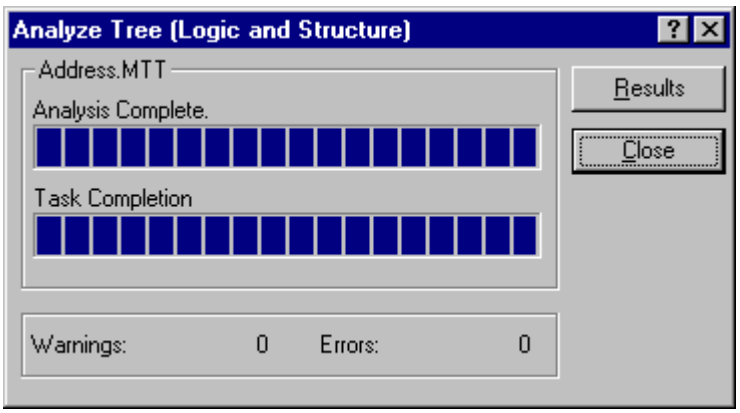
Now, analyze the tree.

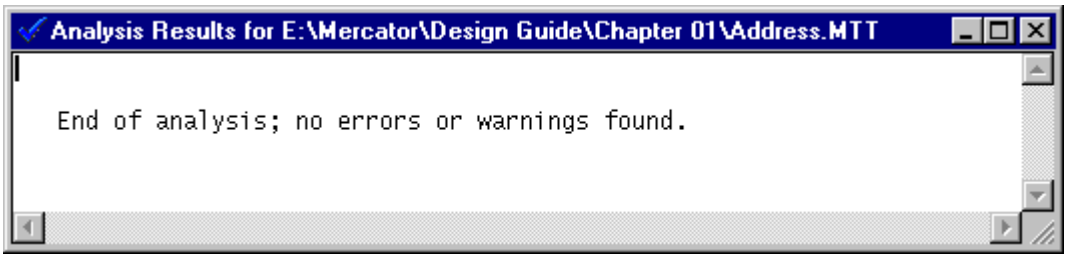
To Analyze the Type Tree

1 From the **Tree** menu, select **Analyze,** then, select **Logic and Structure**.

2 Click **OK**.

Mercator quickly goes through all of the analysis tasks. If your tree is error- and warning-free, the dialog looks like this:





3 If your tree is error free, Click **Close**.

Did you get errors? You can fix them and then analyze the tree again.

If You Have Errors

If you have errors, look at the error messages and warnings from the Type Tree Analyzer.

1 In the Analyze Tree dialog, click the Results button. The Results window is displayed.

The Results window includes the analysis errors and warnings. Here is an example of an error in the Results window:

For help in resolving errors see the topic, “Analyzer Error and Warning Messages,” in Chapter 15 – The Type Tree Analyzer, of the Mercator *Type Editor Reference Guide*.

2 Fix the errors.

3 Analyze the tree again.

Save the Type Tree Again

When your type tree is error-free, save it again.

To Save the Type Tree

1 From the **File** menu, select **Save**.

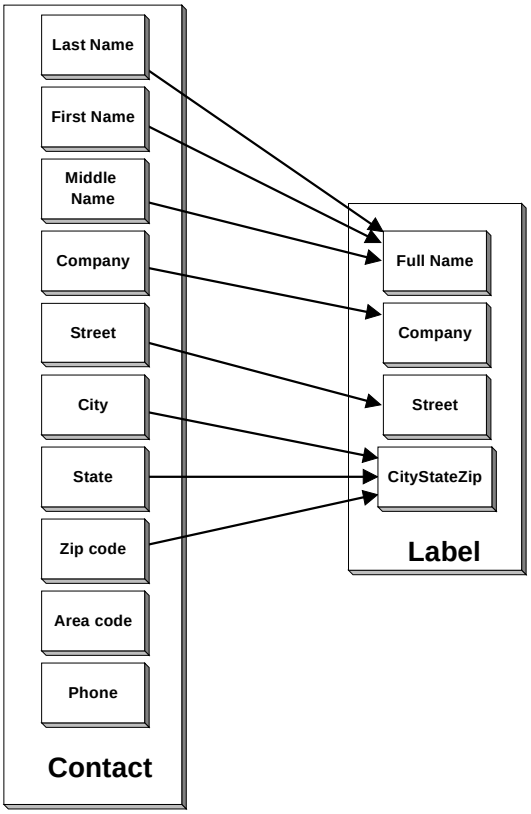
Or, click the **File Save** tool. 

Now that you have a type tree, use the Map Editor to create a map.

Creating a Map

Use the Map Editor to create a map. A map defines how to create a data object of a specific type. The data object you want to create is the mailing label. The map tells Mercator to generate the label by using certain data objects from the contact record.

Here is a diagram of how to generate the Label data object from the Contact data object:

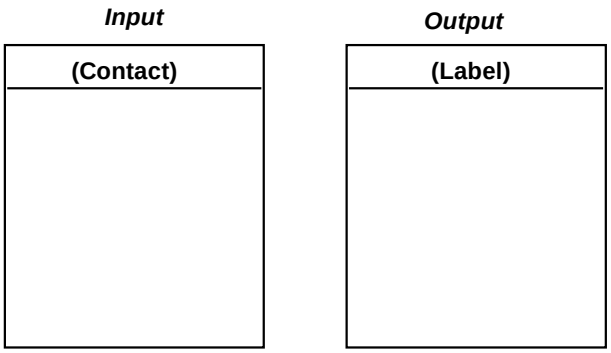


Map Cards

In the Map Editor, you create *cards* to represent data objects. Each card stands for one data object that has been defined as a particular type. Output cards represent output data objects. Input cards represent input data objects.

You want Mercator to generate one data object of the type Label. Therefore, create an output card to represent this data object. You want Mercator to use one data object of the type Contact, to generate the Label. Therefore, create an input card to represent this data object.

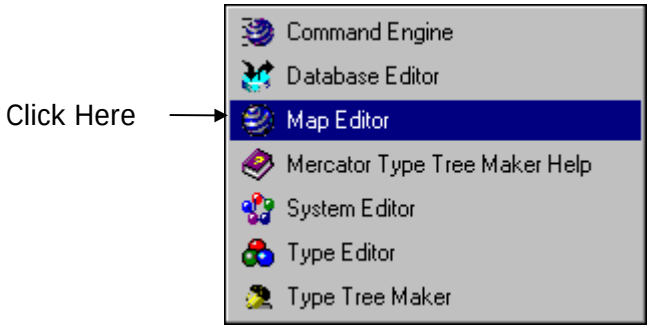
Your map consists of these cards:



In the output card, you enter map rules that tell Mercator how to generate the label.

Using the Map Editor

Start the Map Editor by clicking Start, and then the Map Editor icon in the Mercator Group.



Save the Source File

When you open the Map Editor, you see an empty file. This is the map source file. The name of this file is initially *untitled*. You need to save this file with a new name.

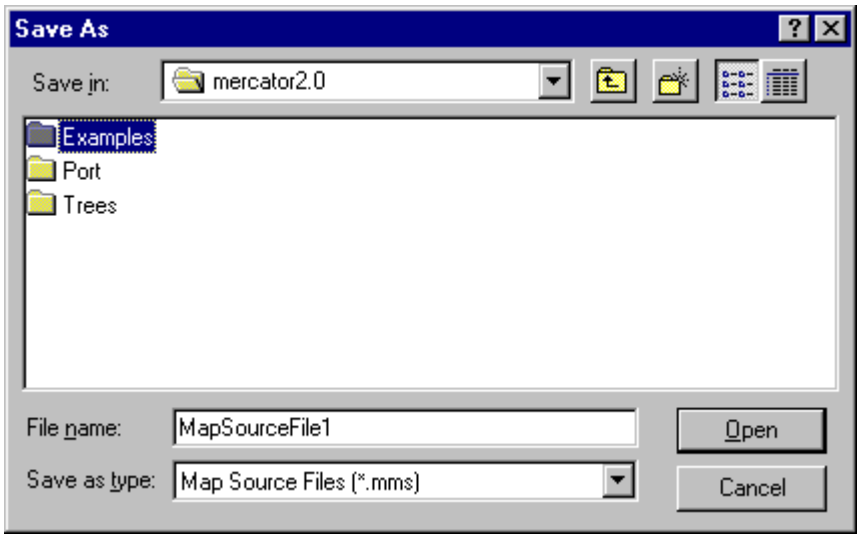
1 Click on the **New Map File** tool. 

The Save As dialog is displayed.

Save the source file in the *Examples* directory. This is where the data file is located. This makes things easier later.

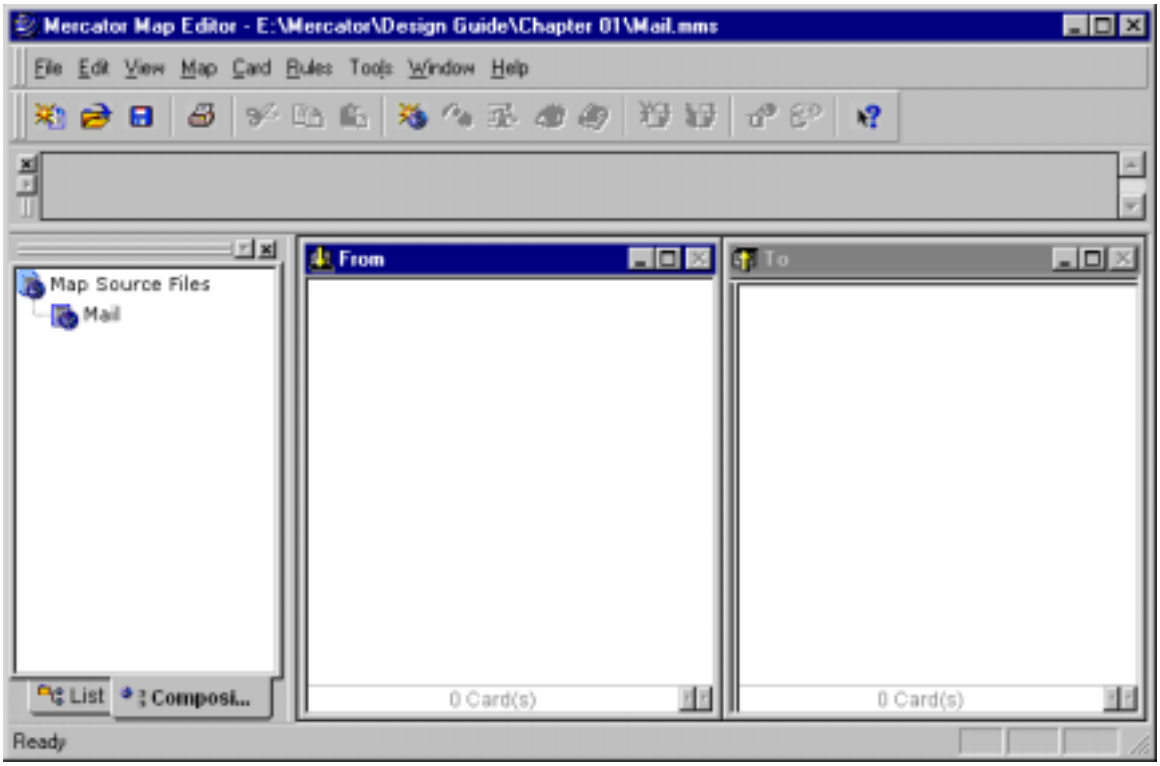
2 Double-click the *Examples* directory.

3 In the File Name box, type over the default name *MapSourceFile1* and enter *mail*.



4 Click **OK**.

Mercator adds the file name extension, **.mms**, which stands for *Mercator Map Source*. Now the file name—*mail.mms*—is displayed in the title bar of the Map Editor. See the example on the following page.



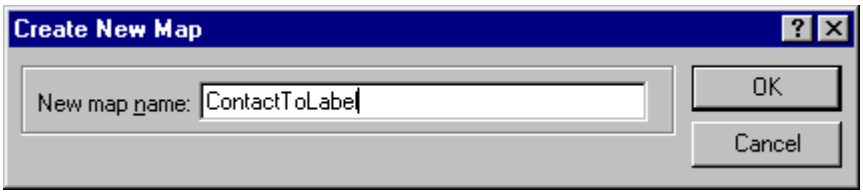
Rename the Map

Currently, the map you are building has no name. You need to name it.

This map transforms one Contact to one Label, so name it *ContactToLabel*.

1 From the **Map** menu, select **New**.

The Create New Map window is displayed.



2 In the New Map Name box, enter *ContactToLabel*.

3 Click **OK**.

Now the name of the map is displayed in the title bar, right next to the file name you created earlier.

Create Map Cards

Now, create the cards to represent the input and output data objects.

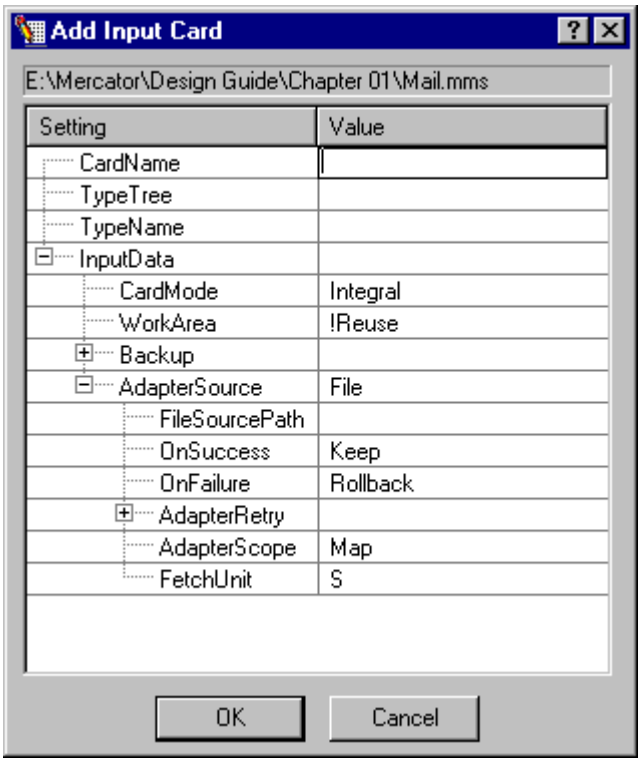
To Create the Input Card

1 Select the **From** window.

2 From the **Card** menu, select **New**.

Or, click the **Add Card** tool. 

The Add Input Card dialog is displayed.



3 In the **Card Name** field, enter the value *Contact*.

The name of each card in a map must be unique. Mercator uses the card name to reference the data object of the card.

4 In the value field for Type Tree, click the Browse button.

Each card represents a data object of a particular type. Select the type tree where that type has been defined. The type of the input card is Contact. You defined this in the type tree *address.mtt*.

The Select Type Tree dialog is displayed.

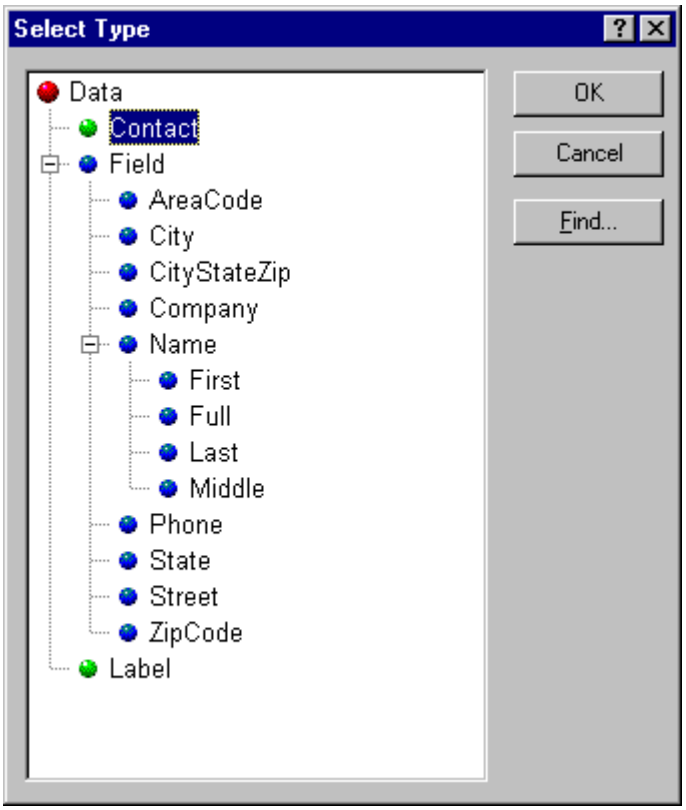
5 Select the file **address.mtt**.

6 In the value field for Type Name, click the browse button to display the browse dialog.

The Browse dialog shows the type tree, as it is displayed in the Type Editor. If the entire tree is not displayed in the window, use the scroll bar to view other parts of it.

8 Select the type *Contact*.

You are pointing Mercator to the particular type that defines this input data object—*Contact*.



9 Click **OK**.

The complete name of the type—*Contact Data*—is displayed in the value field for the Type Name setting.

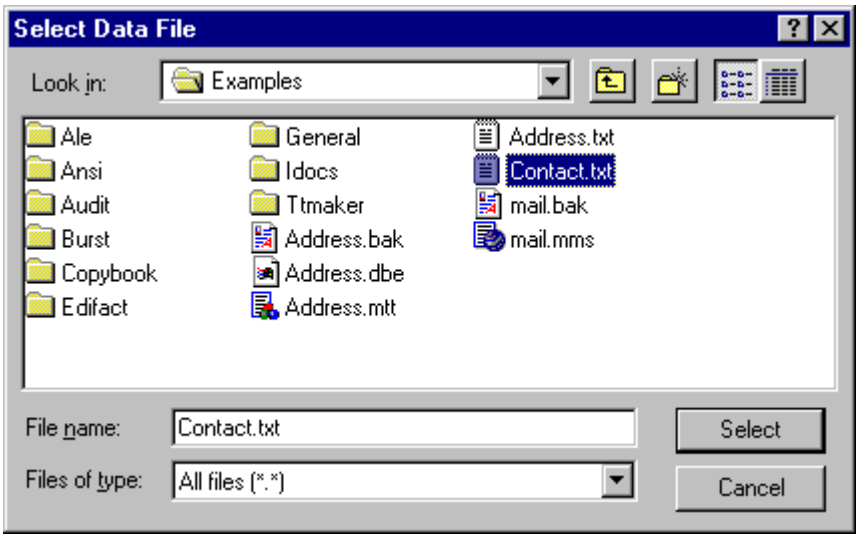
The Data Source is a File.

10 In the AdapterSource value field, select File, if necessary (File is the default setting).

You need to tell Mercator where the input data is located. The data file, *contact.txt*, is one Contact data object.

11 In the FileSourcePath value field, click the Browse button.

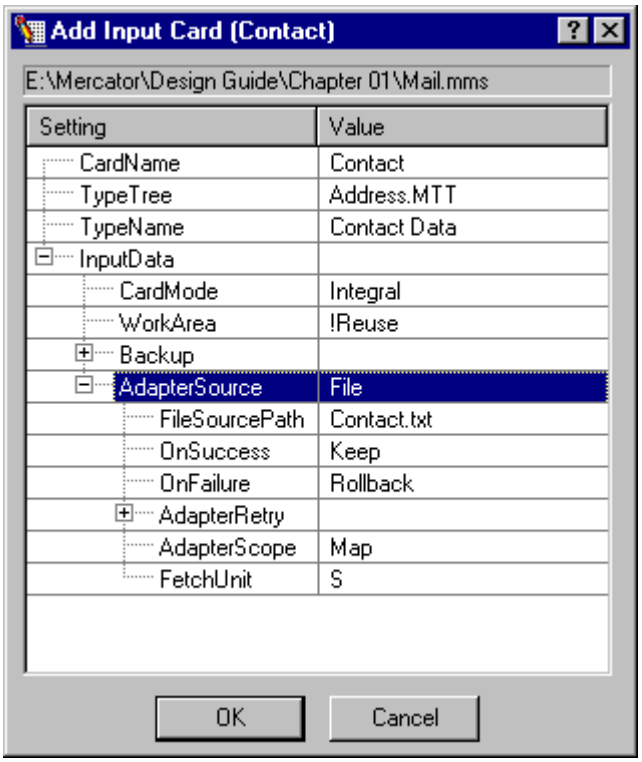
The Select Data File dialog is displayed:



12 Select the file **contact.txt**.

13 Click **Select**.

When you finish defining the card, the Add Input Card dialog should look like this:



14 Click **OK**.

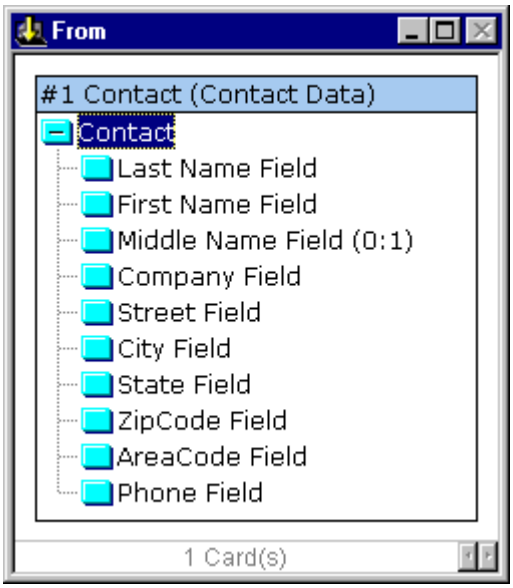
In the From window, the card you just created is displayed. The bottom of the window indicates that there is *1 Card* in this window.

The card number is displayed in the top left corner of the card. Input cards and output cards are numbered consecutively.

The name of the card is displayed in two places—in the top left corner of the card, and next to the top icon of the card.

The type of the card is displayed in the parentheses.

Click the top icon to expand. The components of Contact are displayed. When you expand a type that has a **+** in a rectangular-shaped icon, you are showing the components of that type. A type with a rectangle icon that cannot be expanded (it has a **–** in the icon) has no components, so it is an Item.



Now create the output card to represent the Label data object.

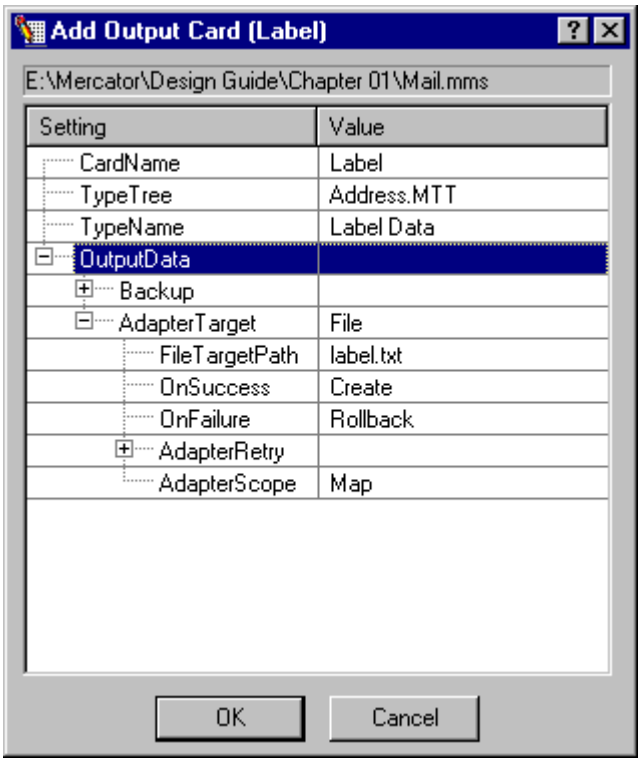
Create Output Card

1 Select the *To* window.

2 From the **Card** menu, select **New**.

Or, click the **Add Card** tool. 

The Add Output Card dialog is displayed.



3 In the Card Name field, enter *Label*.

4 In the TypeTree section, click the browse button.

The Select Type Tree dialog is displayed.

The type Label was defined in the type tree **address.mtt**.

5 Select the file, **address.mtt**.

6 Click **OK**.

7 In the TypeName field, click the browse button.

The browse dialog is displayed, showing the type tree address.mtt.

8 Select the type *Label*.

You are pointing Mercator to the particular type that defines the output data object—Label.

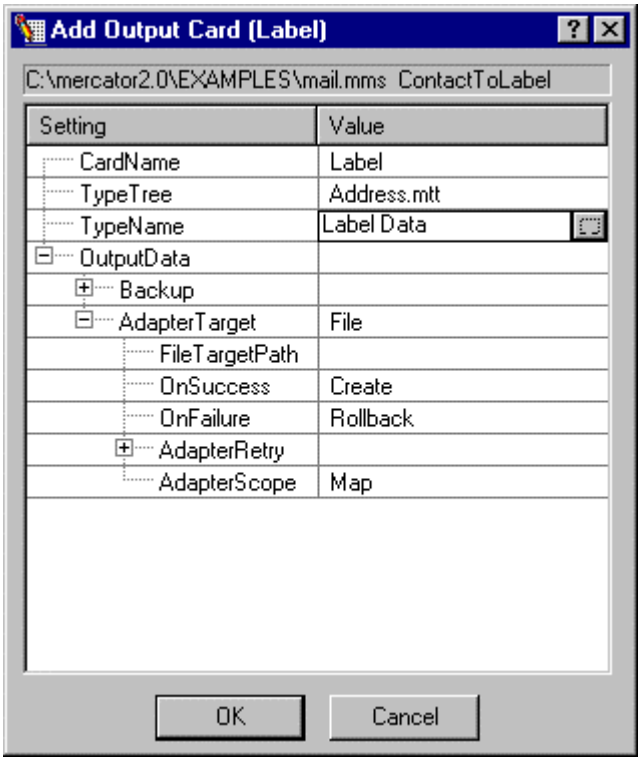
9 Click **OK**.

The complete name of the type—Label Data—is displayed in the Name box.

10 In the File section, enter **label.txt** in the Name box.

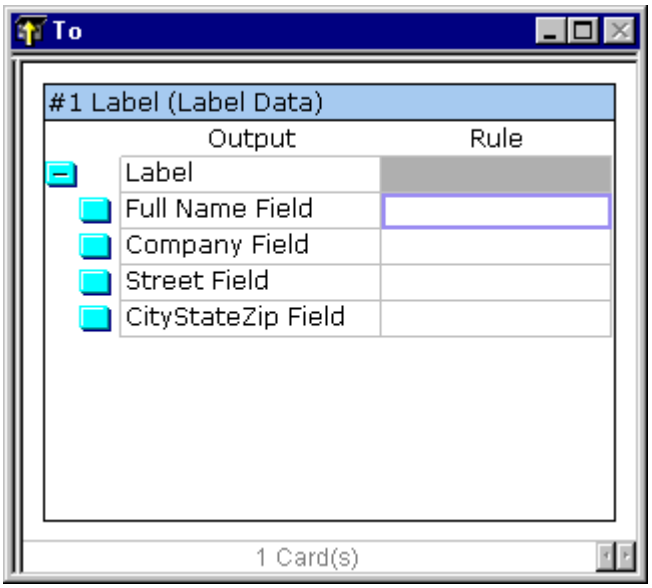
You are telling Mercator to create a data file named label.txt.

When finished, the Add Output Card dialog should look like this:



11 Click **OK**.

The output card you created is displayed in the To window.



Expand the top icon of the output card to see the components of Label:

The data objects shown when the type Label is expanded are the same as the Label's component list that was created in the address.mtt type tree.

Notice that the output card looks similar to the input card. However, the output card has two columns—*Output* and *Rule*.

Next, you are going to enter map rules in the cells of the *Rule* column.

Enter Map Rules

A map rule tells Mercator how to generate a particular data object. Each component of Label has a rule cell. In a given component's rule cell, you enter a map rule, telling Mercator how to generate that component. For example, the rule you enter for the output, Full Name Field, tells Mercator how to generate the Full Name Field data object.

You enter a map rule for each component of Label—starting with the simplest map rule, and then the more complex ones.

Each map rule begins with an equal sign **(=)**.

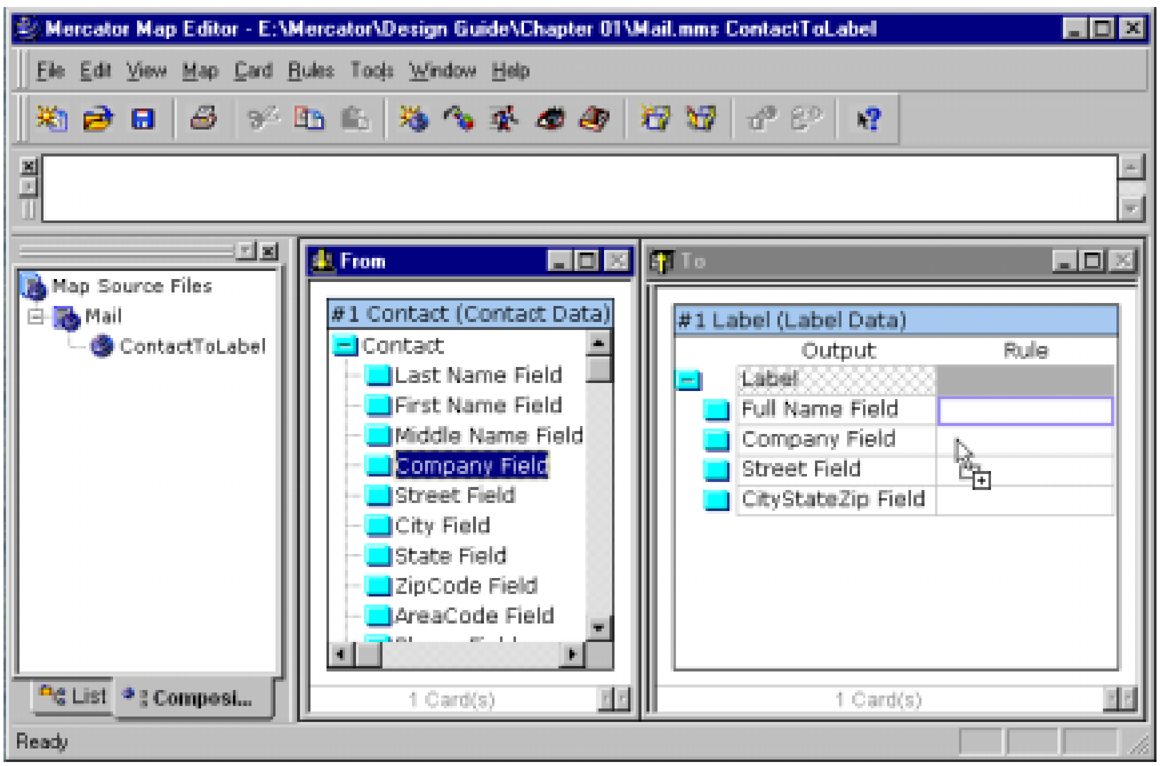
Mapping to the Company Field

You want the Company Field of Label to look like the Company Field of Contact. Therefore, map the Company Field from the input into the output.

```
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,60525,708,3525555
                    |
                    |
                    |          James P Adams
                    +--------->ABC Co.
                               29 Frankford Rd
                               Bloomington, IL 60525
```

To Enter Map Rule for Company Field

1 Select *Company Field* in the input card.



2 Drag and drop *Company Field* on the input card, into the rule cell of *Company Field* on the output card.

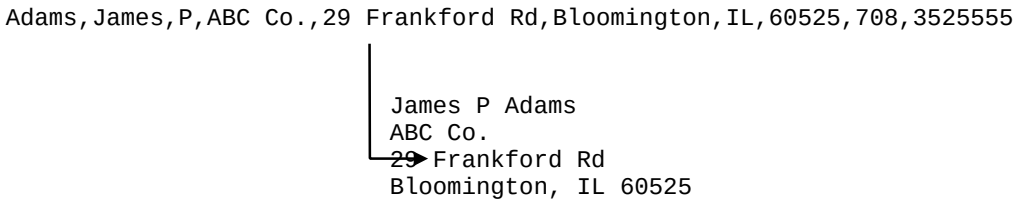
An equal sign is displayed in the rule bar, at the beginning. When you drag and drop an object into a rule cell, Mercator inserts the equal sign automatically.

The *Company Field:Contact* name is displayed in the rule bar. The rule bar shows the entire contents of the rule. In the output card—as shown in the picture of the previous Map Editor screen—the rule is only partly visible.

The colon **(:)** represents a component. The name, Company Field:Contact, means "the Company Field component of Contact." The name represents the path of the object up the card to the top icon, or card name.

Mapping to Street Field

You want to map the Street Field in Label from the Street Field in Contact.



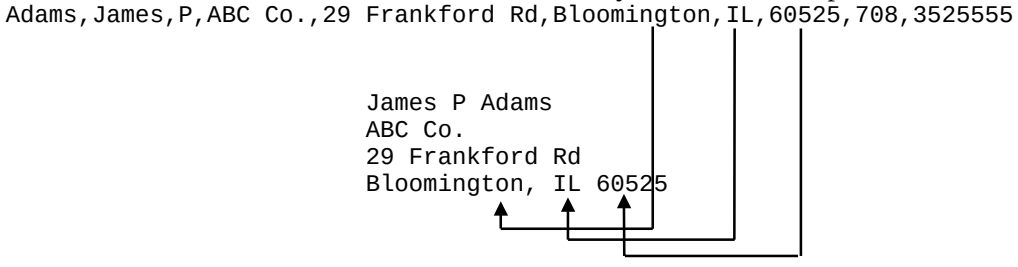
To Enter Map Rule for Street Field

1. Select the object *Street Field* on the input card.
2. Drag and drop it into the rule cell for *Street Field* on the output card.

The object name, Street Field:Contact, is displayed in the rule bar.

Mapping to CityStateZip Field

You want the last line of the Label to include the city, state, and the zip code.



You want a comma and a space between the city and state, and another space between the state and zip code.

You want the CityStateZip Field to look like this:

```
Bloomington, IL 60525
```

The map rule you enter is a text concatenation, including the City, State, and ZipCode fields from Contact, spaces, and a comma. To concatenate text, use the plus sign **(+).** Literal text values are enclosed in double quotes **(“ ”)**.

To Enter a Map Rule for CityStateZip Field

1 Select *City Field* on the input card.

2 Drag and drop *City Field* into the rule cell of *CityStateZip* Field.

You see =*City Field:Contact* in the rule bar.

3 Click in the rule bar, to the right of City Field:Contact, and type a *space*.

This space is to make the rule easier to read. You can enter spaces around object names, and operators. These spaces do not affect the evaluation of the rule.

4 Type + *“, ”* +

5 Drag and drop the *State Field* from the input card *up* into the rule bar.

6 Type + " " +

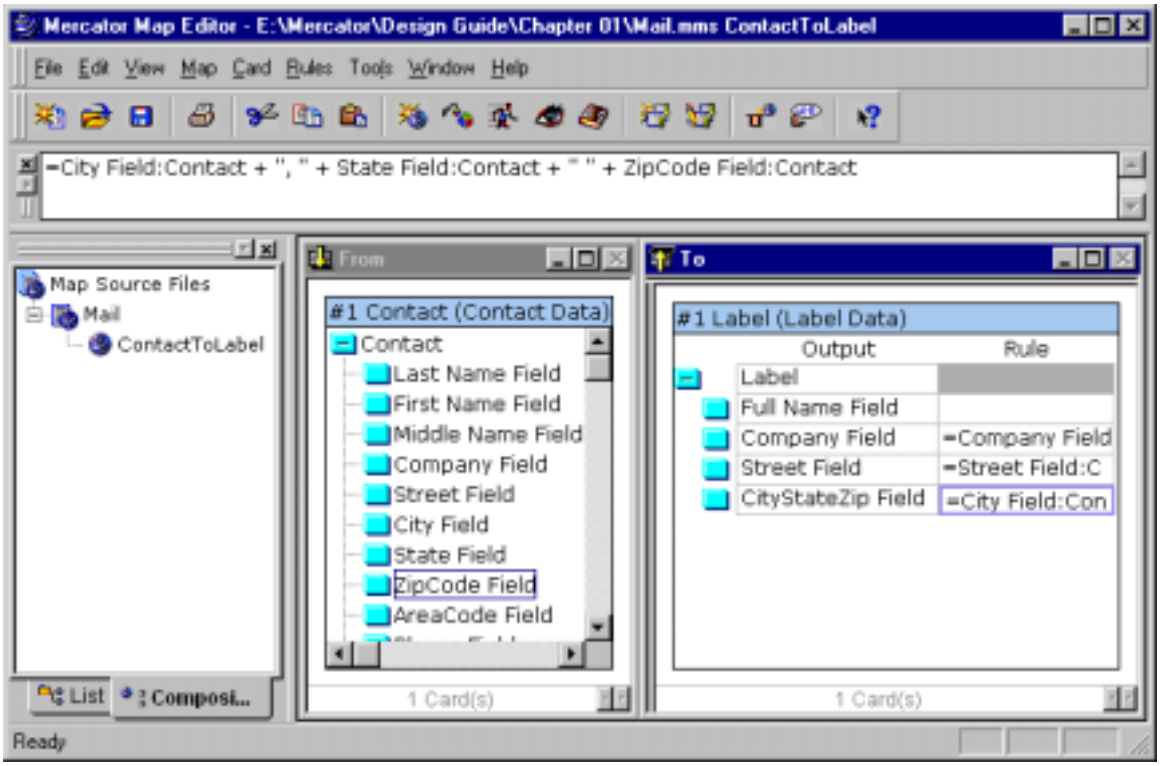
7 Drag and drop *ZipCode Field* into the rule bar.

8 Press the ENTER key.

Note: To enter the rule into the rule cell, you **must** press **ENTER**.

Your rule should now look like this:

```
=City Field:Contact + ", " + State Field:Contact + " " + ZipCode
Field:Contact
```



Mapping to Full Name Field

You want to map the *First*, *Middle*, and *Last* Name Fields in the Contact, to the *Full* Name Field in the Label.

Remember that the *Middle* Name Field is optional—it does not have to appear in the data. Some people do not have a middle name. If the contact person does not have a middle name, you do not want to map the *Middle* Name Field.

Use a rule that checks to see if the Middle Name Field is present. If it is, you map it to the Full Name Field. If it is not, you do not map it.

Functions Used in Map Rule

This rule uses two Mercator functions:

- **PRESENT**
- **IF**

The **PRESENT** function checks the presence of an object. If the object is present, the function returns the boolean *TRUE*. If the object is *not* present, the function returns the boolean *FALSE*. The syntax of the **PRESENT** function is:

PRESENT (*Object to check the presence of*)

The **IF** function has three arguments. The first argument is a condition. If it is TRUE, Mercator performs the second argument. If it is FALSE, Mercator performs the third argument. The syntax of the **IF** function is:

IF (*Condition, Do this if the condition is met, Do this if the condition is not met*)

A description of the map rule is: "If the Middle Name Field is present, map the First, Middle, and Last Name Fields. If the Middle Name Field is *not* present, map only the First and Last Name Fields."

The outer function in the map rule is the **IF** function. Its first argument is the **PRESENT** function. The **PRESENT** function performs the test of a condition. The second argument for the **IF** is the text concatenation of the first, middle and last name—what to map if the condition is true. The third argument for the **IF** is a text concatenation of the first and last name—excluding the middle name—what to map if the condition is false.

The rule you enter looks like this:

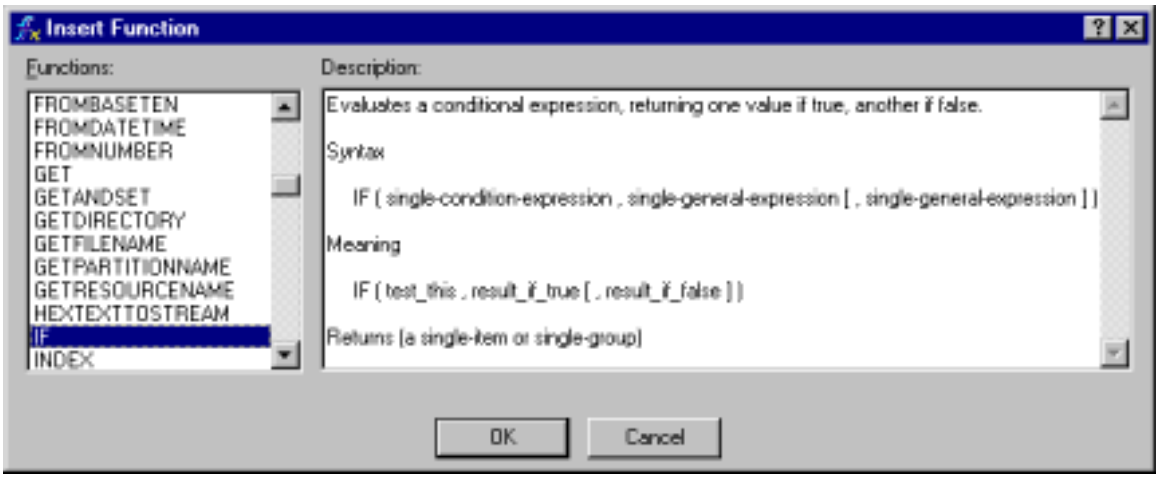
```
=IF(PRESENT(Middle Name Field:Contact),
First Name Field:Contact + " " + Middle Name Field:Contact + " " +
Last Name Field:Contact,
First Name Field:Contact + " " + Last Name Field:Contact)
```

To Enter a Map Rule for Full Name Field

1 Select the rule cell next to Full Name Field.

2 From the **Rules** menu, select **Insert Function**.

The Insert Function dialog is displayed, with a list of all Mercator functions.



3 Scroll down the list of functions until you find **IF**, and select it.

4 Click **OK**.

In the rule bar **=IF()** is displayed. The cursor is displayed between the parentheses, ready for you to enter the first argument of the function.

5 To make the rule easier to read, type a few spaces.

6 Select **Insert Function** from the **Rules** menu.

7 Select the **PRESENT** function.

8 Click **OK**.

9 Type a few spaces.

10 Drag and drop Middle Name Field into the rule bar, between the parentheses of the **PRESENT** function.

11 After the end parenthesis of the **PRESENT** function, type a *comma*, and a *space*.

12 Drag and drop *First Name Field* into the rule bar, after the space you just typed.

13 Type + " " +

14 Type a *space*.

15 Drag and drop *Middle Name Field* up into the rule bar.

16 Type + " " +

17 Type a *space*.

18 Drag and drop *Last Name Field* into the rule bar.

19 Type a *comma*, and a *space*.

20 Drag and drop *First Name Field* into the rule bar.

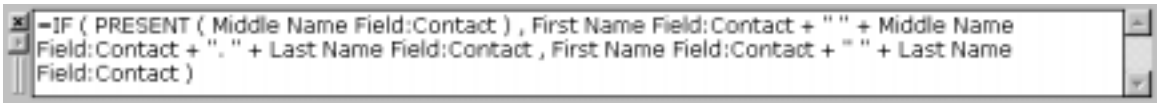
21 Type + " " +

22 Type a *space*.

23 Drag and drop *Last Name Field* into the rule bar.

24 Press the ENTER key.

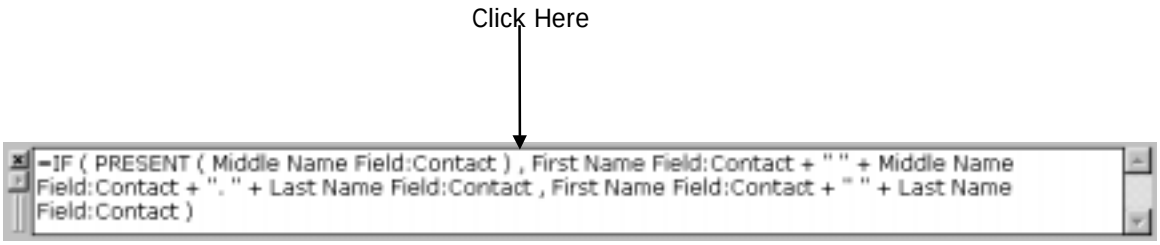
When you finish, the map rule should look like this:



You can format the map rule to be more easily read.

To Format a Map Rule for Full Name Field

1 In the rule bar, click to the right of the first comma.



2 Hold down the CTRL key, and press the ENTER key.

A new line is created in the rule bar.

3 Align the second line under the first line, by typing as many spaces as necessary. For example, you might want the second line to begin under the “P” of the **PRESENT** function.

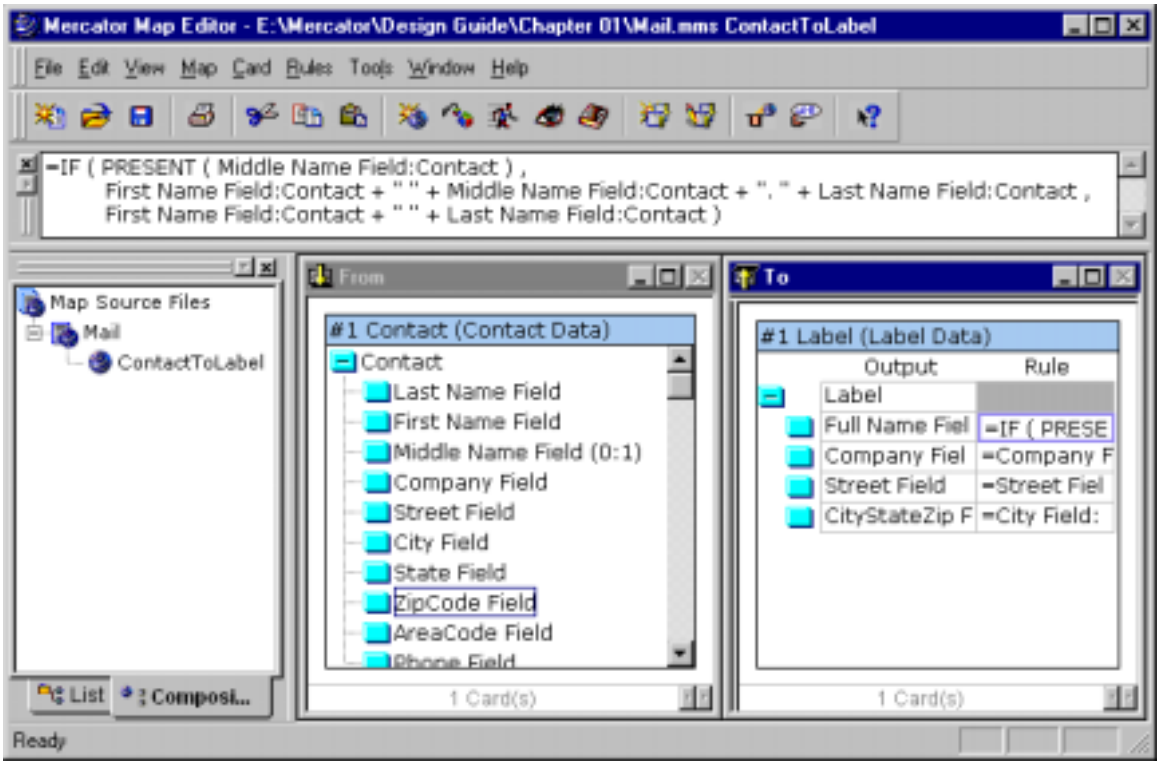
4 Click after the second comma.

5 Hold down the CTRL key, and press the ENTER key.

6 Align the third line under the second line.

7 Press the ENTER key.

When you finish, your rule should look like this:



Save the Source File

Now save the map file.

1 From the **File** menu, select **Save**.

Or, click the **File Save** tool. 

Build the Map

After entering map rules, you build the map.

To Build the Map

1 From the **Map** menu, select **Build**.

Or, click the **Build** tool. 

The Building Map dialog is displayed, and Mercator analyzes the map rules. If no errors occur, Mercator compiles the map.

If You Have Errors

1. In the Building Map dialog, click the **Results** button.
2. Fix the errors.
3. Build the map again.

For help in resolving errors, see Chapter 11 – Building a Map, in the *Map Editor Reference Guide*.

Run the Map

You can run the map from the Building Map dialog.

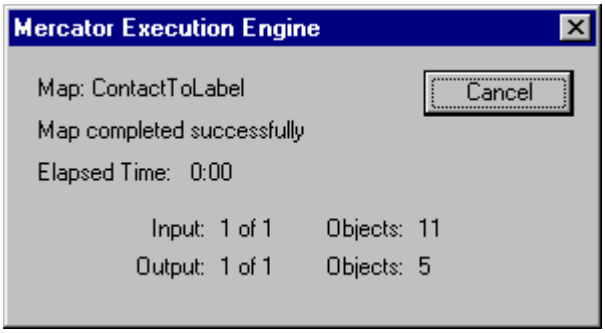
To Run the Map

1. In the Building Map dialog, click **Run**.

Or, if you do not have the Building Map dialog displayed, select Run from the Map menu.

Or, click the Run tool. 

The Command Execution Engine window is displayed.



You should see the message, *Map completed successfully*, in the engine window. If you do not have this message, see Chapter 15 – Debugging a Map, in the *Map Editor Reference Guide*. The time shown in the window may be different from that shown above.

2 From the **File** menu in the Engine window, select **Exit**.

View Results

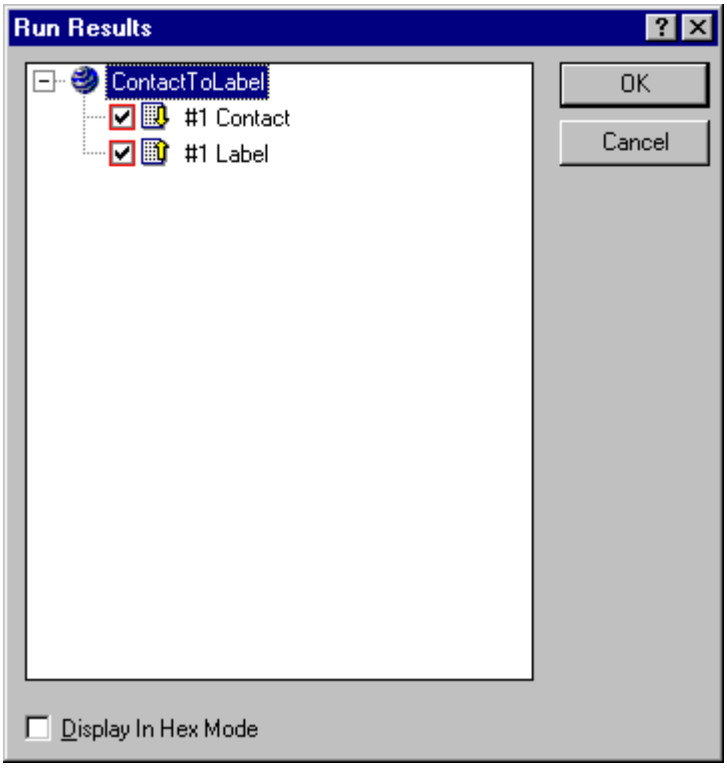
Now, look at the results of your map.

To View Results

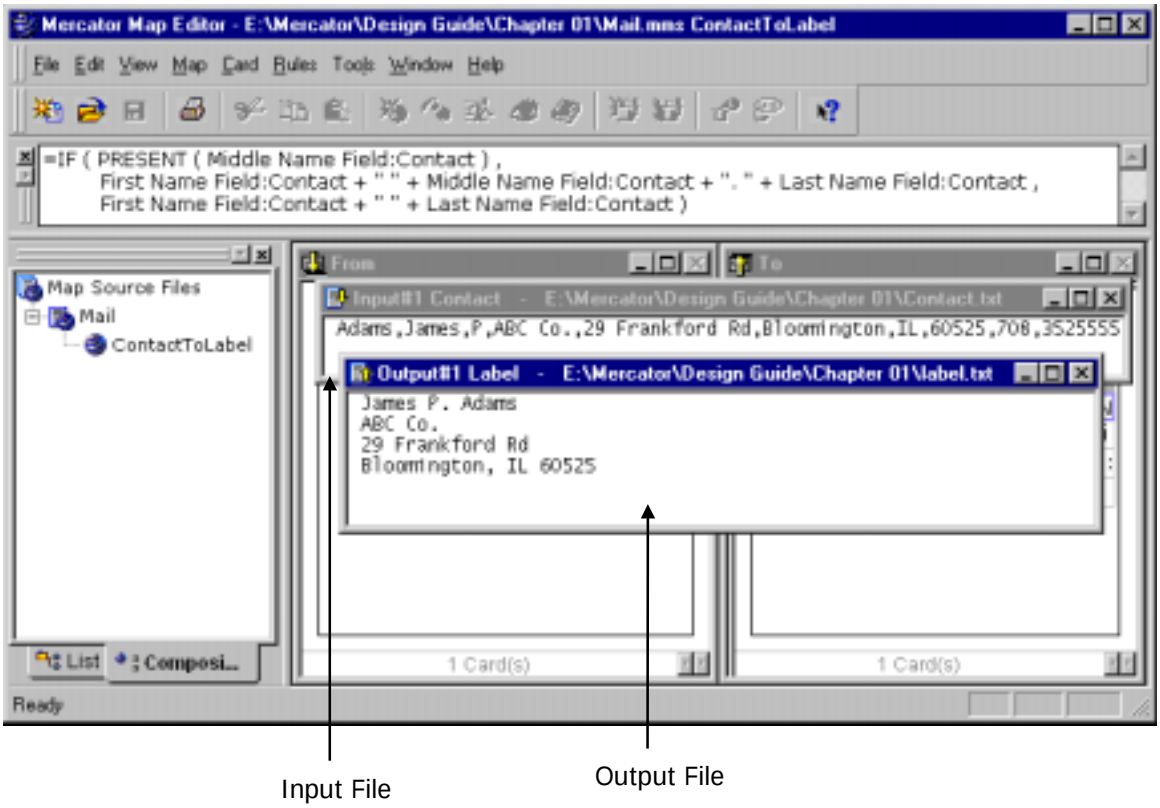
1 From the **View** menu, select **Run Results**.

Or, click the View Run Results tool. 

Each data file is displayed in its own window.



Select a data file and click **OK**. You will then see the results.



CONGRATULATIONS! You have completed your first map.

Chapter 2 – Mapping Records

This chapter explains how to map a file of multiple records to another file of multiple records. It discusses using a previously created executable map as a functional map. This example builds on the work you did in Chapter 1.

What You Want to Do

You have a file that contains many contact records. You want to generate a file of many labels—one label per contact record.

How to Do It

You already created a type tree that defines one contact record and one mailing label. You need to define the entire file made up of many contact records and the entire file made up of many labels.

Then, in the Map Editor, you create a map that maps the contact file to the label file.

Files Used in This Chapter

The following table lists the input file to use and the files to modify and create, as you work through the example in this chapter.

File	Use
address.txt	Use this data file as input. It is located in your mercator\examples folder (directory in Windows xx).
address.mtt	Modify this type tree file that you created by working through the tutorial in Chapter 1.
mail.mms	Modify this map source file that you created by working through the tutorial in Chapter 1.
mail.txt	Running the completed map creates this output file.

Data Descriptions

Following are descriptions of the input and output data for this example.

Input Data

The input data file, address.txt, is made up of an unknown number of contact records.

Here is a portion of the data:

```
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,60525,708,3525555
Miller,Maria,B,Conrad Corp,1234 Smith St,Buffalo Grove,CA,60089,708,3334567
Smith,Fred,A,Sand Inc.,Beach Street,Pismo Beach,FL,33321,407,8123456
Veldin,Beth,M,Any Co.,697 Berry Road,Highland Park,IL,60012,708,4445987
```

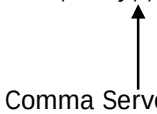
Each contact record matches the definition from Chapter 1.

Optional Data Objects

The Middle Name Field is optional. When it does not exist in the data, the comma delimiter still is displayed—as a placeholder for that field.

For example, Mary Martin does not have a middle name, so there is no data for the Middle Name Field. However, the comma is displayed, to indicate that field.

```
Martin,Mary,,Hooks and Hangers,123 Neverland Ave,Sky,TX,44444,302,6616000
```



Output Data

The output data is a file of mailing labels. You want to generate one label per contact record in the input.

The labels should look like this:

```
James P Adams
ABC Co.
29 Frankford Rd
Bloomington,IL 60525

Maria B Miller
Conrad Corp
1234 Smith St
Buffalo Grove,CA 60089

Fred A Smith
Sand Inc.
Beach Street
Pismo Beach,FL 33321
```

Each mailing label conforms to the definition from Chapter 1.

Using Type Editor

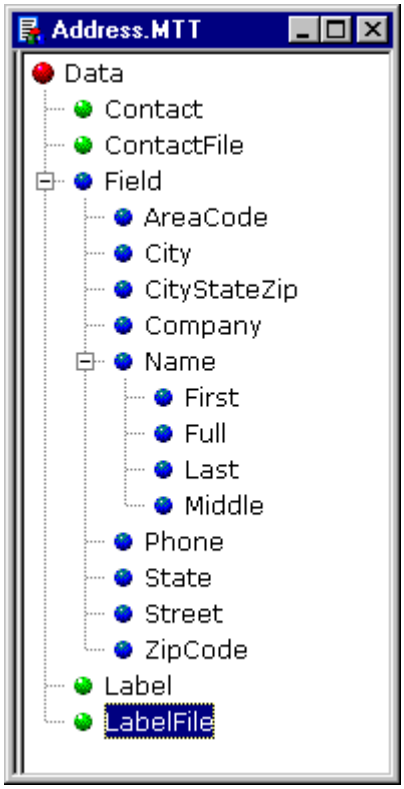
The first thing you need to do is define the input file and the output file in the address type tree.

To Create New Types

Create a type named *ContactFile* to represent the input file and a type named *LabelFile* to represent the output file.

1. In the Type Editor, open the **address.mtt** type tree.
2. Create a Group called *ContactFile* under the root type.
3. Create a Group called *LabelFile* under the root type.

When you finish, the type tree should look like this:



Identifying Properties of File Types

The type ContactFile is made up of Contacts. To determine the format of ContactFile, you can ask the following questions.

Is it fixed?	No—ContactFile contains an unknown number of Contacts. It does not have a fixed length.
Is it delimited?	No—there is no delimiter between the components, Contacts, of ContactFile. The CR/LF at the end of each Contact is already defined as the terminator of Contact. It cannot be considered a delimiter of ContactFile.
Is it implied?	Yes—if it is not fixed and it is not delimited, it is implied.

The format of LabelFile is also implied. It is made up of an unknown number of Labels.

Define Properties

You need to define the properties of ContactFile and LabelFile.

Define Properties of ContactFile

Define ContactFile as having an implied format.

The default Group format is implied, because it is inherited from the root. Unless you changed it, ContactFile should already be defined as implied.

Define Properties of LabelFile

Define LabelFile as having an implied format.

Identifying Components of File Types

The ContactFile is made up of an unknown number of Contacts. The range on the component Contact is **(s)**.

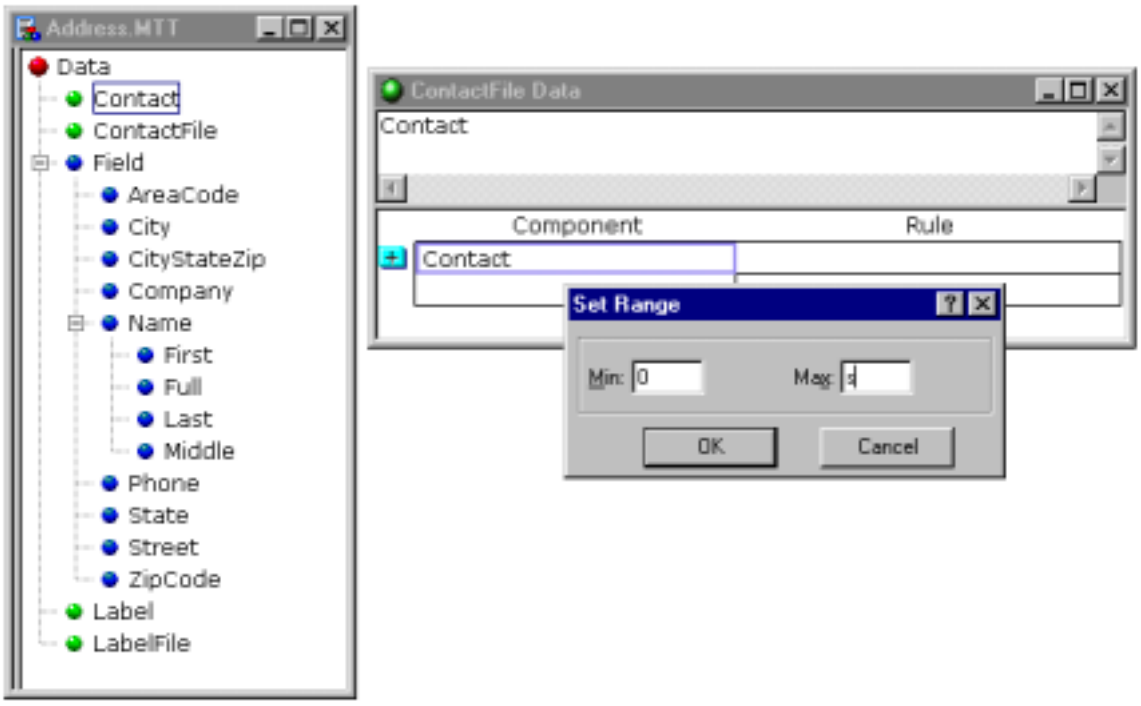
The LabelFile is made up of an unknown number of Labels. The range on the component Label is **(s)**.

Define Components

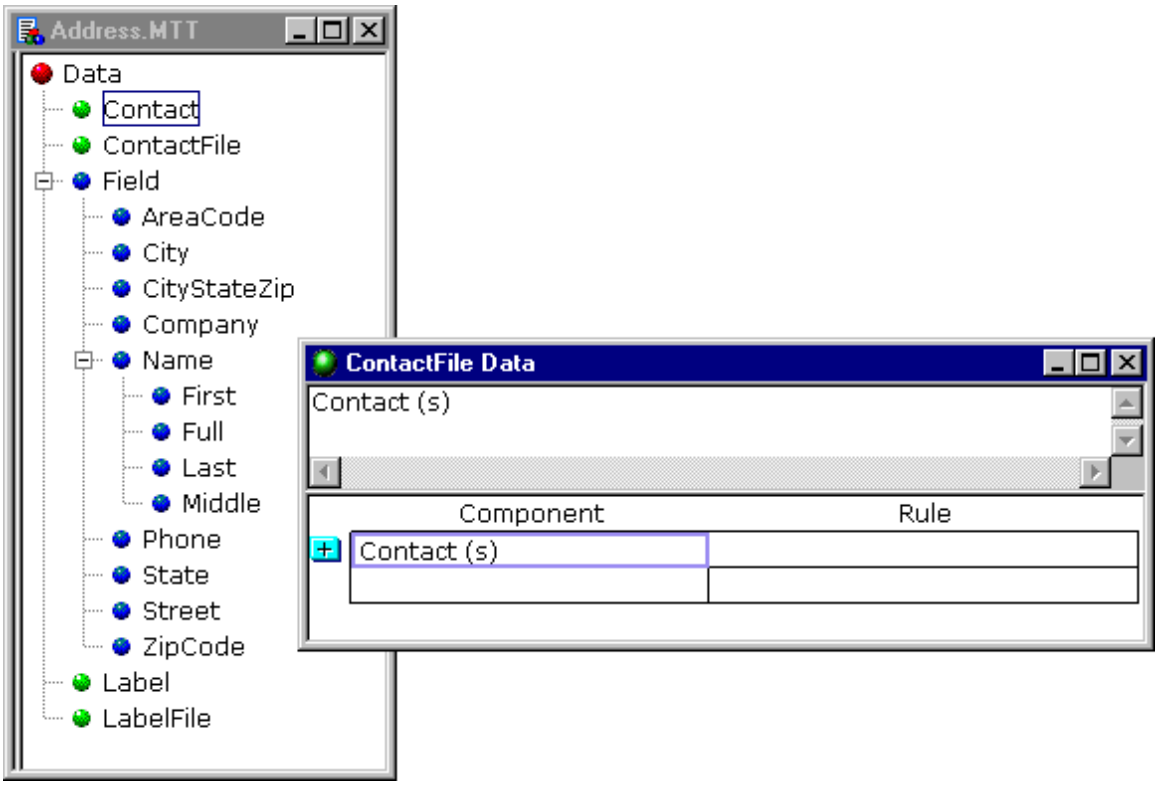
Now, you need to define the components of ContactFile and LabelFile.

To Define Components of ContactFile

1. Double-click *ContactFile*.
2. Drag and drop *Contact* into the component window of *ContactFile*.
3. In the Set Range window, type **s** in the Max field.


Address.MTT
Data
Contact
ContactFile
Field
AreaCode
City
CityStateZip
Company
Name
First
Full
Last
Middle
Phone
State
Street
ZipCode
Label
LabelFile
ContactFile Data
Contact
Component
Rule
Contact
Set Range
Min: 0
Max: 1
OK
Cancel

4 Click **OK**.


Address.MTT
Data
Contact
ContactFile
Field
AreaCode
City
CityStateZip
Company
Name
First
Full
Last
Middle
Phone
State
Street
ZipCode
Label
LabelFile
ContactFile Data
Contact (s)
Component
Rule
Contact (s)

5 Save *ContactFile*.

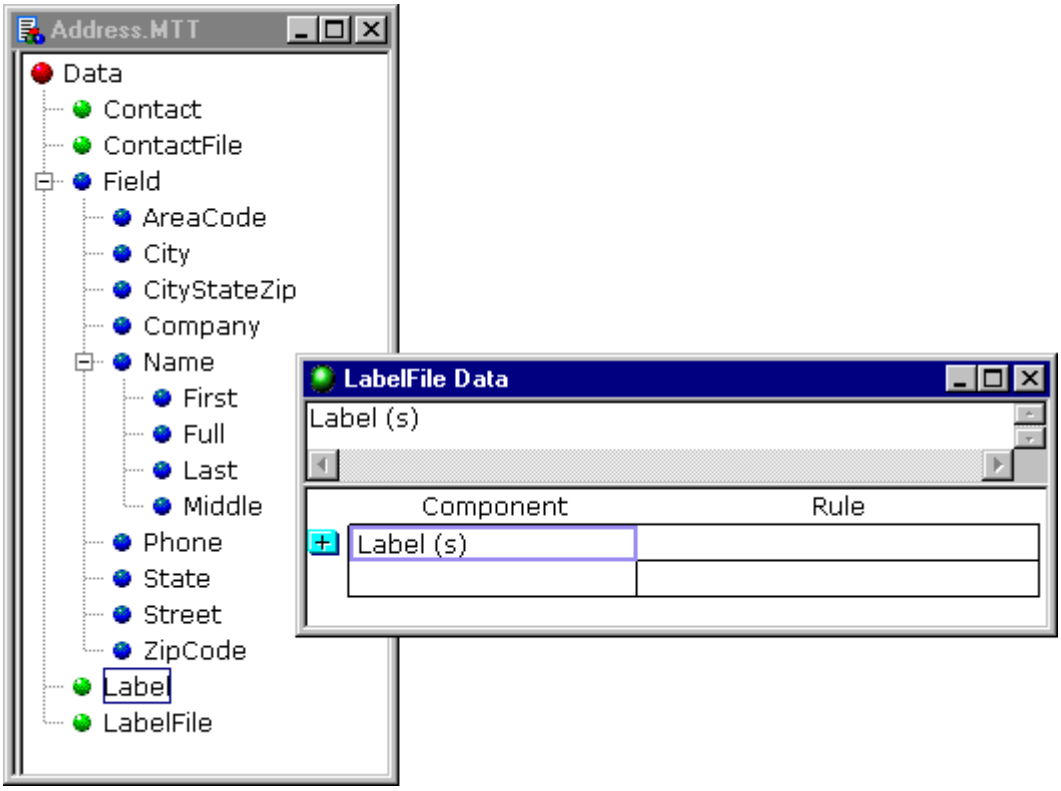
6 Close *ContactFile*.

To Define Components of LabelFile

1 Double-click *LabelFile*.

2 Drag and drop Label into the component window of *LabelFile*.

3 In the Set Range window, type **s** in the Max field.



4 Save *LabelFile*.

5 Close *LabelFile*.

To Analyze Type Tree

After you change a type tree, always analyze it, to make sure the changes made are consistent with the entire definition.

1 From the **Tree** menu, select **Analyze**.

2 Click **OK**.

If you get analysis errors, refer to the topic "Analyzer Error and Warning Messages," in Chapter 15 – The Type Tree Analyzer, of the *Type Editor Reference Guide*.

To Save Type Tree

After analyzing the tree, save it again.

1 From the **File** menu, select **Save**.

Or, click the File Save tool. 

Using Map Editor

To generate the *LabelFile*, create a map and store it in the source file created in Chapter 1—*mail.mms*. Name the new map *AddressToLabelFile*.

To Create a New Map

1 In the Map Editor, open the file **mail.mms**. It is located in the mercator\examples folder.

2 From the **Map** menu, select **New**.

The Create New Map dialog is displayed.

3 Enter *AddressToLabelFile* in the New Map Name box.

4 Click **OK**.

You see the new map, *AddressToLabelFile*.

Create Cards

In this map, the input is *ContactFile* and the output is *LabelFile*. Create an input card for *ContactFile* and an output card for *LabelFile*.

To Create an Input Card

1 Select the *From* window.

2 From the **Card** menu, select **Add Input**.

3 In the Card Name box, enter *ContactFile*.

4 Click the Files button in the Type section.

5 Select the Mercator Type Tree **address.mtt** and click **OK**.

6 Click the Browse button.

7 Select the type *ContactFile* and click **OK**.

8 In the Data section, click the Files button.

9 Select the file **address.txt** and click **OK**.

10 Click **OK**.

To Create an Output Card

1 Select the To window.

2 From the **Card** menu, select **Add Output**.

3 In the Card Name box, enter *LabelFile*.

4 In the Type section, click the Files button.

5 Select the Mercator Type Tree **address.mtt** and click **OK**.

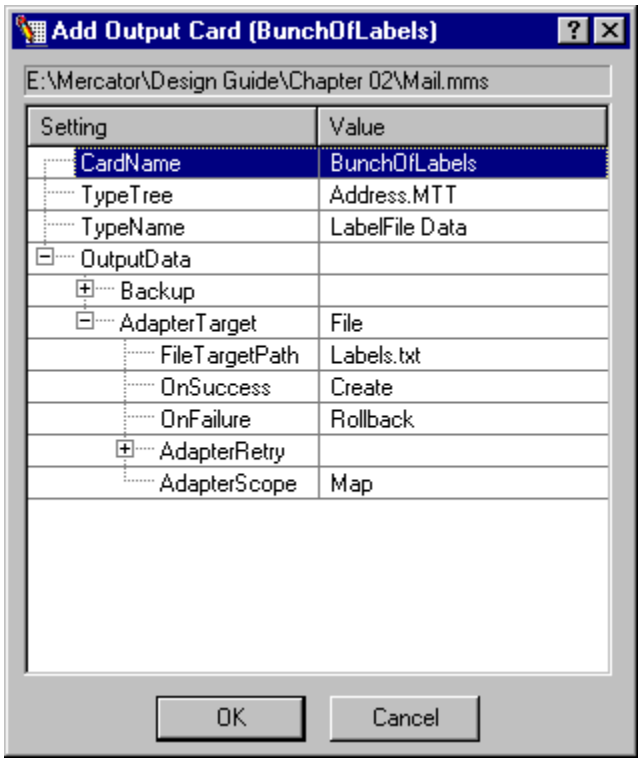
6 Click the Browse button.

7 Select the type *LabelFile* and click **OK**.

8 In the Name box of the File section, enter **mail.txt**.

9 Click **OK**.

The expanded output card should look like this:

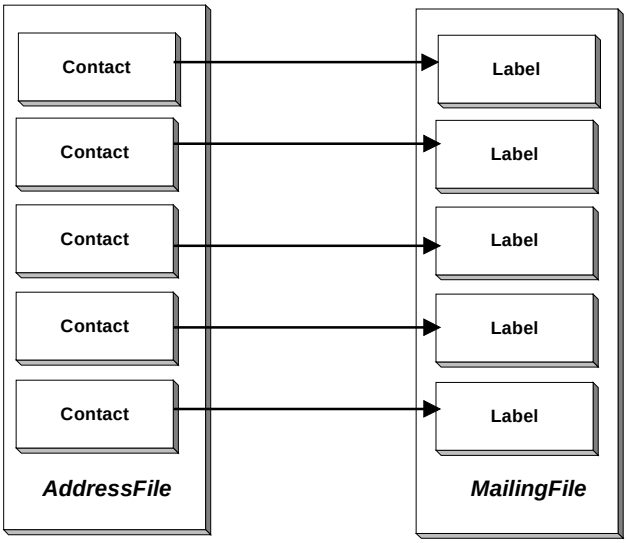


Enter Map Rules

Expand the output so you can see all of the nested components. Notice that the only empty rule cell is on the output Label (s). There is an unknown number of Labels. Whenever an output Group has a range that indicates more than one occurrence, you must decide how many of that output you want to generate.

How many Labels do you want to generate?

You want to generate one Label per Contact in the input—you want as many Labels as there are Contacts.

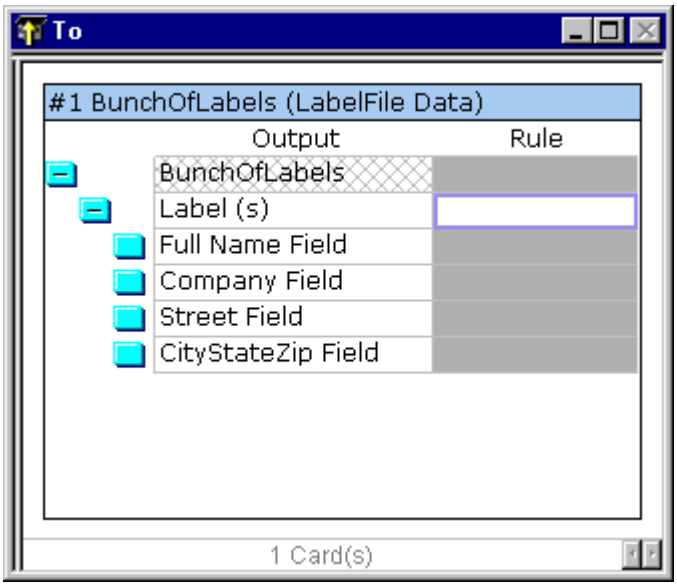


To generate one occurrence of an output Group for each occurrence of an input, use a functional map.

You need a functional map that makes one Label from one Contact. You already have a map like that! You created it in Chapter 1—the map *ContactToLabel*. You used it before as an executable map—the top level map, the one in charge of all the data. Now, use it as a functional map.

To Enter Map Rule for Output Label (s)

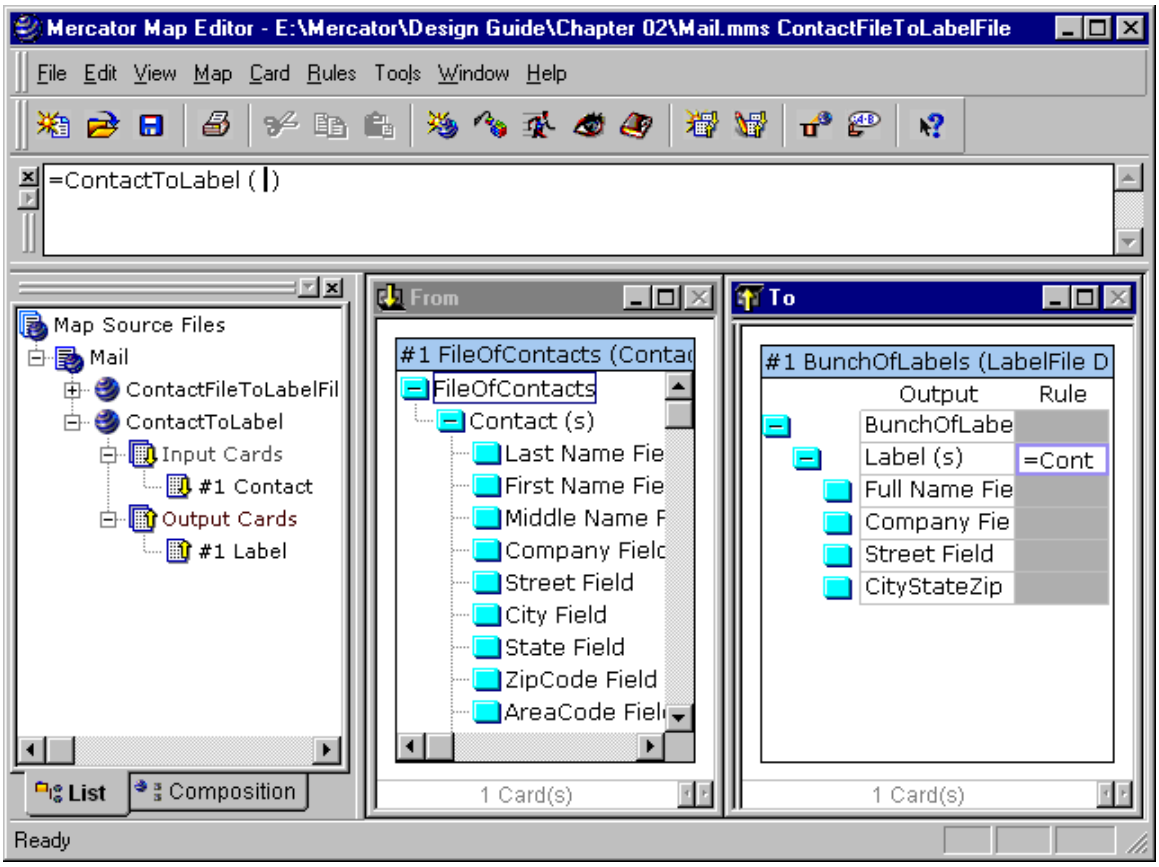
1. Select the rule cell for Label (s).
2. From the **Rules** menu, select **Insert Function**.
3. The Insert Map/Function dialog is displayed.



4 Select the map ContactToLabel.

5 Click **OK**.

6 In the rule bar you see =ContactToLabel ()

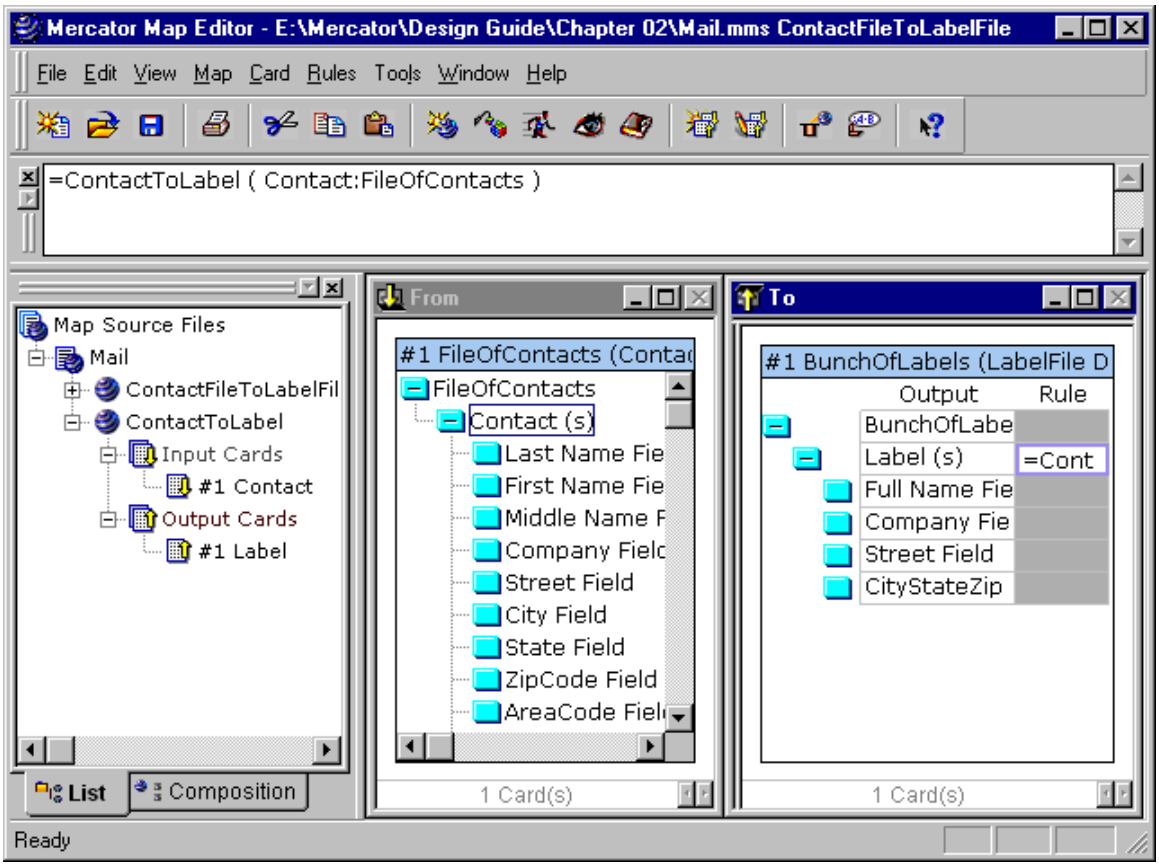


7 Drag and drop the input Contact up into the rule bar, between the parentheses.

8 The object name Contact:ContactFile is displayed.

9 Press the ENTER key.

The map should look like this:



To Build the Map

1 From the **Map** menu, choose **Build**.

Or, select the Build tool. 

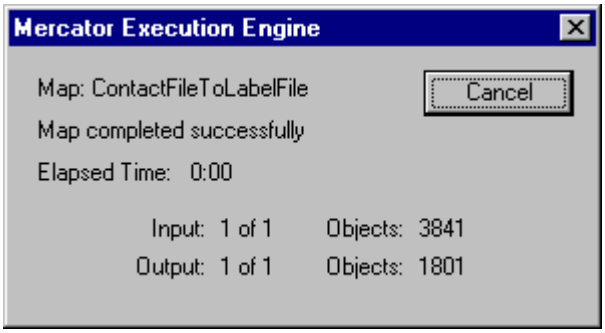
If you have errors, see Chapter 11 – Building a Map, in the *Map Editor Reference Guide*.

To Run the Map

1 In the Build Map dialog, click the **Run** button.

Or, select the Run tool. 

The Execution Engine dialog is displayed. After Mercator is finished running the map, you see the message, Map completed successfully. If you did not receive this message, see Chapter 15 – Debugging a Map, in the *Map Editor Reference Guide*.



1 Click **Cancel** to exit this dialog.

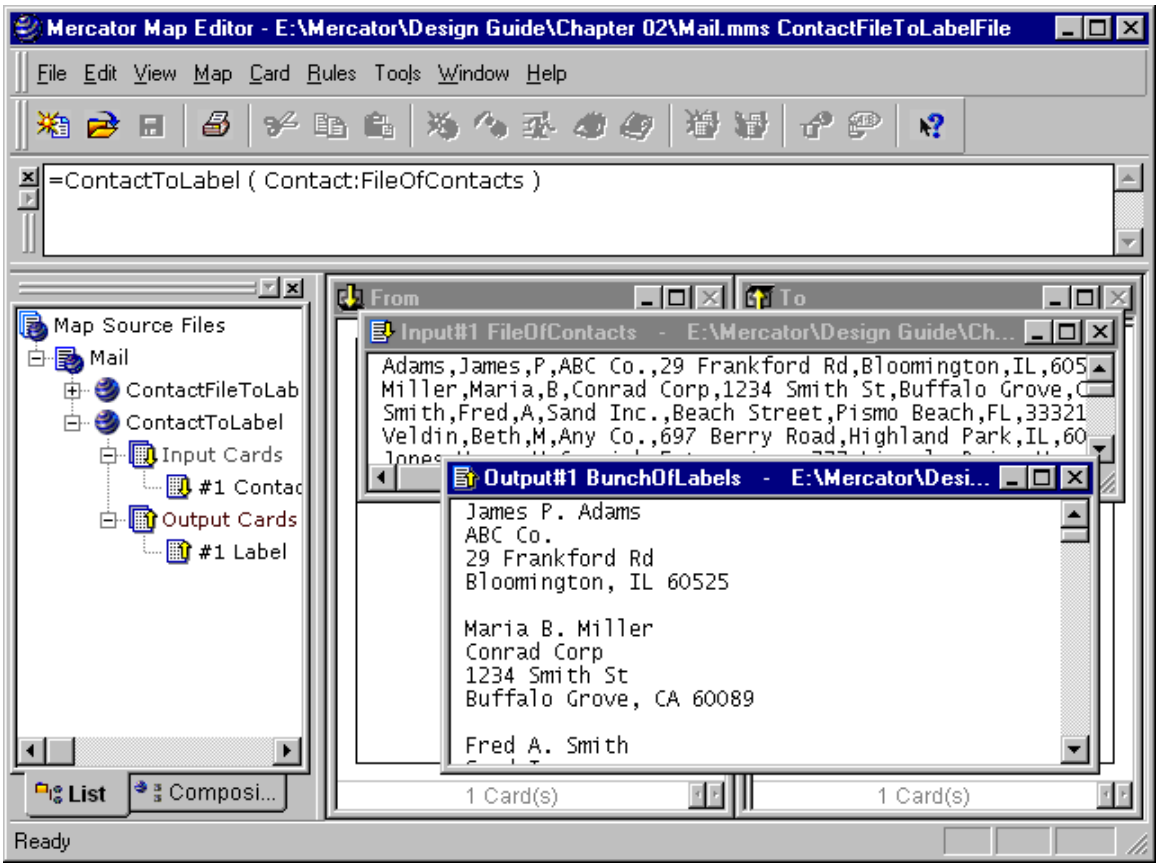
Note: Your map's execution time may be different from what is shown here. Execution times may vary from machine to machine.

To View the Results

1 From the **Map** menu, choose **Run Results**.

Or, select the View run results tool. 

You should see the input file, *address.txt*, and the output file, *mail.txt*, each in a separate window.



Now, save your work.

To Save the Source File

1 From the **File** menu, select **Save**.

Or, click the File Save tool. 

Chapter 3 - Using the UNIQUE Function

This example uses the UNIQUE function, to map data based on unique data values.

What You Want to Do

You may have noticed that some of the contact records in the address file occur multiple times. Suppose that you do not want to create duplicate contact records. You want to generate a file containing only the unique contact records.

How to Do It

You can modify the map you already created—AddressToLabelFile—in the source file mail.mms. To map only the unique contacts, you use the UNIQUE function.

Files Used in this Example

This table lists the input file to use, and the files to modify and create, as you work through the example in this chapter.

File	Use
address.txt	Use as an input data file. It is located in your mercator\examples folder (directory in Windows 3.1).
address.mtt	You use this type tree file that you created in Chapter 1 and modified in Chapter 2.
mail.mms	You modify this map source file that you created in Chapter 1 and modified in Chapter 2.
unique.txt	This output file is created by running the completed map.

Using the Map Editor

In the Map Editor, you copy the input card from AddressToLabelFile, to a new map. This automatically copies the input card *and* generates the new map at the same time. Name the new map UniqueContacts.

To Copy the Input Card to the Input of a New Map

1. In the Map Editor, choose **Open** from the **File** menu. Open the file mail.mms. It is located in the mercator\examples folder.

You should see the map AddressToLabelFile. If you do not, choose it from the Map Source Files list.

2. Select the input card.
3. From the **Card** menu, select **Copy**.

The Copy Input Card dialog is displayed.

4. In the Map section, enter *UniqueContacts*.
5. Click **OK**.

Because the output of this map is going to also be an Address File, you can copy the same card to the output of UniqueContacts.

To Copy the Input Card to the Output of a New Map

1 Select the input card in the map AddressToLabelFile.

2 From the **Card** menu, select **Copy**.

The Copy Input Card dialog is displayed.

3 In the Map section enter UniqueContacts.

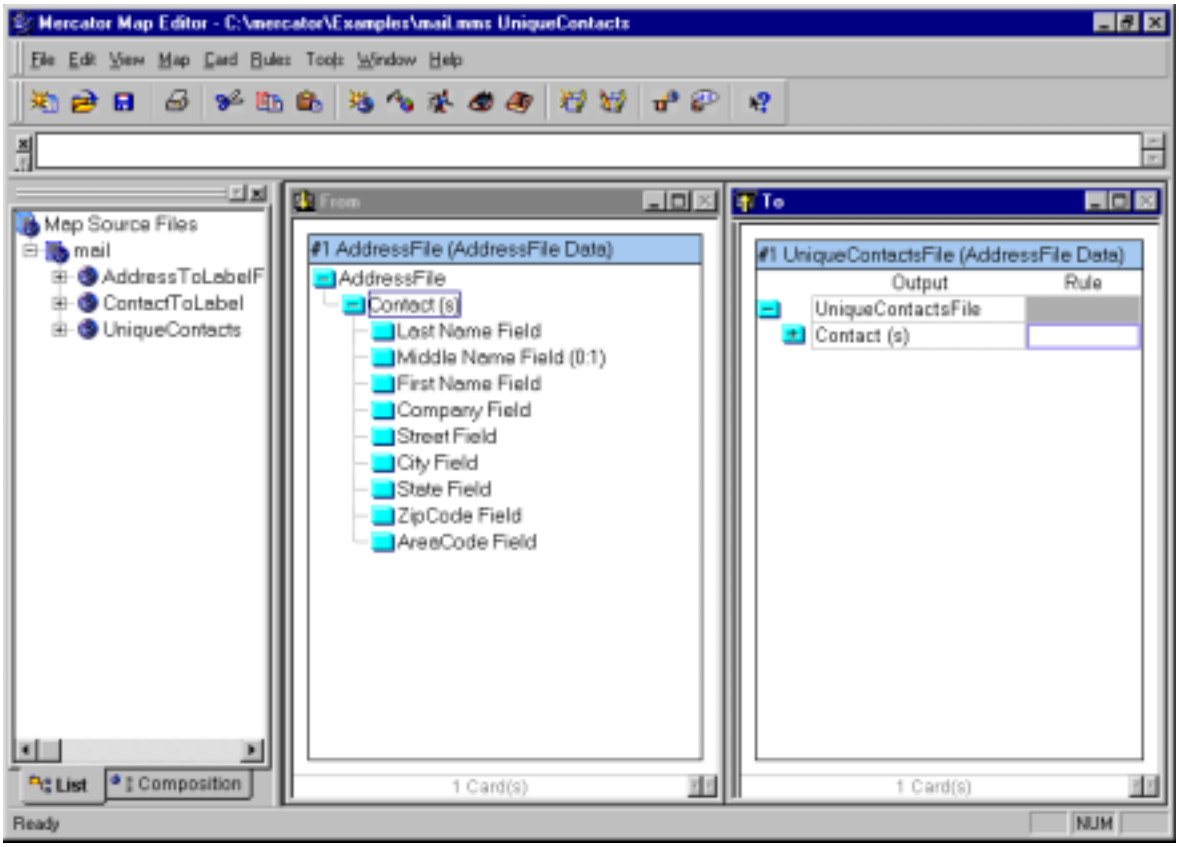
4 In the Card Name section, enter *UniqueContactsFile*

5 In the Card section, select the Copy Card As Output button.

6 Click **OK**.

Now, go to the map UniqueContacts, by selecting it.

The map UniqueContacts is displayed.



To Edit the Output Card

You do not want to overwrite the input data file, so you need to change the name of the output data file.

1 Select the output card.

2 From the **Card** menu, select **Edit**.

3 In the Name box of the File section, enter *unique.txt*.

4 Click **OK**.

In this map, you want to map only the unique Contacts. You use the UNIQUE function.

The UNIQUE function evaluates a series of objects, belonging to some type. It returns the unique data objects in that series. The syntax of the UNIQUE function is:

UNIQUE (*Series whose unique objects you want*)

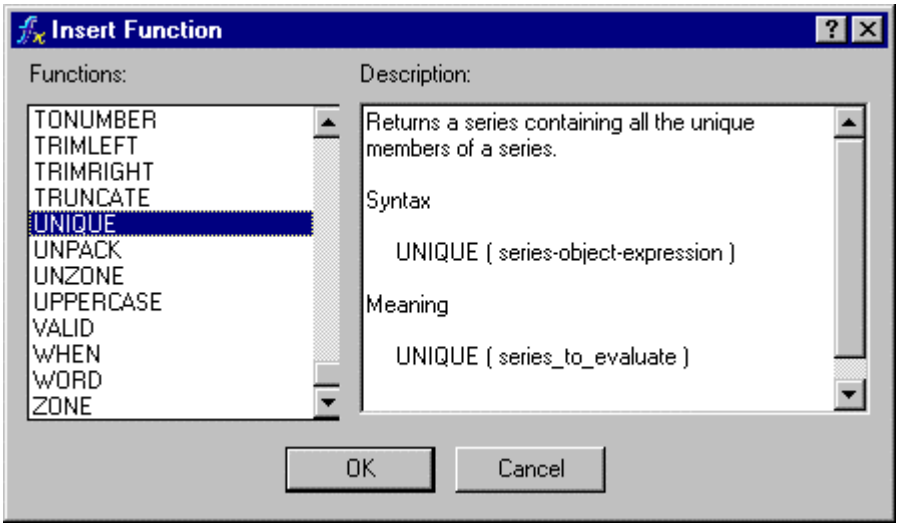
To Enter the Map Rule

1 Expand the output until you see the Contact (s).

2 Select the Rule cell for Contact(s).

3 From the **Rules** menu, select **Insert Function**.

The Insert Function dialog is displayed.



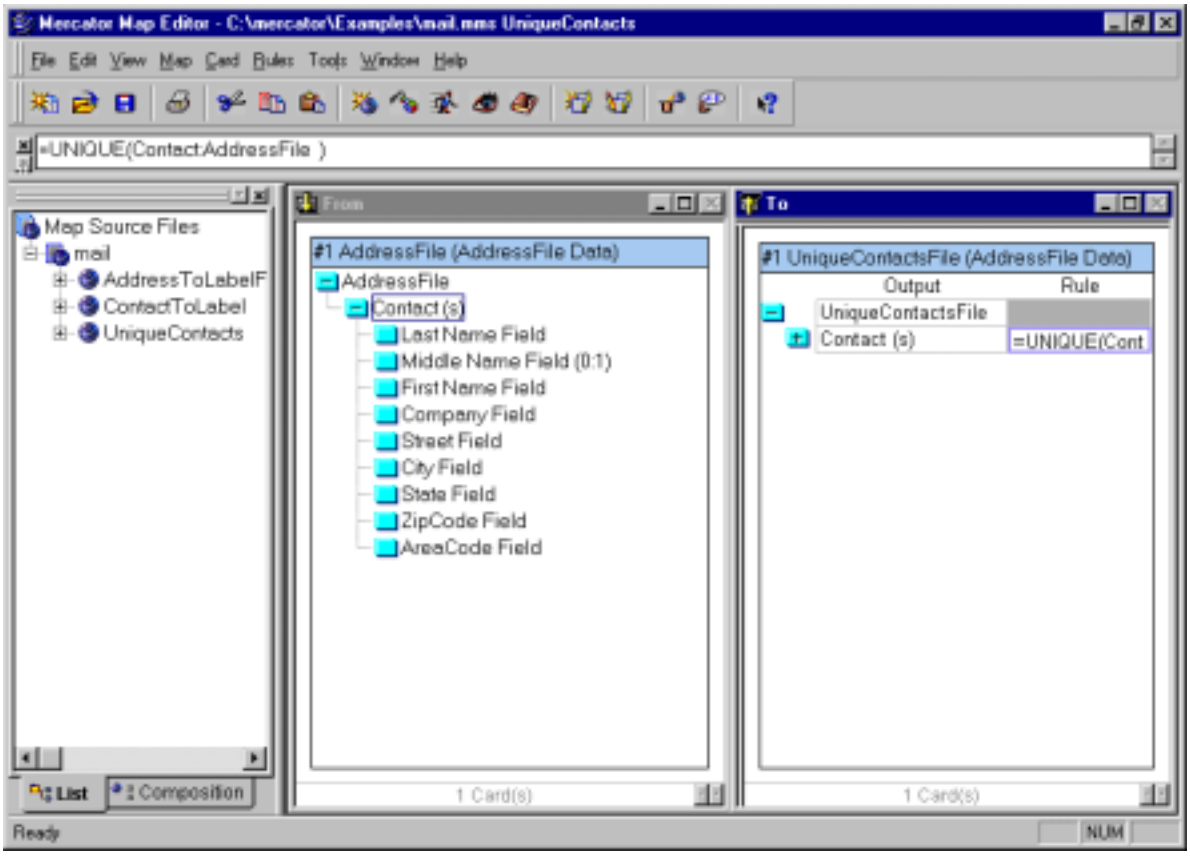
4 From the list of functions, select the UNIQUE function.

5 Click **OK**.

You see *UNIQUE ()* in the map rule.

7 Drag Contact from the input into the rule bar and drop between the parentheses.

8 Press the ENTER key.



To Build the Map

2 From the **Map** menu, choose **Build**.

Or, select the Build tool. 

If you have errors, see Chapter 11 – Building a Map, in the *Map Editor Reference Guide*.

To Run the Map

2 In the Build Map dialog, click the **Run** button.

Or, select the Run tool. 

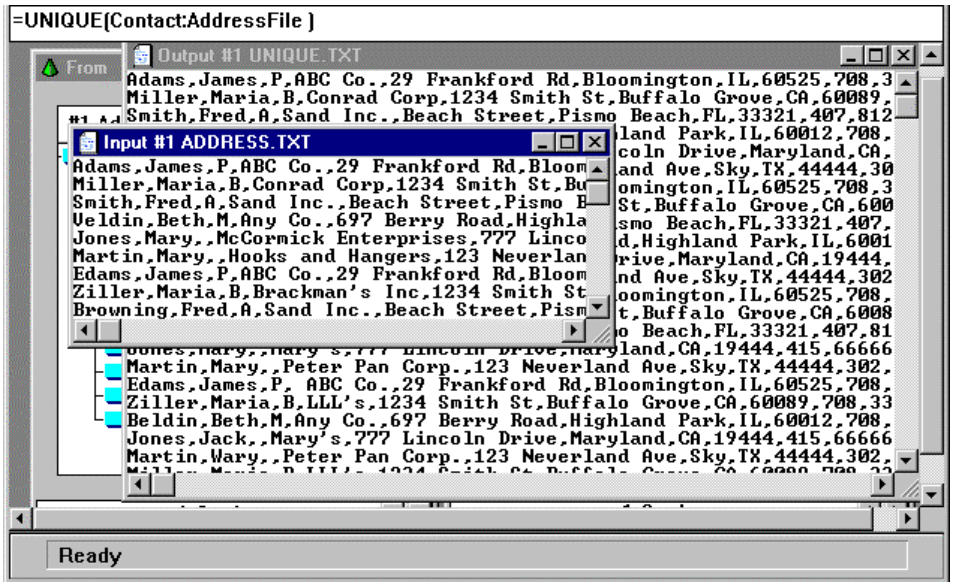
The Execution Engine dialog is displayed. After Mercator finishes running the map, you see the message, Map completed successfully. If you do not get this message, see Chapter 15 – Debugging a Map, in the *Map Editor Reference Guide*.

To View the Results

2 From the **Map** menu, choose **Run Results**.

Or, select the View run results tool. 

The input file, address.txt, and the output file, unique.txt, are displayed—each in a separate window. In the output, there are no duplicate contact record.



To Save the Source File

Finally, you need to save your work.

1 From the **File** menu, choose **Save**.

Or, select the **Save** tool. 

Chapter 4 - Using the EXTRACT Function

In this chapter, there are two examples that use the EXTRACT function to generate outputs only for specific inputs. Beginning with Case 2, it is assumed that you are familiar with type trees and maps as the result of working through previous chapters. Starting with Case 2 and in subsequent chapters, the problem and solution are described, but the solution is not shown step-by-step.

Case 1 – Extracting Contacts for a Specific State

Suppose you want to create labels only for your California customers.

How to Do It

You modify the map you already created—AddressToLabelFile—in the source file mail.mms. To map only the California contacts, use the EXTRACT function.

Files Used in Case 1

The following table lists the input file to use, and the files to modify and create, as you work through the first example in this chapter.

File	Use
address.txt	Use as an input data file. It is located in your mercator\examples folder.
address.mtt	You use this type tree file, which was created in Chapter 1 and modified in Chapter 2.
mail.mms	You modify this map source file, which was created in Chapter 1 and then modified in Chapter 2. (Working through the example in chapter 3 is not required.)
ca.txt	This output file is created by running the completed map.

Using the Map Editor

You copy the map AddressToLabel File, save it under a new name and modify it. The new map is named CaliforniaLabels.

To Copy the Map

1 In the Map Editor, open the file **mail.mms**. It is located in the mercator\examples folder.

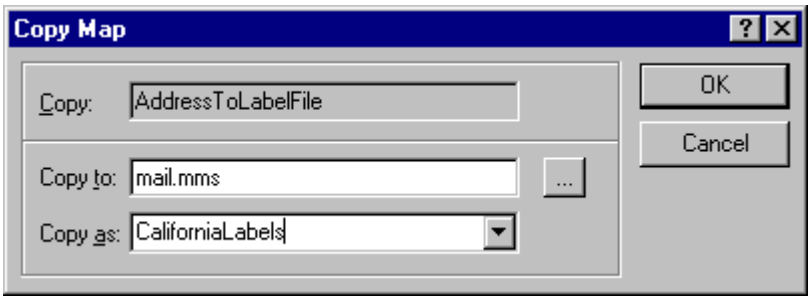
You should see the map AddressToLabelFile. If you do not, choose it from the Maps list.

2 From the **Map** menu, choose **Copy**.

The Copy Map dialog is displayed.

3 Click the Browse button.

4 Select the file **mail.mms**



5 Click **OK**.

To Edit the Output Card

To avoid overwriting the data file you created with the AddressToLabelFile map, you change the name of the output data file.

1 Select the output card.

2 From the **Card** menu, choose **Edit**.

3 In the File Target Path section, enter *ca.txt*.

4 Click **OK**.

Enter the Map Rule

Currently, the functional map ContactToLabel is used to map every contact to a label. In this example, you will use the EXTRACT function so that ContactToLabel only maps contacts that have the value "CA" in the State Field.

The EXTRACT function is used to select, from a series of objects of some type, all the ones that meet a particular condition. It has two arguments. The first is the series you want to evaluate. The second is the condition. EXTRACT returns the data objects of the first argument if any corresponding evaluation of the second argument is TRUE. The syntax of the EXTRACT function is:

EXTRACT (*Series whose objects you want to extract, Condition*)

To Enter the Map Rule

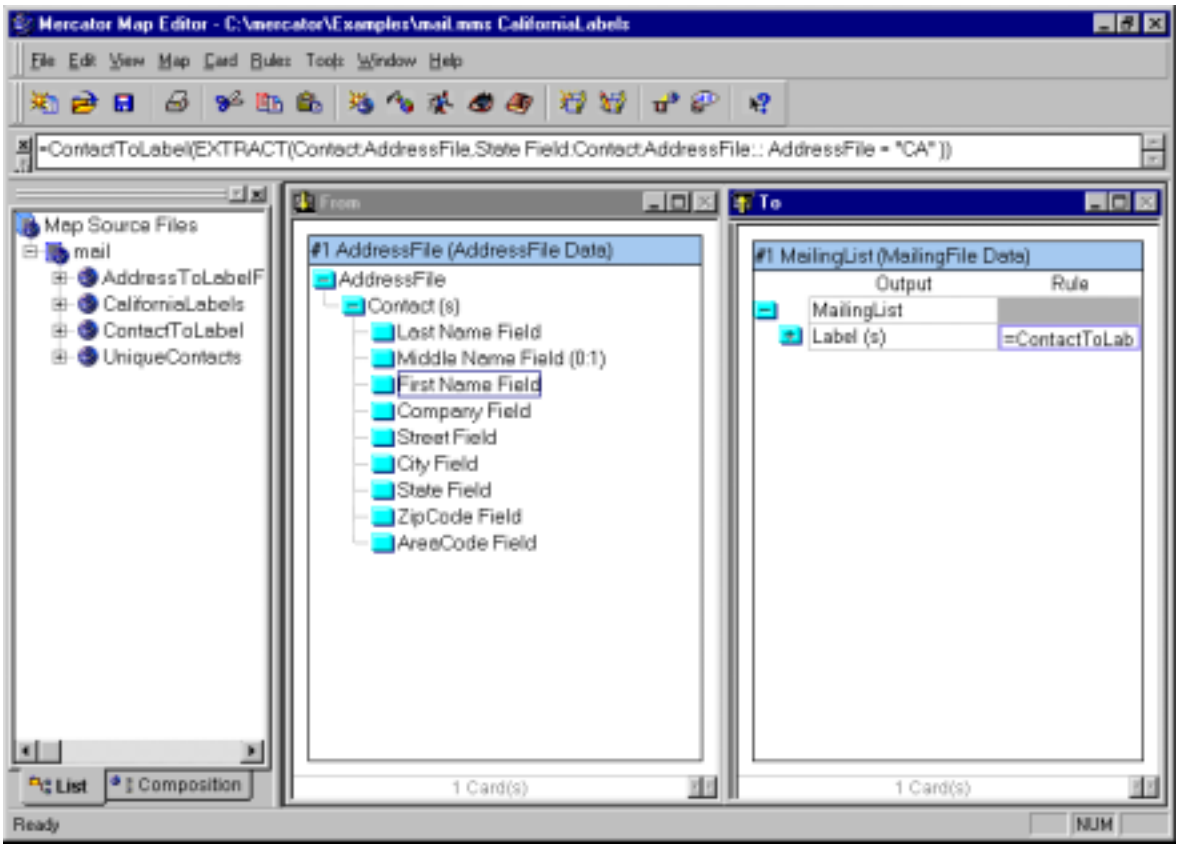
1. Click in the rule bar.
2. From the **Rules** menu, select **Insert Function**.

The Insert Function dialog is displayed.

3. Scroll down the list of functions, and select EXTRACT.
4. Click **OK**.

   You now see *EXTRACT ()* inserted into the map rule.

5. Delete the end parenthesis of the EXTRACT function.
6. Type a comma after "Contact:ContactFile."
7. Drag and drop State Field up into the rule bar, after the comma.
8. Type = *"CA"*
9. Type a right parenthesis at the very end of the rule.
10. Press the ENTER key.



To Build the Map

1 From the **Map** menu, choose **Build**.

Or, click the Build Tool. 

If you have errors, see Chapter 11 – Building a Map, in the *Map Editor Reference Guide*.

To Run the Map

1 From the **Map** menu, choose **Run**.

Or, click the Run Tool. 

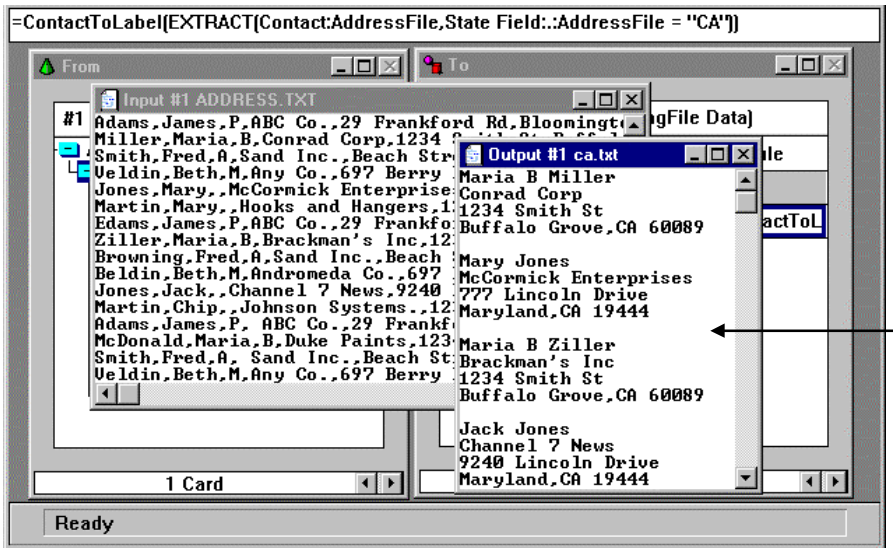
The Execution Engine dialog appears. After Mercator finishes running the map, you get the message, “Map completed successfully.” If you do not get this message, see Chapter 15 – Debugging a Map, in the *Map Editor Reference Guide*.

To View the Results

1 From the **Map** menu, choose **Run Results**.

Or, select the View run results tool. 

The input file, address.txt, and the output file, ca.txt, each are displayed in a separate window. Each label that was generated has the value “CA” for the State Field.



To Save the Source File

Finally, you want to save your work.

1 From the **File** menu, choose **Save**.

Or, select the **Save** tool. 

Case 2 – Extracting Contacts that are Preferred

Suppose you want to generate labels only for the preferred customers in the address file.

How to Do It

Generate a lookup file containing the names of preferred customers. You can use the lookup file to determine whether a contact is a preferred customer and map only these.

Define the lookup file in a type tree.

In the Map Editor, make a copy of the AddressToLabelFile map and use the EXTRACT function in the rule for Label.

Files Used in Case 2

The following table lists the input files you use, and the files you modify and create, as you work through Case 2.

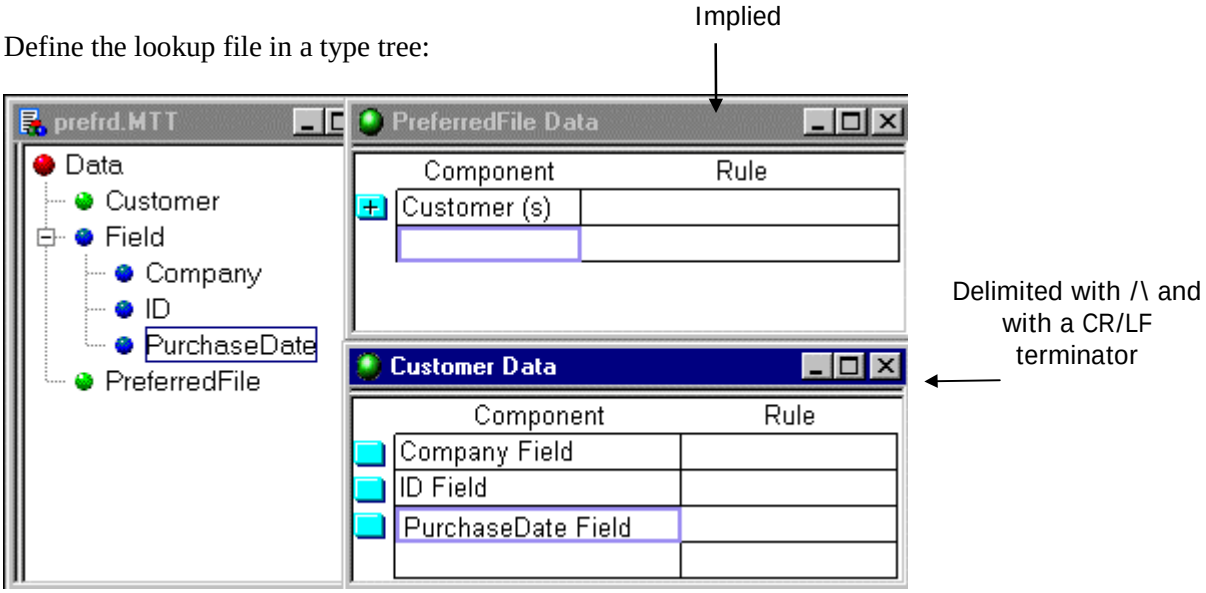
File	Use
address.txt	Use as an input data file. It is located in your mercator\examples directory (folder in Windows 3.1).
lookup.txt	You create this file to use as an input data file.
address.mtt	Use this type tree that defines the address.txt input data. It was created in Chapter 1 and modified in Chapter 2.
prefrd.mtt	You create this type tree to define the lookup file.
mail.mms	You modify this map source file, which was created in Chapter 1, and then modified in Chapter 2 and in Case 1 of Chapter 4. (Working through the example in chapter 3 is not required.)
pref.txt	This output file is created by running the completed map.

Using the Type Editor

You need to create the type tree for the following lookup file of preferred customers: If you want to work through this example, you also need to create the lookup file, containing these three records:

```
ABC Co.,35abc,970322
Sand Inc.,529heu,970912
Andromeda Co.,577ecc,960506
```

Define the lookup file in a type tree:



Create the Lookup Data

Create the following lookup data in a file:

```
ABC Co./\35abc/\970322
Sand Inc./\529heu/\970912
Andromeda Co./\577ecc/\960506
```

One way to create the data is simply type it in a word processor, and save the file as text. Another way to create the data is to create a map that has one output card—whose type is PreferredFile—and no input cards. Index three occurrences of Customer. In each map rule for each field, hard-code the field values. For example, the rule on the Company of the first Customer would be = “ABC Co.”. After you enter rules for the three Customers, build and run the map to generate the lookup file. In this example, the lookup file is called lookup.txt. For more information on using a map to generate data, see the *Map Editor Reference Guide*.

Using the Map Editor

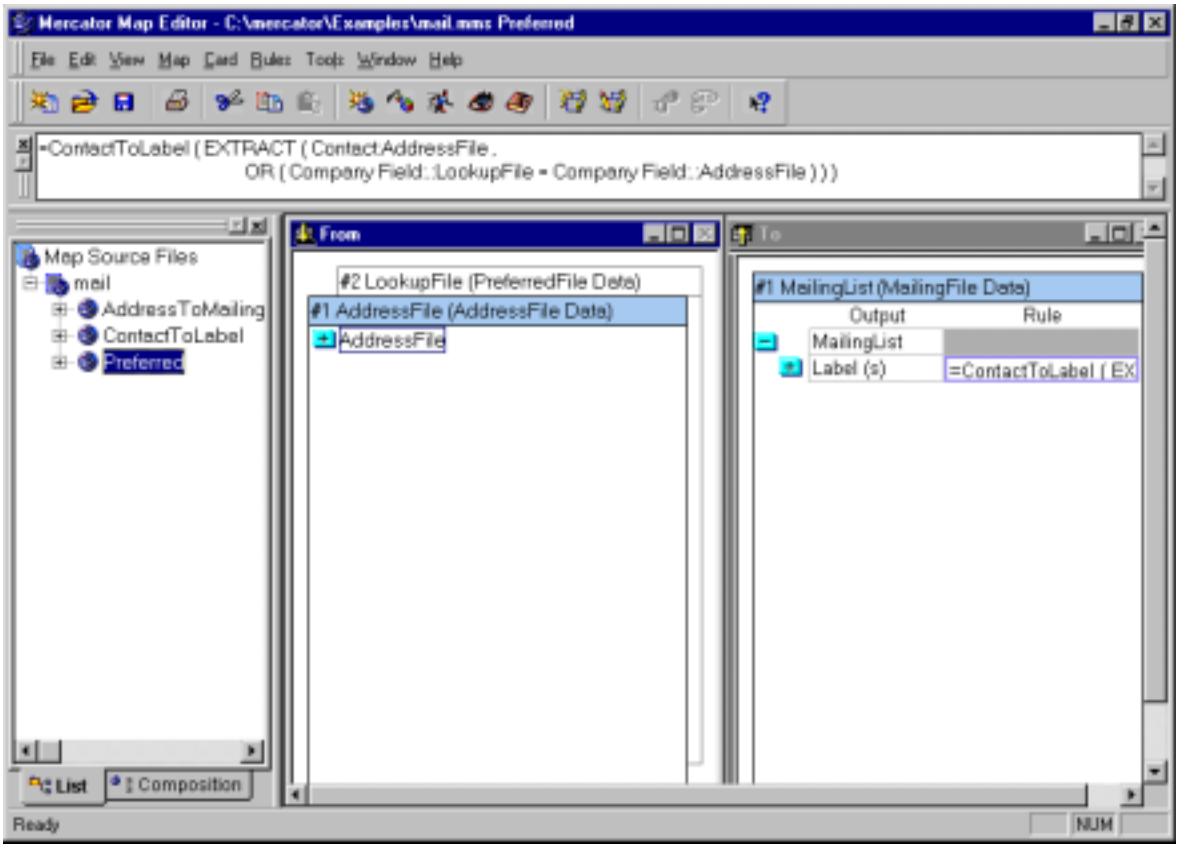
1. Open the file mail.mms.
2. Create a map called Preferred by copying AddressToLabelFile.
3. Edit the output card to change the file name to pref.txt.
4. Add a new input card for the lookup file.
5. Change the rule to use EXTRACT.

In the executable map named Preferred, there are two inputs—the address file and the lookup file. The rule on the output Label calls the functional map ContactToLabel. The input argument to this map uses the EXTRACT function to extract the Contacts, and the OR function to test if the Company Field appears in the LookupFile.

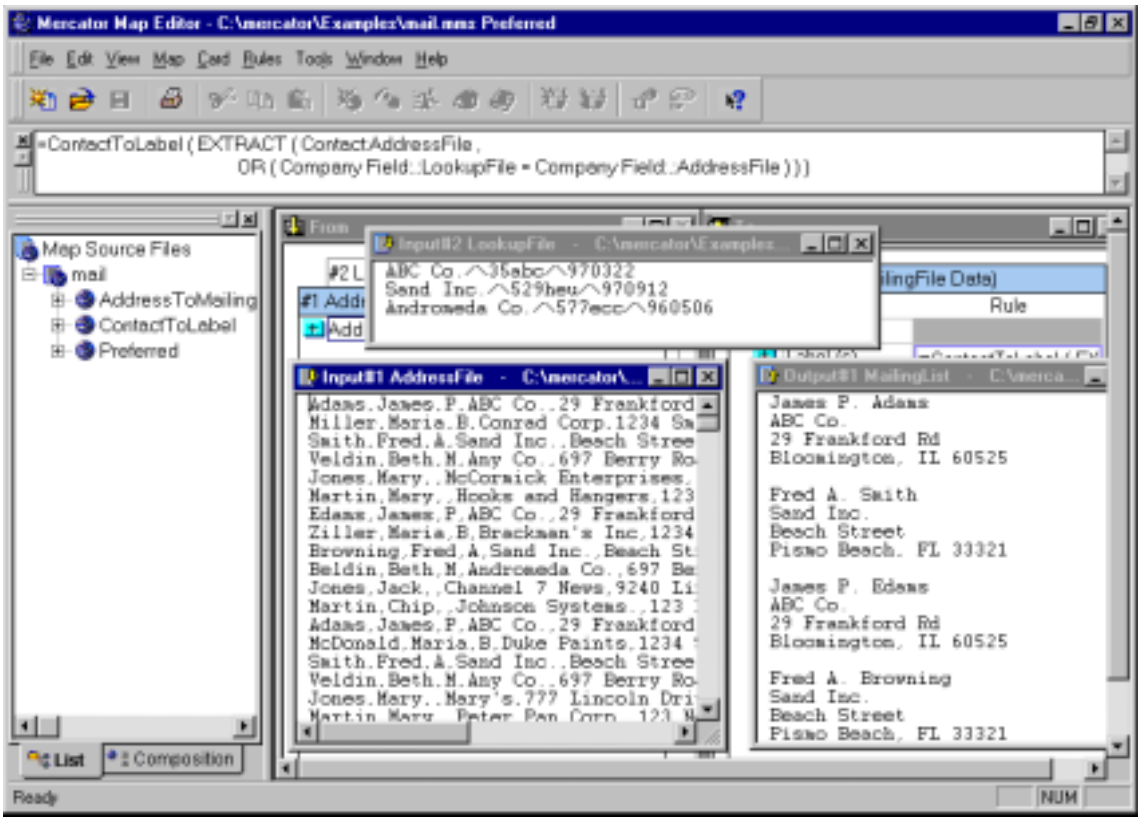
Here is the map rule on Label:

```
= ContactToLabel (EXTRACT (Contact:ContactFile,
OR (Company Field:.:ContactFile = Company Field:.:LookupFile))
```

Here is the map:



In the output, the only labels created are those for the companies that were present in the lookup file—ABC Co., Sand Inc., and Andromeda Co.



Chapter 5 - Testing the Existence of Data

This example uses the IF and PRESENT functions.

What You Want to Do

Generate an address file containing only those contacts with a middle name.

How to Do It

The input and output data have already been defined in Chapter 4. The definition of the input file is the same as that for the output file.

In the map, you use the IF and PRESENT functions, to check the presence of the Middle Name Field.

Files Used in this Example

The following table lists the files used in this example.

File	Use
address.txt	Use this file as the input data file. It is in your mercator\examples directory.
address.mtt	This is the type tree file last used in Chapter 4, Case 2.
mail.mms	You modify this map source file, which was last used in Chapter 4.
middle.txt	This output file is created by running the map.

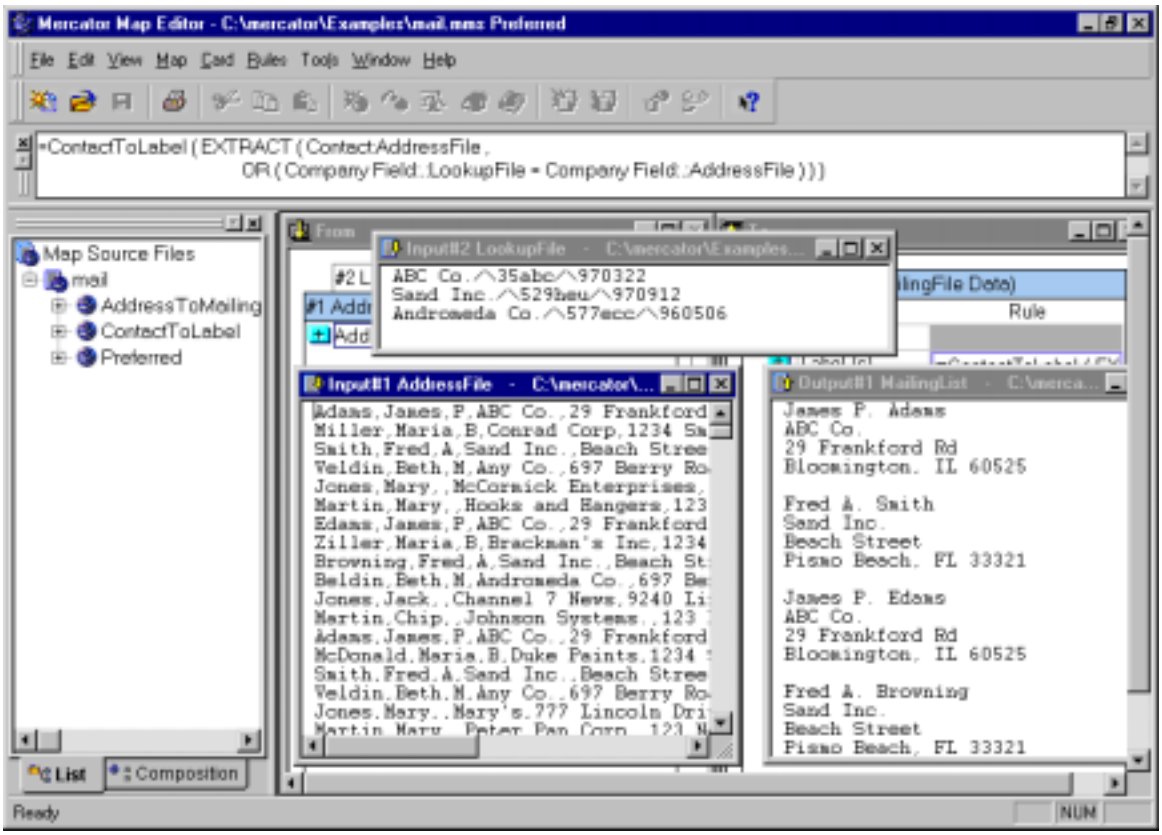
Using the Map Editor

You need only one map. The input is ContactFile, and the output is ContactFile.

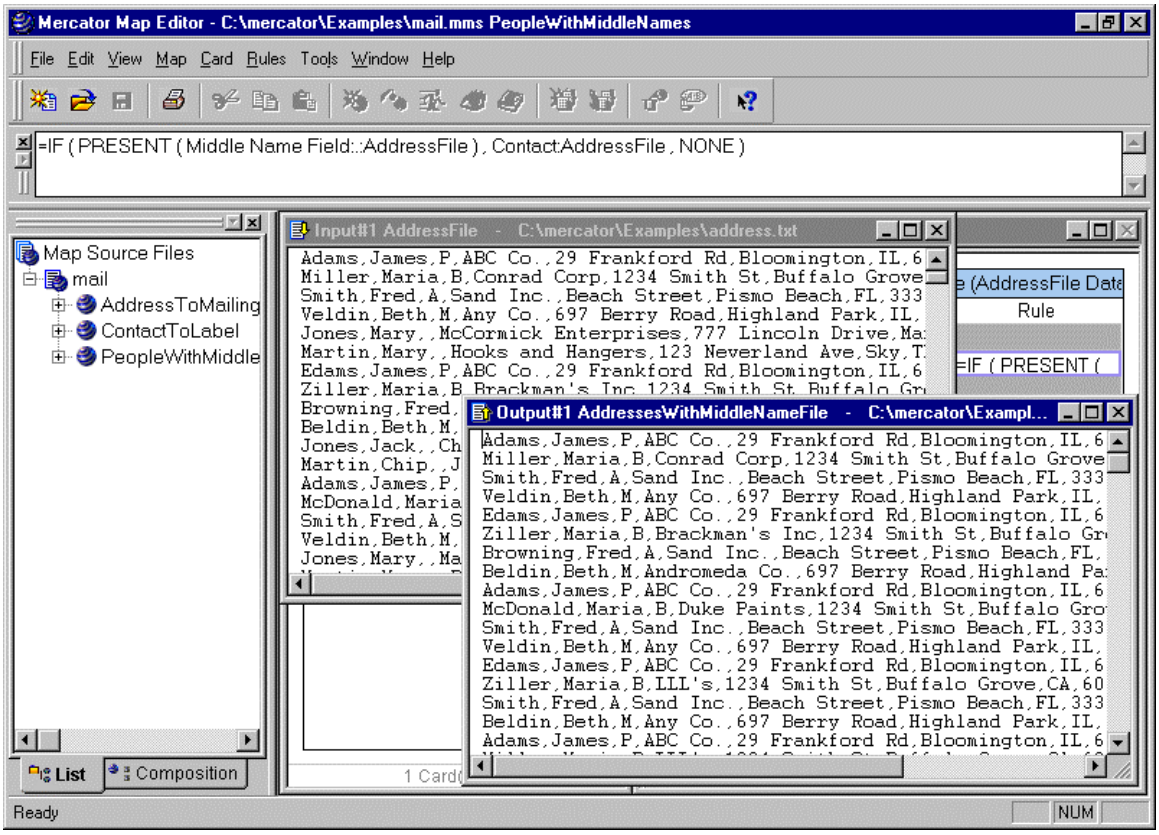
On the output Contact(s), you use the following rule:

```
=IF (PRESENT (Middle Name Field:.:Input), Contact:Input, NONE)
```

The map looks like this:



The resulting output file contains only those contacts with a middle name. For example, Mary Jones was not mapped to the output.


Mercator Map Editor - C:\mercator\Examples\mail.mms PeopleWithMiddleNames
File Edit View Map Card Rules Tools Window Help
=IF (PRESENT (Middle Name Field::AddressFile) , Contact:AddressFile , NONE)
Map Source Files
mail
AddressToMailing
ContactToLabel
PeopleWithMiddle
List
Composition
Input#1 AddressFile - C:\mercator\Examples\address.txt
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
Miller,Maria,B,Conrad Corp,1234 Smith St,Buffalo Grove
Smith,Fred,A,Sand Inc.,Beach Street,Pismo Beach,FL,333
Veldin,Beth,M,Any Co.,697 Berry Road,Highland Park,IL,
Jones,Mary,,McCormick Enterprises,777 Lincoln Drive,Ma
Martin,Mary,,Hooks and Hangers,123 Neverland Ave,Sky,T
Edams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
Ziller,Maria,B,Brackman's Inc,1234 Smith St,Buffalo Gr
Browning,Fred,
Beldin,Beth,M,
Jones,Jack,,Ch
Martin,Chip,,J
Adams,James,P,
McDonald,Maria
Smith,Fred,A,S
Veldin,Beth,M,
Jones,Mary,,Ma
(AddressFile Date
Rule
=IF (PRESENT (
Output#1 AddressesWithMiddleNameFile - C:\mercator\Exampl...
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
Miller,Maria,B,Conrad Corp,1234 Smith St,Buffalo Grove
Smith,Fred,A,Sand Inc.,Beach Street,Pismo Beach,FL,333
Veldin,Beth,M,Any Co.,697 Berry Road,Highland Park,IL,
Edams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
Ziller,Maria,B,Brackman's Inc,1234 Smith St,Buffalo Gr
Browning,Fred,A,Sand Inc.,Beach Street,Pismo Beach,FL,
Beldin,Beth,M,Andromeda Co.,697 Berry Road,Highland Pa
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
McDonald,Maria,B,Duke Paints,1234 Smith St,Buffalo Gro
Smith,Fred,A,Sand Inc.,Beach Street,Pismo Beach,FL,333
Veldin,Beth,M,Any Co.,697 Berry Road,Highland Park,IL,
Edams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
Ziller,Maria,B,LLL's,1234 Smith St,Buffalo Grove,CA,60
Smith,Fred,A,Sand Inc.,Beach Street,Pismo Beach,FL,333
Beldin,Beth,M,Any Co.,697 Berry Road,Highland Park,IL,
Adams,James,P,ABC Co.,29 Frankford Rd,Bloomington,IL,6
1 Card(
Ready
NUM

Chapter 6- Using Cross-Referenced Data

To cross-reference data from a file, and incorporate it into your output, you can use one of the following Mercator functions:

- LOOKUP - to get an object from a list that is not organized in any particular order
- SEARCHDOWN - to get an object from a list that is in descending order
- SEARCHUP - to get an object from a list that is in ascending order
- CHOOSE - to get an object at a certain position in a series

This chapter explains how to decide which function to use when you want to cross-reference data, and how to use the function. It provides a map example using each of these functions.

The examples also illustrate using the Functional Map Wizard to create a functional map.

When to Use LOOKUP, SEARCHDOWN, and SEARCHUP

Following are three different examples, each illustrating when to use one of these functions: LOOKUP, SEARCHDOWN, and SEARCHUP.

Case 1 - Using LOOKUP for Unordered Cross-Reference Data

The first example uses cross-reference data that is not in any particular sequence.

You have a file of unique contacts. You want to generate a file that contains records consisting of the name of each of your customer contacts, their company, and their geographical region.

How to Do It

The geographical region information is not in the input file, so you need another file, which contains this information. Because you do not have this cross-reference file, you create it. You define the lookup file and the output file in a type tree.

Then you create a map that has a single output card, and enter the data values you want. The cross-reference file contains zip codes and their corresponding geographical region.

Next, you create a map that uses the lookup file and the address file as inputs, and generates the output file you want.

Files Used in Case 1

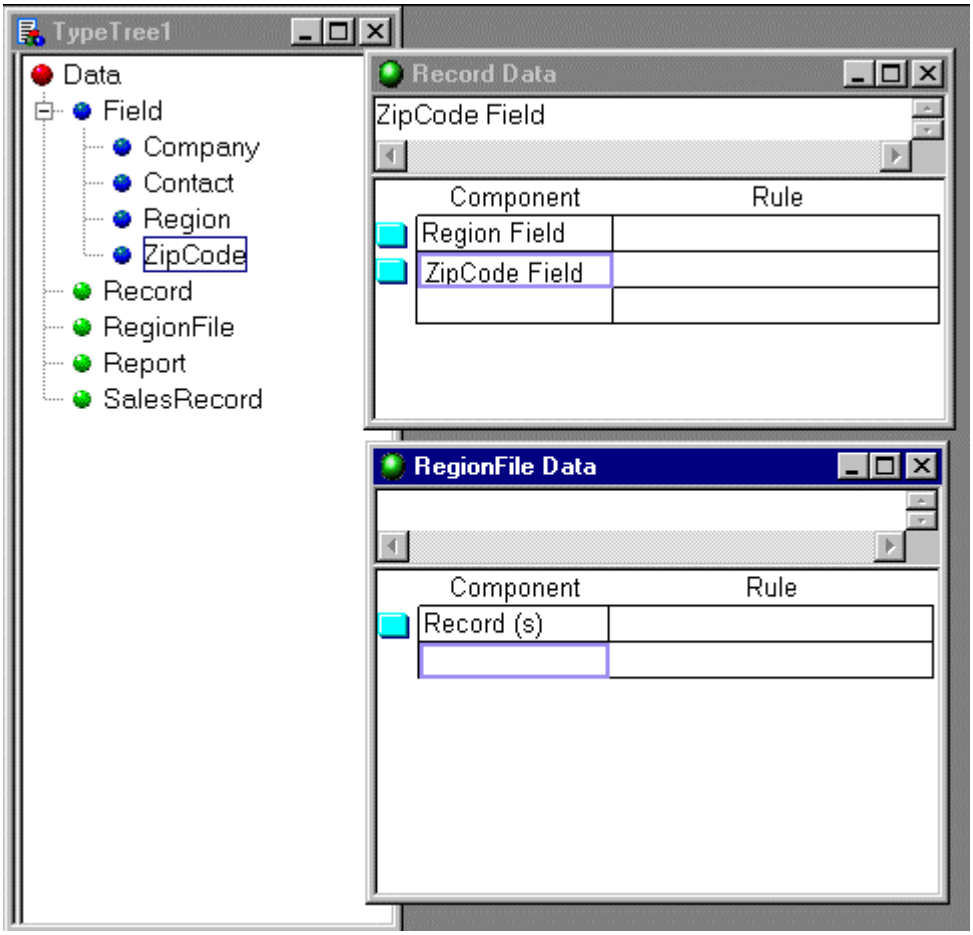
The following table lists the input files to use, and the files to modify and create, in the first example in this chapter.

File	Use
unique.txt	Use as an input data file. This file is created by running the map created in Chapter 3.
region.txt	This output file is created by running the first map. This file is then used as an input data file to another map. The lookup data, zip codes, are not in any order.
customer.mtt	You create this type tree file to define the lookup file and the output file.
customer.mms	You create this map source file.
report.txt	This output file is generated by running the second map created in this example.

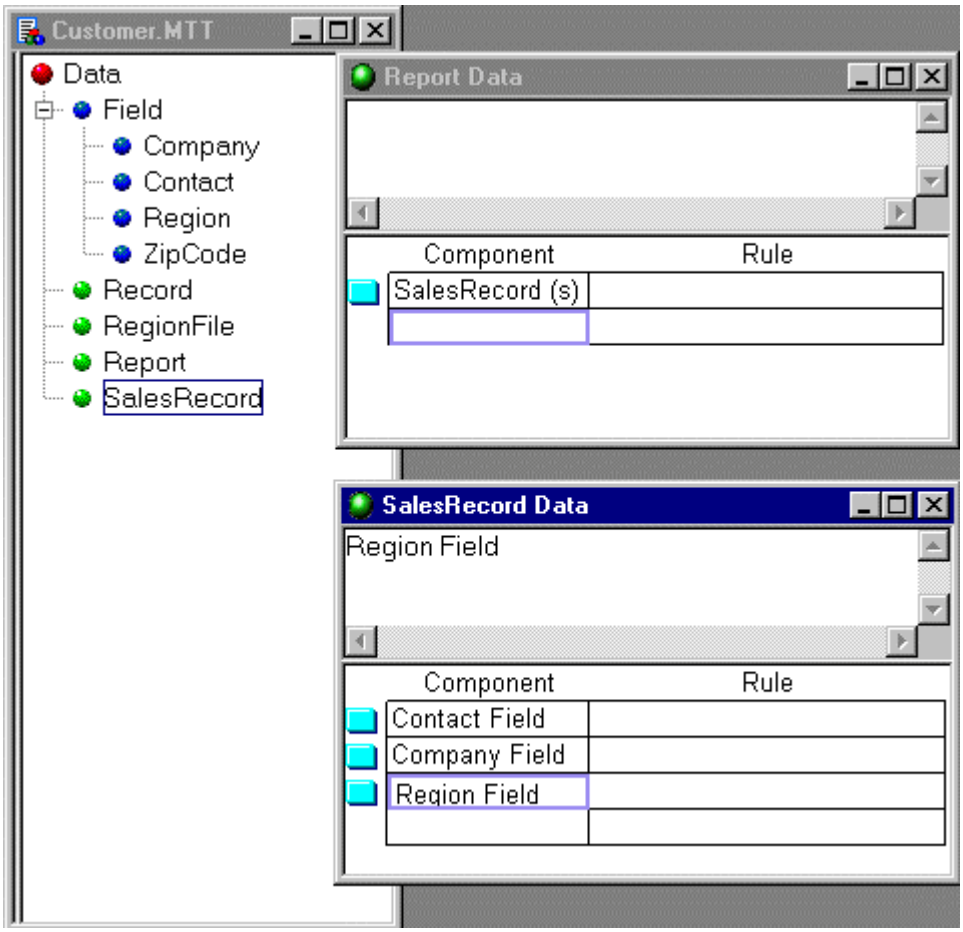
Using the Type Editor

Create a type tree that defines the following data. Define each record as infix delimited with two spaces:

```
60525  North
60089  West
33321  South
60012  North
19444  West
44444  South
```



Now, define the output file. The output file is made up of Sales Records. Each record contains the contact person, the company name, and the region.

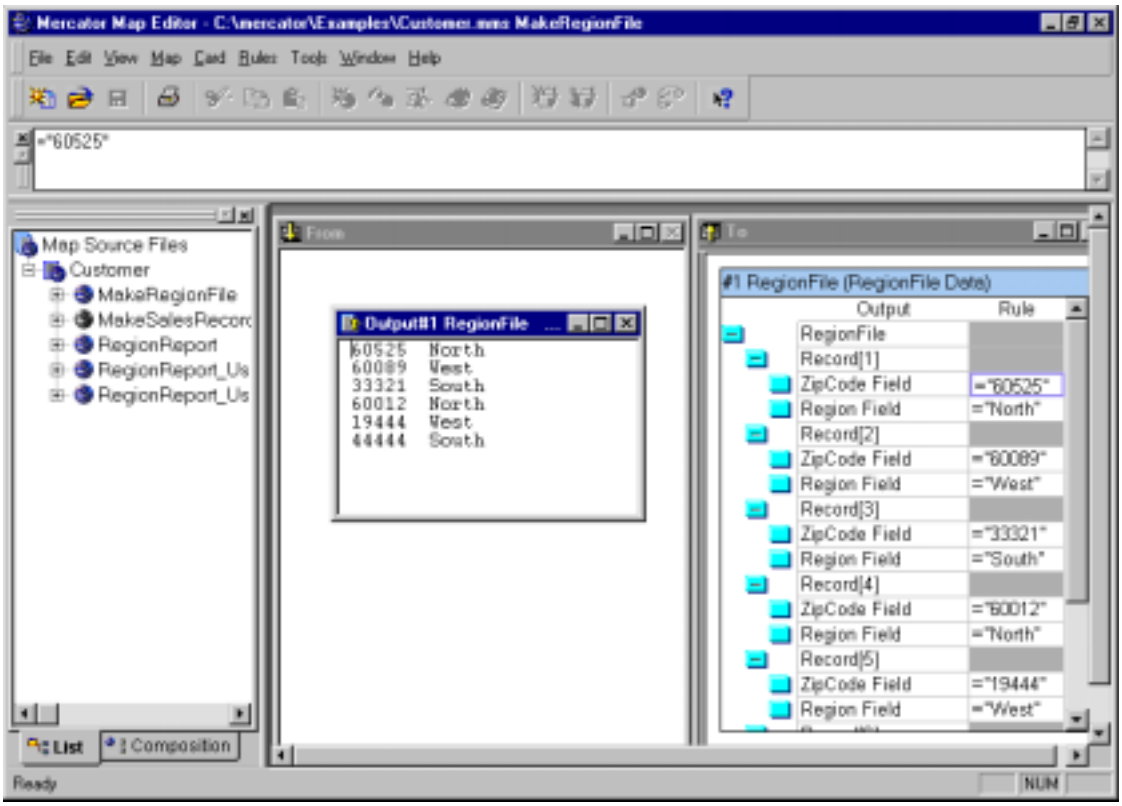


Using the LOOKUP Function

Create a map that generates the RegionFile data:

```
60525  North
60089  West
33321  South
60012  North
19444  West
44444  South
```

Do not create any input cards. Create one output card, whose type is RegionFile. Index six records. Then, simply enter text values for the fields. For more information on indexing an output, see Chapter 5 - Formulating Map Rules in the *Map Editor Reference Guide*.



Now, create a map that uses the unique contact file and the region lookup file as inputs. In the map rule for the output record, use the LOOKUP function. The LOOKUP function is the appropriate cross-reference function to use, because the lookup file is not ordered in any specific way.

In the executable map, the rule on SalesRecord(s) references the functional map MapSalesRecord. The two arguments are Contact, and the record whose ZipCode matches the ZipCode of the Contact. The LOOKUP function is used.

The LOOKUP function has two arguments. The first argument is the series of objects, belonging to one type, to be searched. The second argument is the condition you want to base the search on. LOOKUP sequentially searches objects in the series.

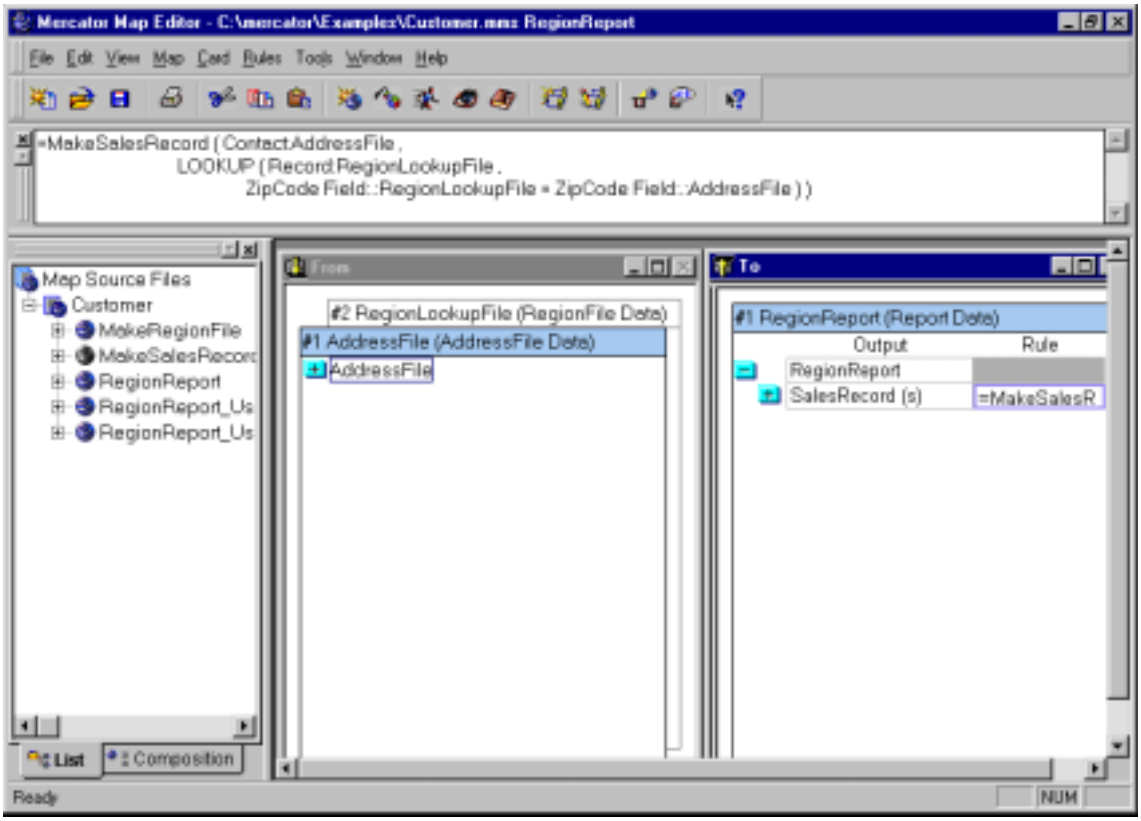
It returns the first object in the series where any corresponding evaluation of the condition is TRUE. The syntax of LOOKUP is:

LOOKUP (*Series you want to lookup, Condition*)

The rule on the SalesRecord output is:

```
=MakeSalesRecord (Contact:ContactFile,
   LOOKUP (Record:RegionLookupFile,
   ZipCode Field:.:RegionLookupFile = ZipCode Field:.:ContactFile))
```

Here is the executable map:



Using the Functional Map Wizard

Now, use the Functional Map Wizard to create the functional map MakeSalesRecord.

To use the Functional Map Wizard

1 Select the rule for which you want to generate the functional map. In this case, it is the rule in the executable map:

```
=MakeSalesRecord (Contact:ContactFile,
   LOOKUP (Record:RegionLookupFile,
   ZipCode Field:.:RegionLookupFile = ZipCode Field:.:ContactFile))
```

2 From the **Rules** menu, choose **Functional Map Wizard**.

Or click the Functional Map Wizard tool. 

The Functional Map Wizard dialog is displayed.

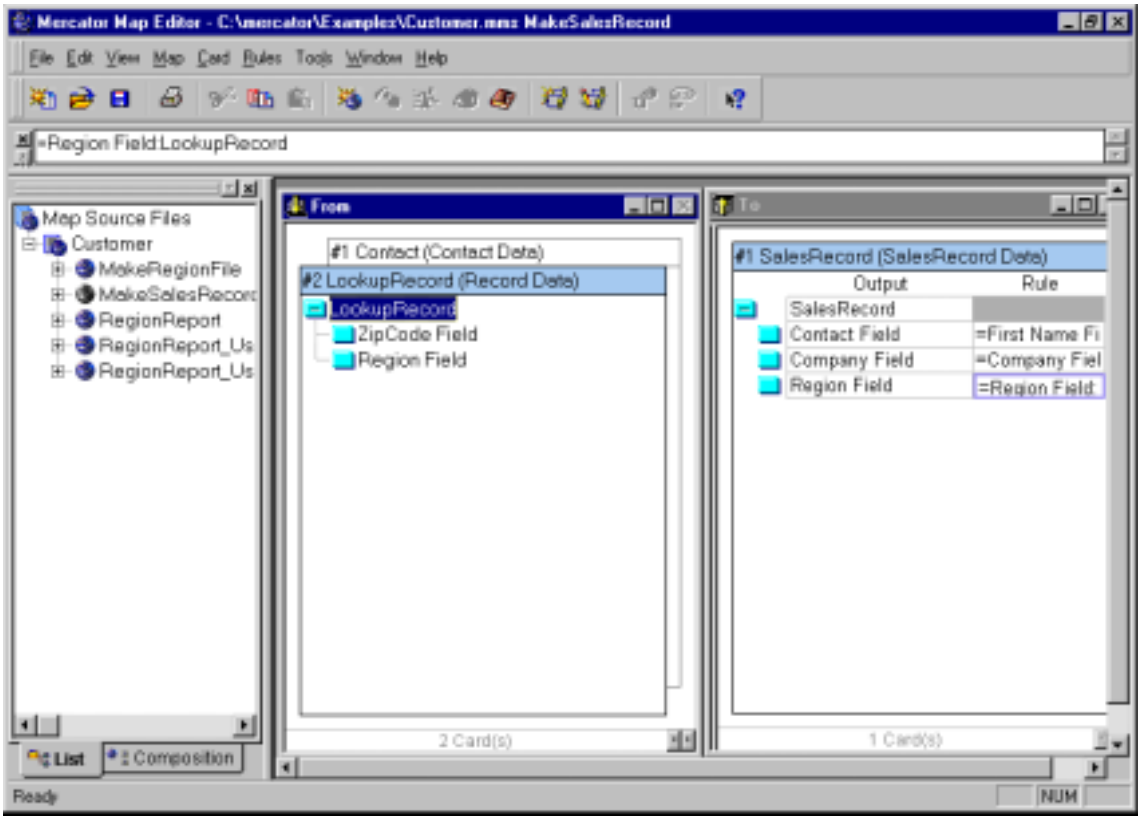
3 In the Functional Map Wizard dialog, name the first input card “Contact”, the second input card “LookupRecord”, and the output card “SalesRecord”.

4 Click the Create button.

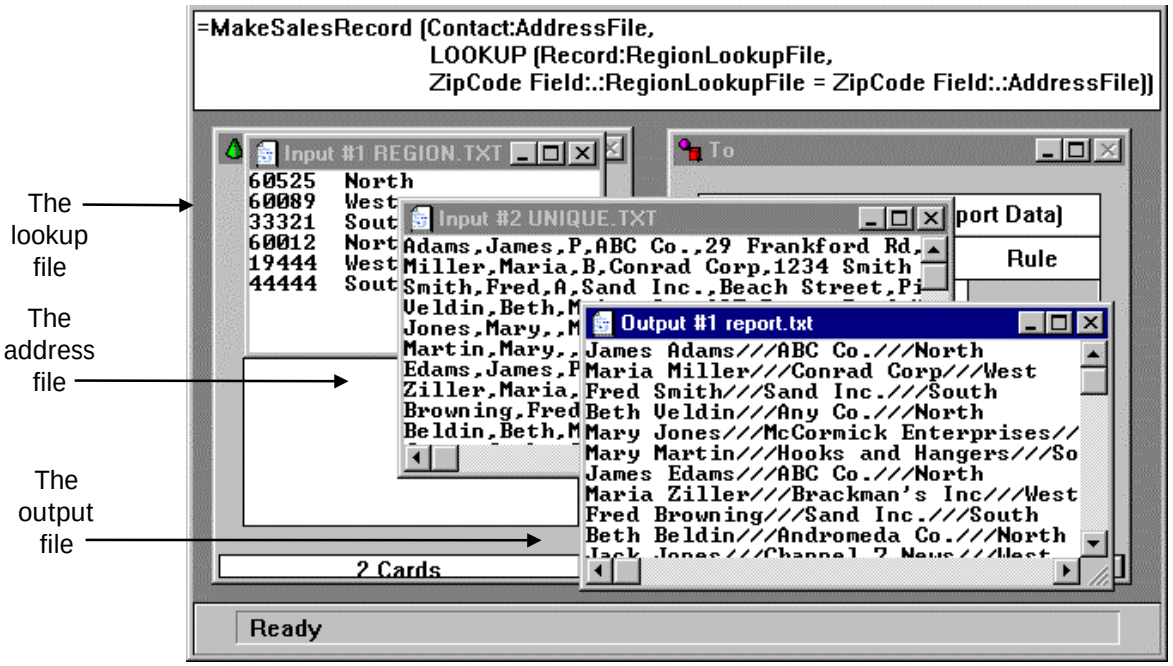
5 Click the Done button.

The Functional Map Wizard will create the functional map MakeSalesRecord.

In MakeSalesRecord, Contact field is created by concatenating the First and Last Name fields from the input. The Company field is simply mapped from the Company field in the input. The Region from the lookup record is mapped to the output Region.



Each record in the output file contains the Contact and the Region - information you mapped from both the address file and the region lookup file.



Case 2 - Using the SEARCHDOWN Function

The SEARCHDOWN function is appropriate to use when the values in the cross-reference file are in descending order. Using the SEARCHDOWN function, instead of the LOOKUP function, optimizes the execution.

The SEARCHDOWN function searches an ordered series of objects (argument #2), using a binary search, by comparing members of the series to another object (argument #3). The ordered series is in descending order. The result (argument #1) is an object related to the ordered series.

The syntax of the SEARCHDOWN function is:

```
SEARCHDOWN (Object you want to get, Ordered series to search,
   Object to compare)
```

Note For SEARCHDOWN to work properly, the second argument must be in descending order.

Files Used in Case 2

The following table lists the input files you use, and the files you modify and create, in the first example in this chapter.

File	Use
unique.txt	Use as an input data file. This file is created by running the map created in Chapter 3.
region3.txt	Input file used for lookup. The lookup data - zip codes - are in descending order.
customer.mtt	This type tree file defines the lookup file and the output file. It is the same type tree used in Case 1.
customer.mms	This is a continuation of the map source file created in Case 1.
report.txt	This output file is created by running the map.

Using the Map Editor

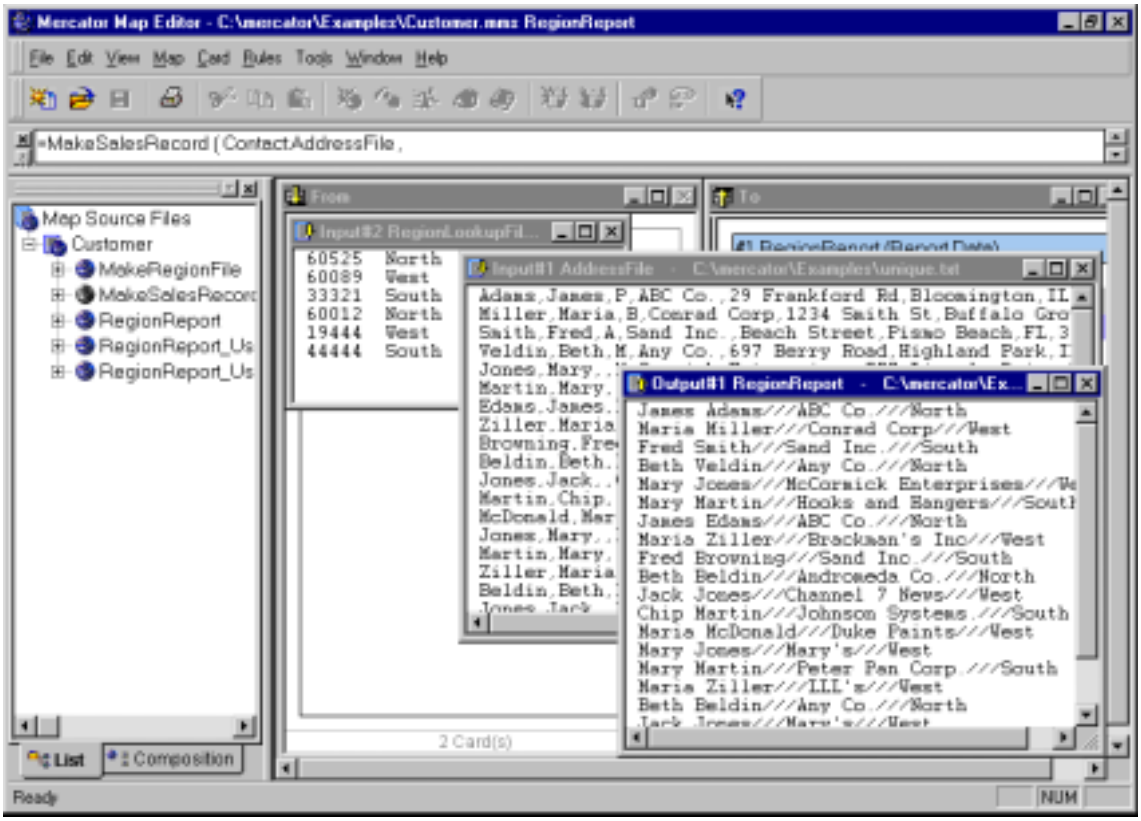
Suppose the Region Lookup file is ordered by the ZipCode Field. The ZipCodes are in descending order. Create a map that generates the following data:

```
60525  North
60089  West
60012  North
44444  South
33321  South
19444  West
```

Now, the second argument to the map MakeSalesRecord is the SEARCHDOWN function:

```
=MakeSalesRecord (Contact:ContactFile,
   SEARCHDOWN (Record:RegionLookupFile,
   ZipCode Field:.:RegionLookupFile, ZipCode Field:.:ContactFile))
```

The results of running the map show the output with each customer Contact and Region:



Case 3 - Using the SEARCHUP Function

Use the SEARCHUP function when the cross-reference file is in ascending order.

The SEARCHUP function searches an ordered series of objects (argument #2), using a binary search, by comparing members of the series to another object (argument #3). The ordered series is in ascending order. The result (argument #1) is an object related to the ordered series.

The syntax of the SEARCHUP function is:

```
SEARCHUP (Object you want to get, Ordered series to search,
   Object to compare)
```

Note For SEARCHUP to work properly, the second argument must be in ascending order.

Files Used in Case 3

The following table lists the input files to use, and the files to modify and create, in the first example in this chapter.

File	Use
unique.txt	Use as an input data file. This file is created by running the map created in Chapter 3.
region9.txt	You create this input file. The lookup data - zip codes - are in ascending order.
customer.mtt	This type tree file defines the lookup file and the output file. It is the same type tree used in Cases 1 and 2.
customer.mms	This is a continuation of the map source file created in Case 1.
report.txt	This output file is created by running the map.

Using the Map Editor

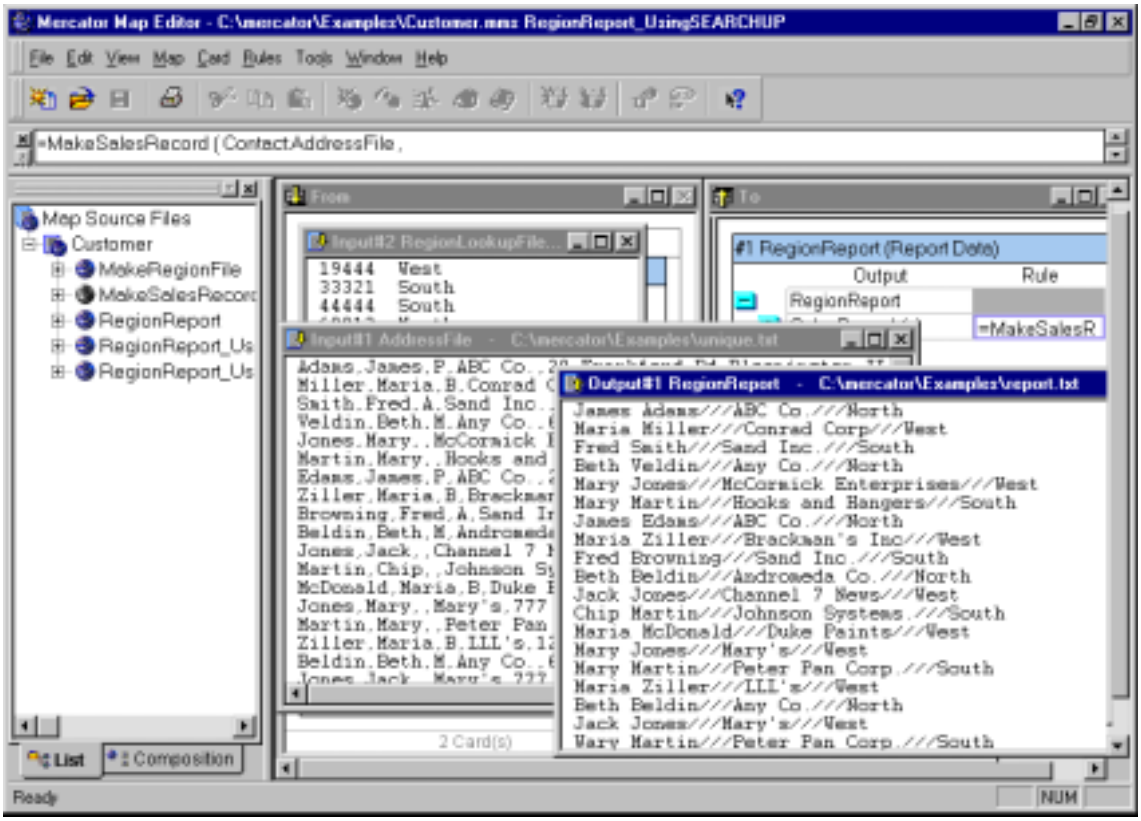
Suppose the Region Lookup file is ordered by the ZipCode Field. The ZipCodes are in ascending order. Create a map that generates the following data:

```
19444  West
33321  South
44444  South
60012  North
60089  West
60525  North
```

The second argument to the map MakeSalesRecord is the SEARCHUP function.

```
=MakeSalesRecord (Contact:ContactFile,
   SEARCHUP (Record:RegionLookupFile,
   ZipCode Field:.:RegionLookupFile, ZipCode Field:.:ContactFile))
```

When you run the map and view the results, you see:



Case 4 - Using the CHOOSE Function

Use the CHOOSE function when you want to select an object at a given index in a series.

The CHOOSE function selects an object from a series. The second argument is the index of the object chosen. The syntax for CHOOSE is:

```
CHOOSE (Series you want to choose from, Index of the object you
want to choose)
```

What You Want to Do

Suppose you conduct a raffle. You have a file listing the winners in the raffle. The first person in the file won first place, the second person won second place, and so on.

Here is the file of winners, in order. Helen is the first prize winner, Florence is second, and so on:

```
Helen
Florence
Karen
Glenn
Betty
Robert
Toby
```

You also have a file listing the raffle prizes, and the place a winner must get, to receive the prize.

Here is the prizes file:

```
Hot tub, 4
Couch, 6
Stereo, 3
Dog bone, 7
House, 1
Set of luggage, 5
Car, 2
```

You want to generate a file that notifies each winner of their prize.

You want the output file to look like this:

```
Congratulations, Somebody1!  You have won a PRIZE X
Congratulations, Somebody2!  You have won a PRIZE Y
```

How to Do It

Define the three files in a type tree - the prize file, the winner file, and the output file.

Create a map that uses the prize and winner files as inputs. Use the CHOOSE function to match a prize with the appropriate winner.

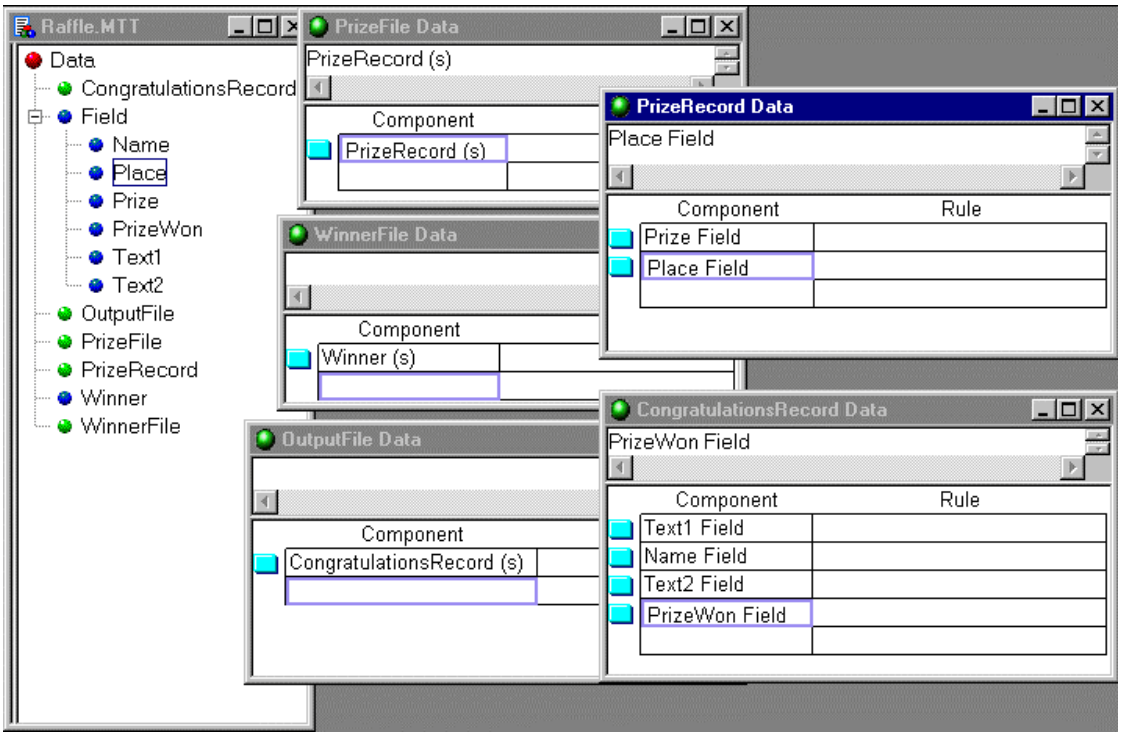
Files Used in Case 4

The following table lists the input files to use, and the files to modify and create, in this Case 4 example.

File	Use
prizes.txt	You create this file to use as an input data file.
winners.txt	You create this file to use as an input data file.
raffle.mtt	You create this type tree file, which defines the prizes file, the winners file, and the output file.
raffle.mms	You create this map source file.
output.txt	This output file is created by running the map.

Using the Type Editor

The type tree looks like this:



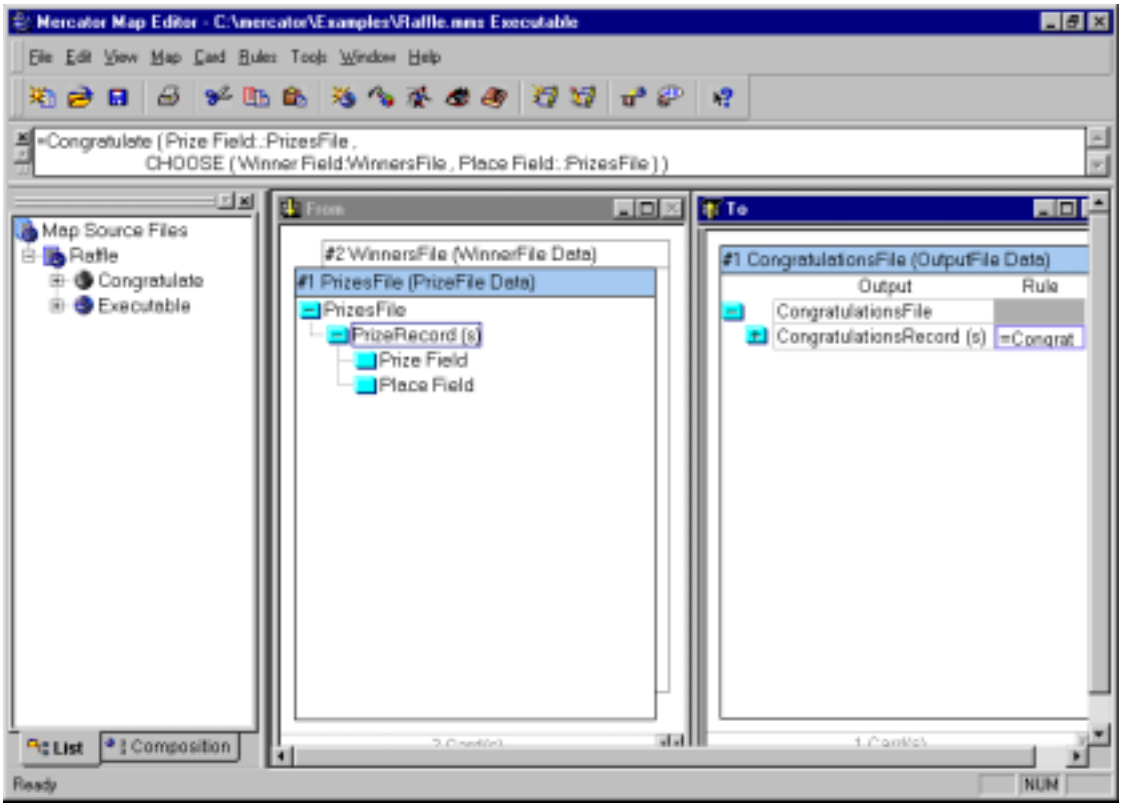
Using the Map Editor

In the executable map, the map rule on the output CongratulationsRecord references a functional map, Congratulate. The two inputs to this map are a prize from the prize file, and the winner whose place in the raffle matches the place of the prize.

The map rule for CongratulationsRecord is:

```
= Congratulate (Prize Field:.:PrizesFile,
   CHOOSE (Winner:WinnersFile, Place Field:.:PrizesFile))
```

The executable map is:



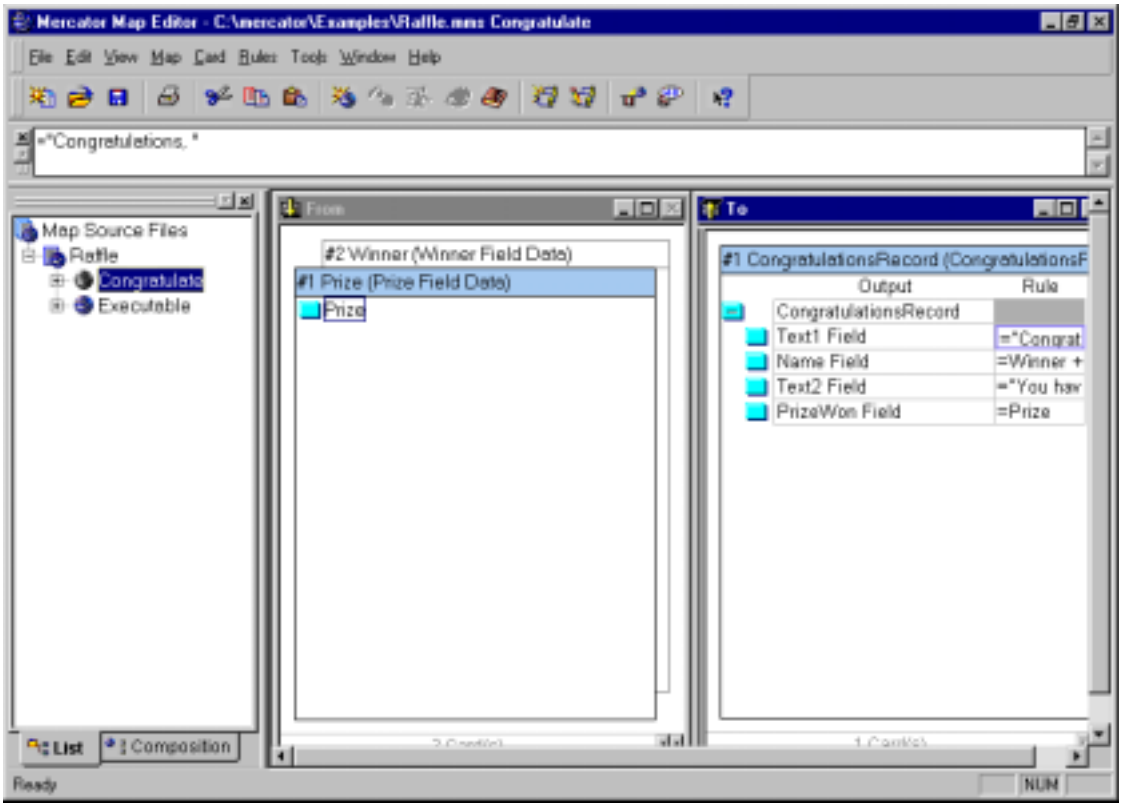
Now, you would create the functional map “Congratulate”. You can do it manually, or use the Functional Map Wizard.

Using the Functional Map Wizard

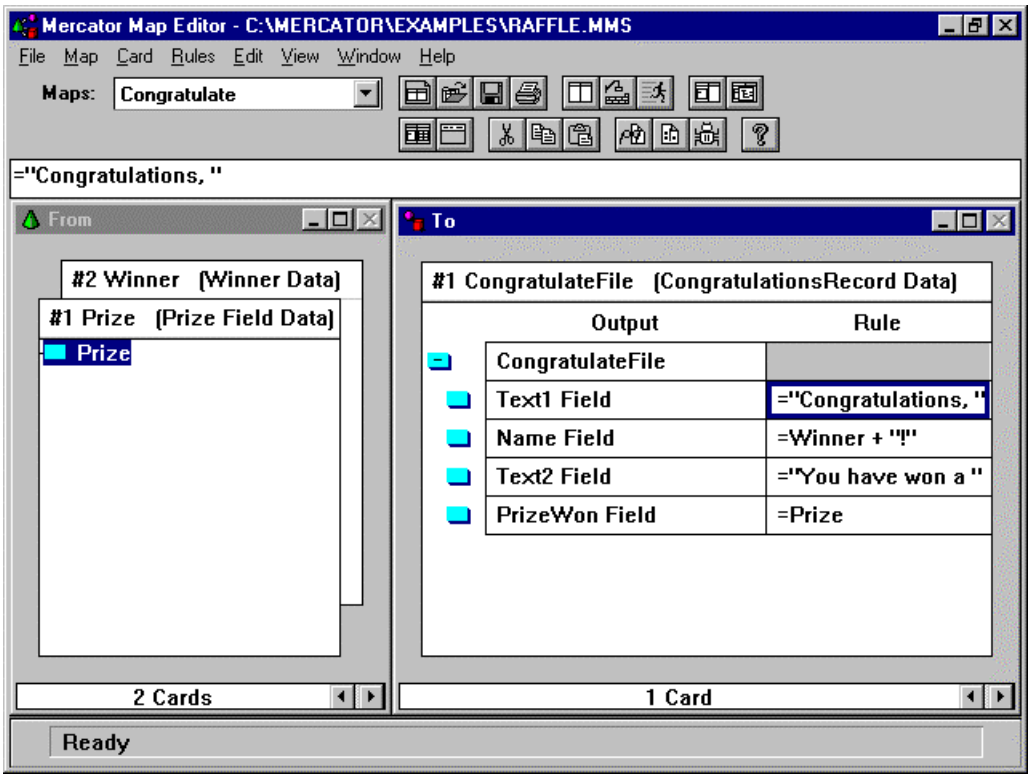
1 Select the rule and choose the Functional Map Wizard tool.

2 Name the first input card “Prize”, and the second input card “Winner”.

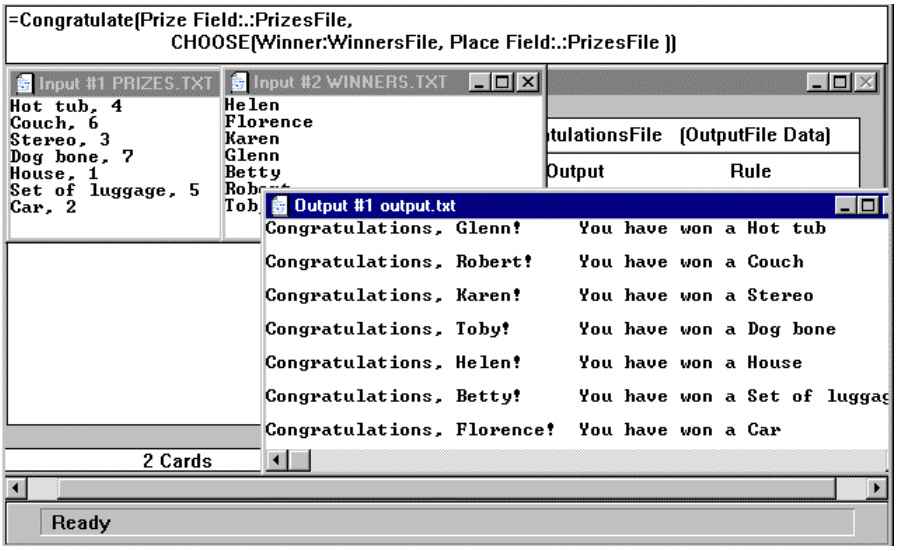
The Functional Map Wizard creates the map Congratulate:



In the functional map Congratulate you enter the text “Congratulations, ” in the Text1 field, and “You have won a” in the Text2 field. The name of the winner and the prize are mapped from the winner and the prize in the input.



After entering the map rules for Congratulate, you select the executable map, build the map, and run it. The results show that each winner was matched with an appropriate prize. Toby, the dog, won the dog bone:



See Chapter 7 - Using the Functional Map Wizard in the *Map Editor Reference Guide* for information on how the Functional Map Wizard creates the functional map.

Chapter 7 - Using Control-Break Logic to Define Data

Sometimes you have data with physical characteristics - such as delimiters, or the length of a fixed object - to determine where one data object ends and the next begins. If the data has delimiters or is a fixed length, you can define these characteristics in the Mercator Type Editor.

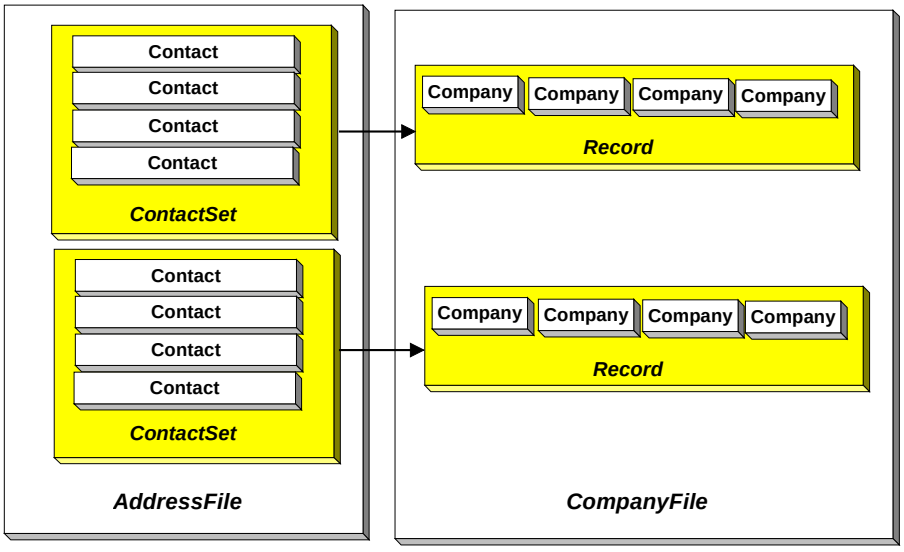
Sometimes data is not fixed, or it may not contain delimiters. Other methods are required to define a break in the data. This chapter contains two examples of using other methods to define a break. The first example shows how to define a break in the data pattern when the count of a certain object reaches a specific value. The second example shows how to define a break in the data pattern when the value of a certain Item changes.

Case 1 - Breaking Data by Counting Objects

Sometimes, you may want to define a specific number of data objects as a single data object.

What You Want to Do

Suppose you need Mercator to recognize each set of four Contacts as a single data object, because you want to map each set of four to a single Record in your output.



Within each record, there are four Company fields. You want to map the Company field of one Contact, to one of the Company fields in Record.

How to Do It

In the Type Editor, define the address file in terms of ContactSets, and use a component rule on the Contact component of ContactSet. Define the company file of records.

In the Map Editor, create a map that maps the address file to the company file. Use a functional map to map a single ContactSet to a single Record.

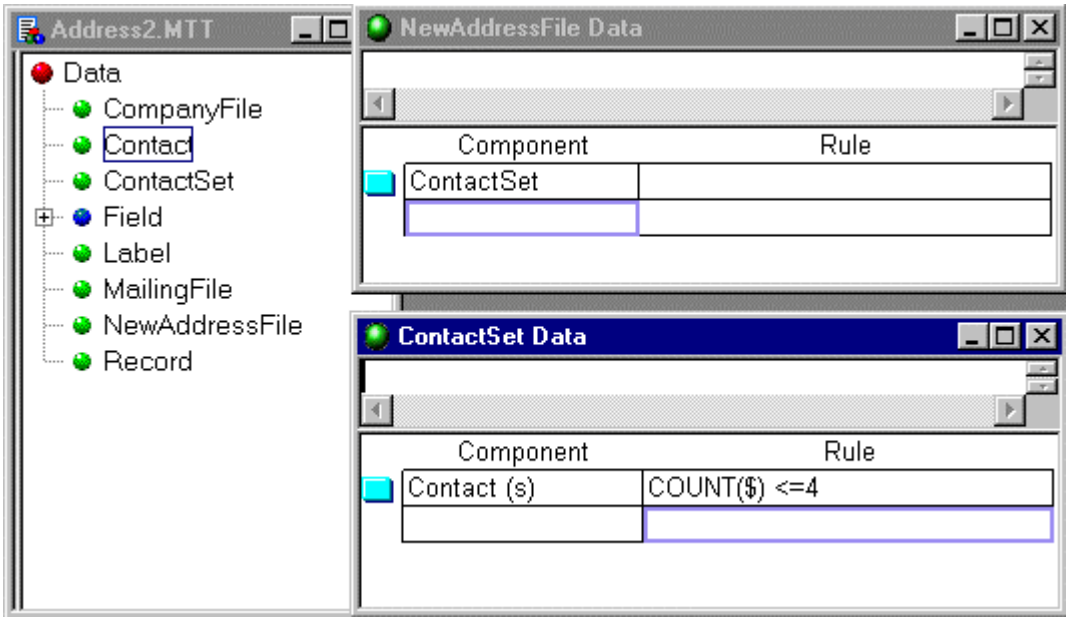
Files Used in Case 1

The following table lists the input files to use, and the files to modify and create, in this example.

File	Use
address.txt	Use as an input data file. This file is in your mercator\examples directory (folder in Windows 95)
address2.mtt	This type tree file defines the address file, the company file, and the output file.
customer.mms	This is a continuation of the map source file created in Chapter 5.
app.txt	This output file is created by running the map.

Using the Type Editor

Open the type tree *address.mtt* and save it as *address2.mtt*. Add the types CompanyFile, ContactSet, and Record. Rename the type “ContactFile” to “NewContactFile”. Then define it with a series of ContactSet(s) as its component.



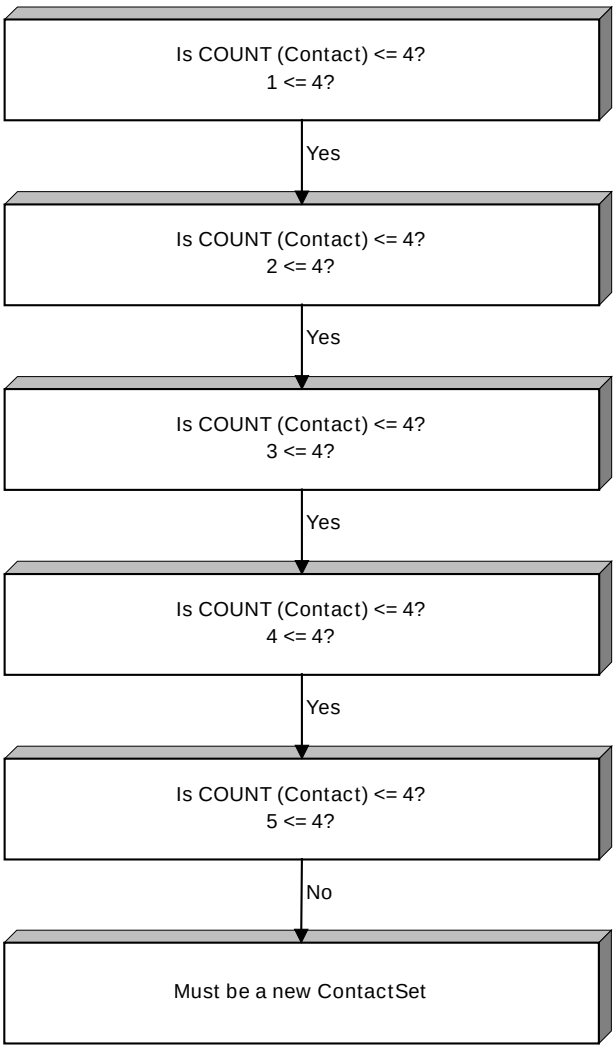
The rule on the Contact(s) component of ContactSet ensures that a ContactSet contains up to four Contacts.

The component rule is:

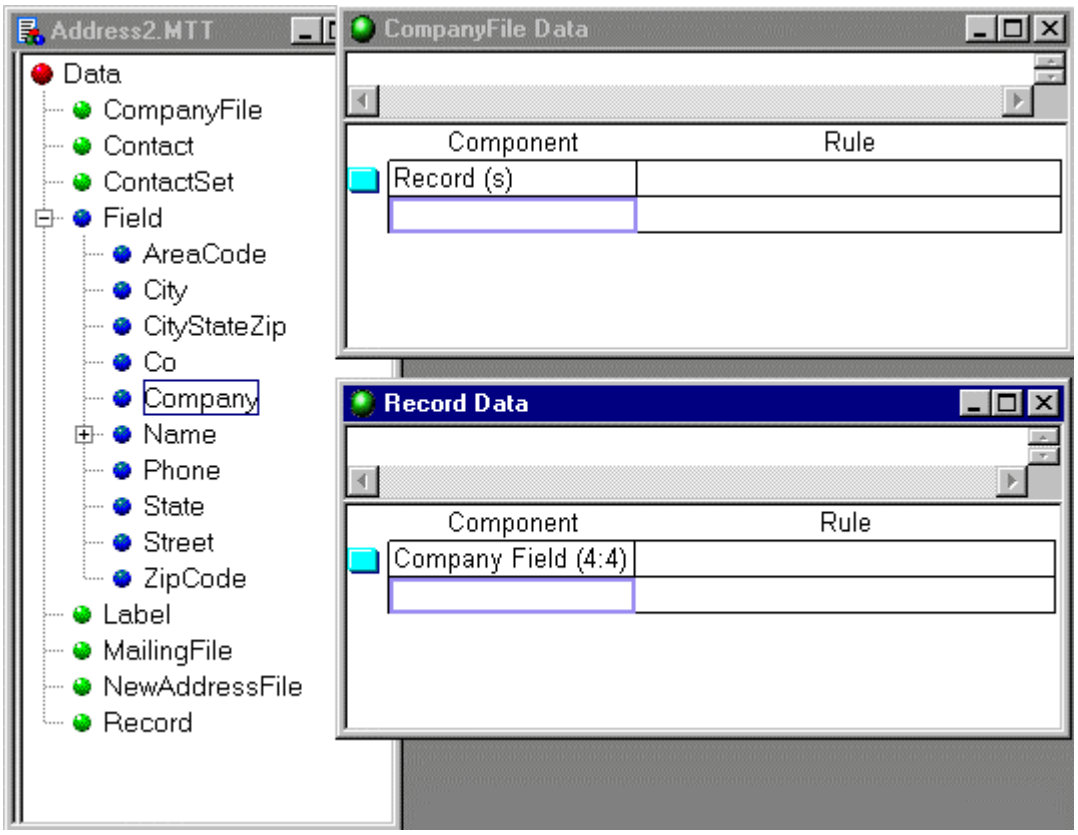
```
COUNT ($) <=4
```

Mercator recognizes each set of four Contacts as a ContactSet. If the number of Contacts in the file is not divisible by four, the last Contacts are still considered a ContactSet, because the component rule allows for the possibility that a set may have less than four.

This diagram illustrates how Mercator behaves as it proceeds through the data:

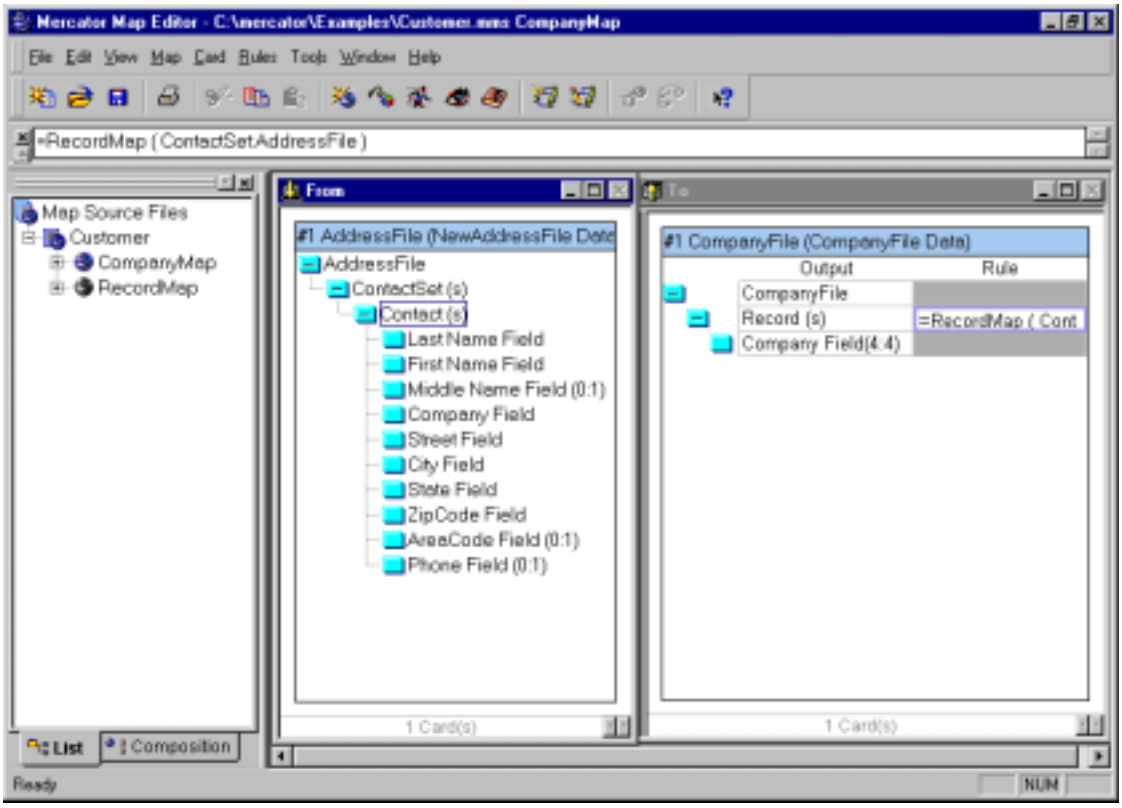


The output type CompanyFile is made up of Record(s). A Record is made up of four Company fields.



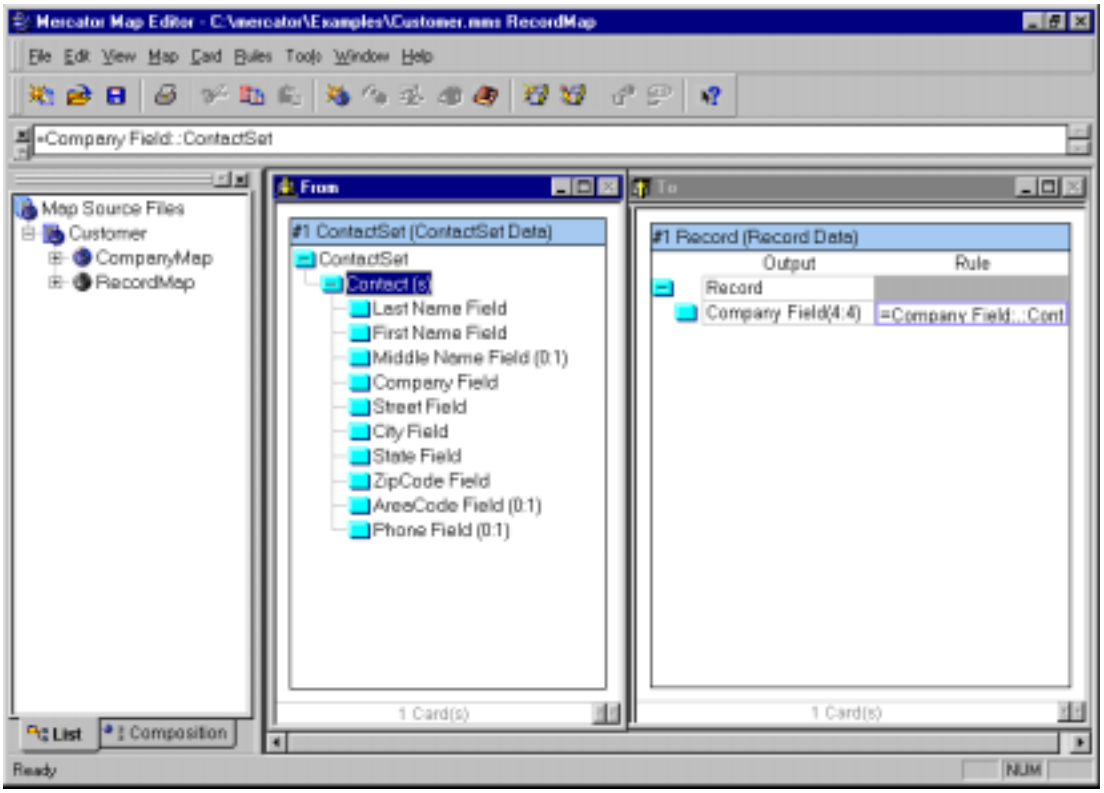
Using the Map Editor

You want to map the file of ContactSets to the company file of Records. Within each Record, there are four Company fields. First, create the executable map CompanyMap.

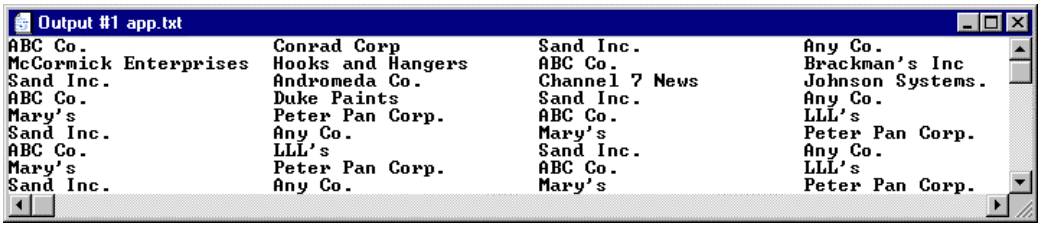


A functional map, RecordMap, is used to generate one output Record per ContactSet.

In RecordMap, the Company field of each Contact is mapped to the Company field in the Record.



After you run the map, you see that each Record in the output file is made up of four Company fields from the input.



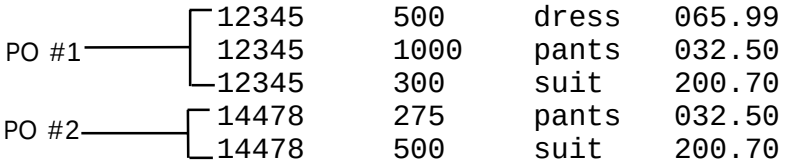
Case 2 - Breaking Data by a Change in a Data Value

Sometimes, the thing that determines where one object ends and the next begins is a change in the value of a certain field.

What You Want to Do

Suppose you have a file of purchase orders. Each purchase order (PO) is made up of line item records - one record per item. The PO number appears at the beginning of each record. Therefore, the PO is a set of consecutive line item records that have the same PO number. How would you define this kind of data in the Type Editor?

In your data each line item record contains a PO#, Quantity, Item, and Price field. The data looks like this:



Suppose you want to generate a file that contains just one field - which gives the count of the POs in the input file.

How to Do It

Define the file of purchase orders, using a component rule to distinguish one PO from the next. Define the simple output file.

Create a map that maps the PO file to the simple count file.

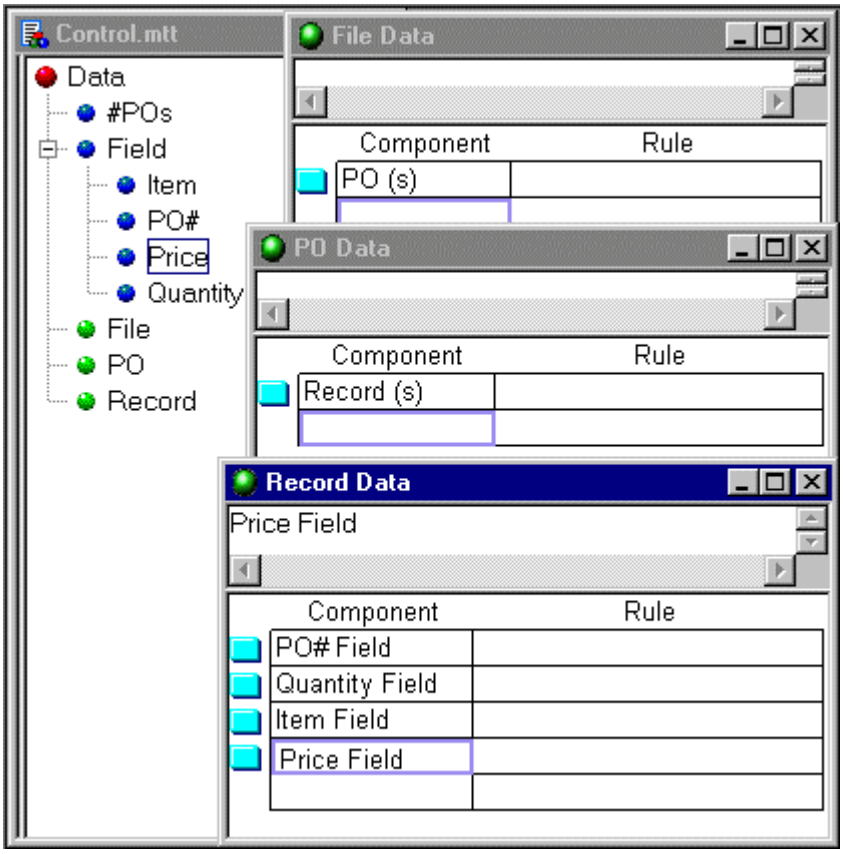
Files Used in Case 2

The following table lists the input files to use, and the files to modify and create, in this example.

File	Use
cntrl.txt	You create this file to use as an input data file.
control.mtt	You create this type tree file to define the input data and the output data.
control.mms	You create this map source file.
countpo.txt	This output file is created by running the map.

Using the Type Editor

Create a type tree to look something like this:



Mercator must be able to recognize the PO data object. If you create this tree and analyze it, you get an error. Mercator tells you that it cannot distinguish one PO from the next.

You can tell the difference between one PO and the next by noticing when the PO number changes. You can use a component rule to bind together, into a single PO, all the Records that have the same value for the PO# Field.

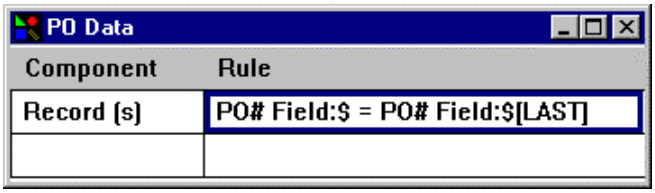
Suppose that Mercator is looking at a given record in the data stream. The component rule says that the PO# of the given record is equal to the PO# in the previous record of that PO. If it is *not*, Mercator knows that the given record is part of a different PO.

Use a rule on the Record(s) component of PO. Remember, whenever you see a colon (:) in a rule, that means component. When used in a component rule, the index value [LAST] refers to the last occurrence of Record that was found.

The component rule is:

```
PO# Field:Record = PO# Field:Record[LAST]
```

You can use **$** as a shorthand notation for Record, so the component rule looks like the one below.



Using the component rule in this way defines a PO as all the consecutive Records that have a PO# matching the PO# in the previous Record.

```
12345       500        dress          065.99
12345       1000       pants          032.50
12345       300        suit               200.70
14478        275       pants          032.50
14478       500        suit           200.70
```

How does Mercator handle the data you have? Mercator looks at the first record, and checks that the component rule comes out to be true.

Note The index value [LAST] is interpreted as [1] when there are no previous occurrences. For example, on the very first Record in the PO file, the component rule is interpreted as:

```
Is PO# Field:Record[1] = PO# Field:Record[1]?
```

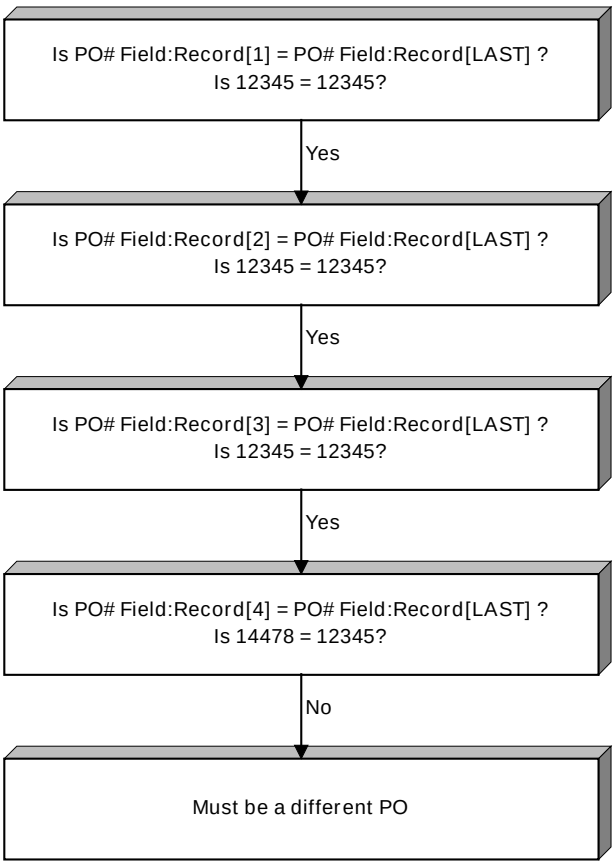
```
Is PO# Field:Record[1] = PO# Field:Record[LAST]?
Is 12345 = 12345?                    Yes.
```

Next, Mercator looks at the second record.

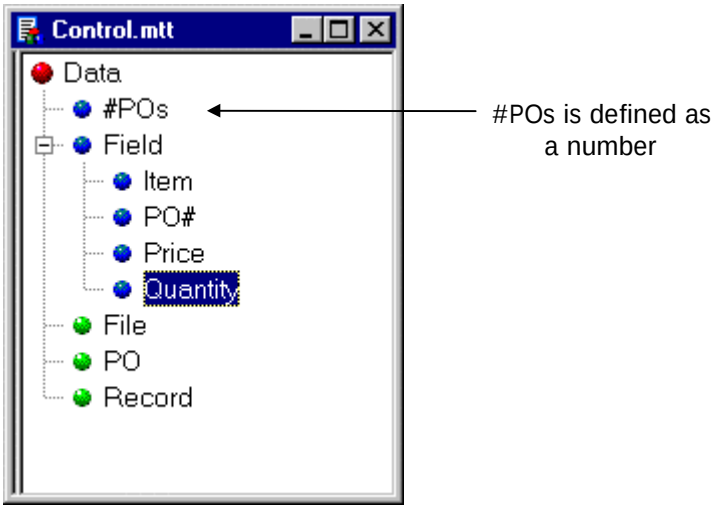
```
Is PO# Field:Record[2] = PO# Field:Record[LAST]?
Is 12345 = 12345?                    Yes.
```

etc.

The diagram below shows how Mercator evaluates the component rule for the first four records in the data. As a result, Mercator knows that the first three records make up a single PO.



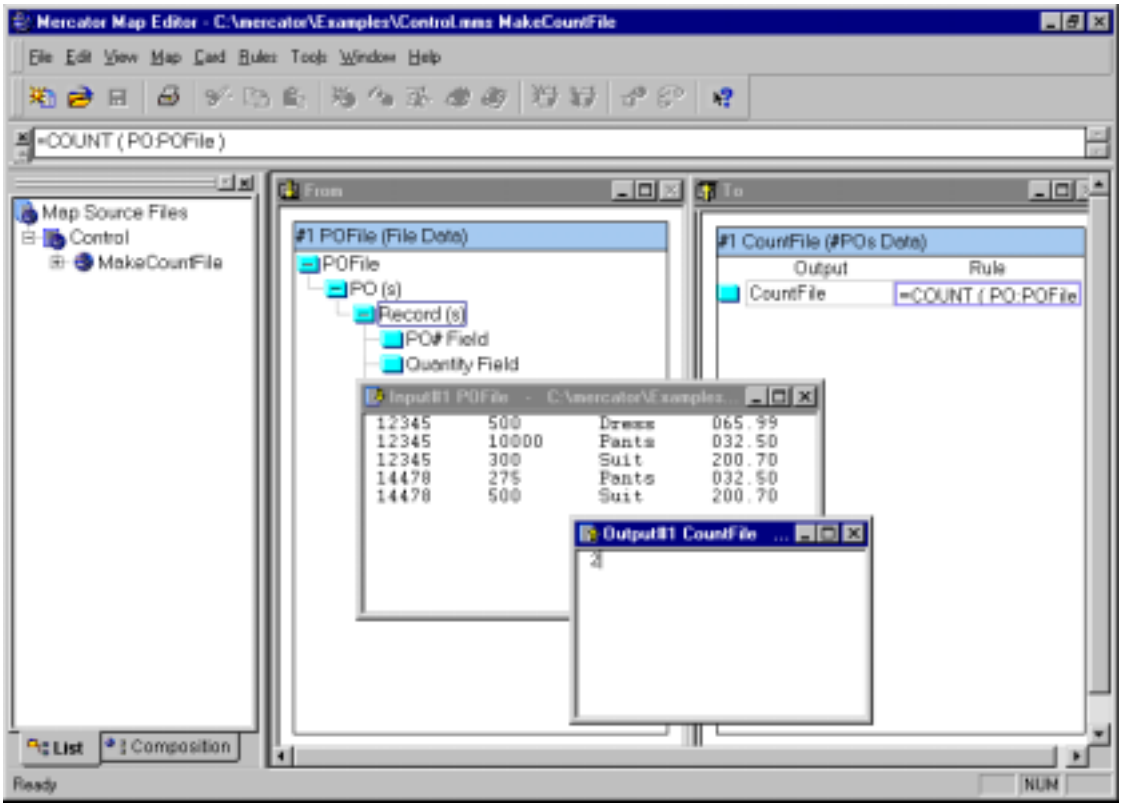
For the output file, you define a numeric Item.



You will use the #POs Item as your output card type in the Map Editor.

Using the Map Editor

In your map, you use the COUNT function to count the POs in the input file. You see that the control-break logic allowed Mercator to recognize the correct number of POs in the file - two.



Chapter 8 - Using Partitioning to Simplify Map Rules

This chapter explains how to partition types, to make your map rules simpler.

What You Want to Do

Suppose you have a file containing a collection of records, in no particular order. Each record can come from one of three kinds of business partners - customers, suppliers, or distributors. In addition, each record comes from one of three different applications - a forecast, purchase order, or invoice application.

A particular field in each record tells you what business partner it came from, and another field tells you which application it belongs to.

Suppose you want to create a file of only invoices to send to your accounting department, and a file of only purchase orders and forecasts to send to your order entry department.

In addition, you want to send the MIS department a report on the activity of your customers, suppliers, and distributors.

How to Do It

In the Type Editor, define the data file of records. Partition Record into the different kinds of records. In addition, define the activity report.

In the Map Editor, create a map that has two output cards - one for the accounting department file, and one for the order entry department file. Create another map to generate the activity report.

Files Used in this Example

For this example, use the following files, which are in your mercator\examples\general\deliver directory (folder in Windows 95).

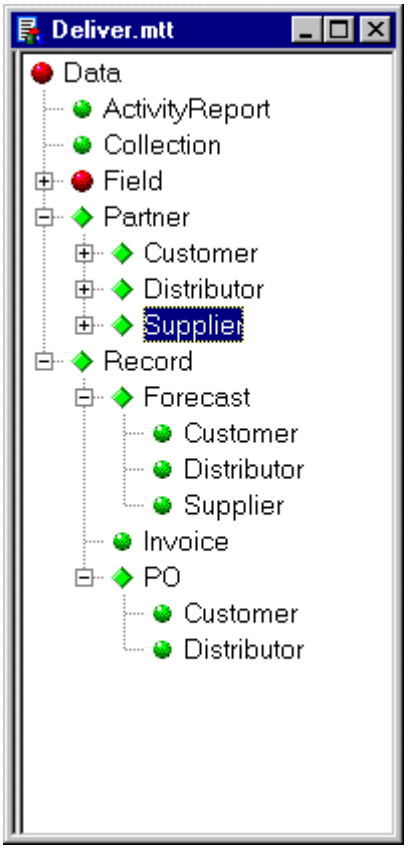
File	Use
deliver.txt	Use this file as input data.
deliver.mtt	This type tree defines the data files.
deliver.mms	This file contains the map explained in this

example.

report.txt This output file is created by running the map.

Using the Type Editor

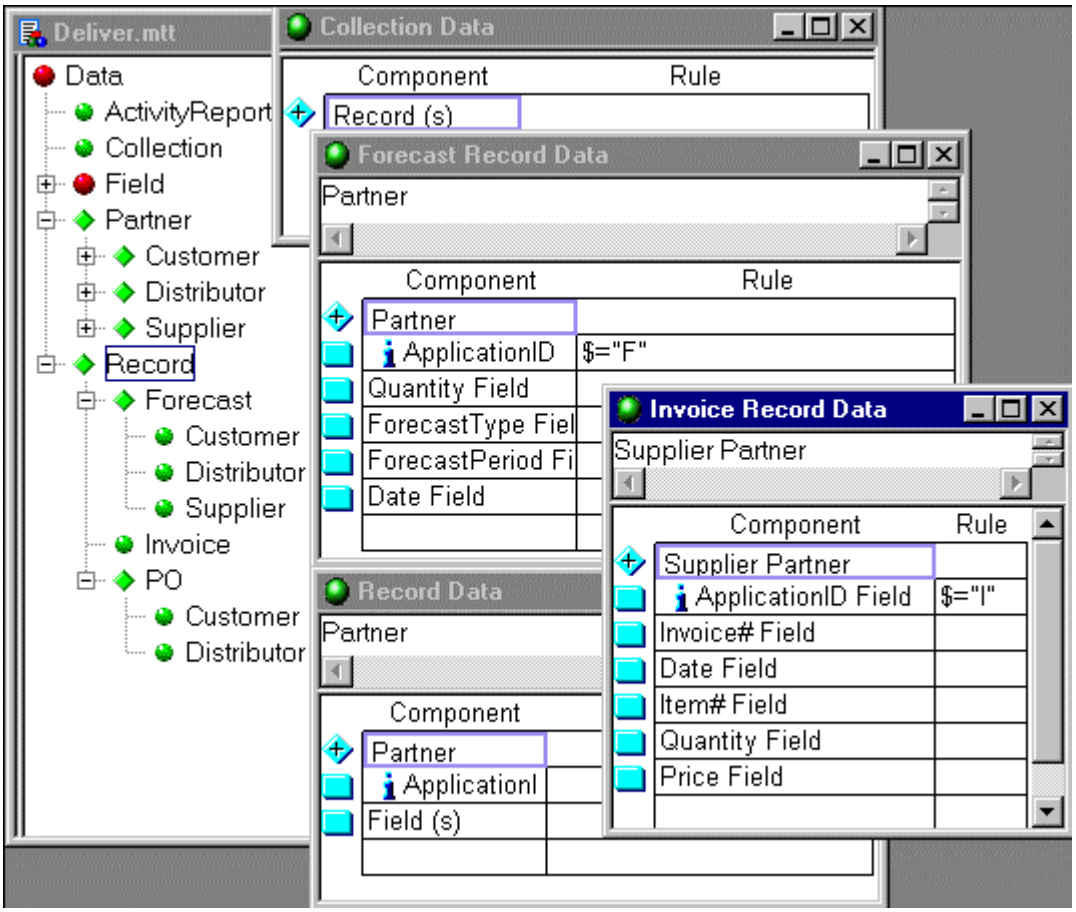
The type tree **deliver.mtt** defines both the input and output data. The type Record is partitioned into Forecast, Invoice, and PO. Then, Forecast and PO are partitioned even further. All in all, there are six different kinds of records.



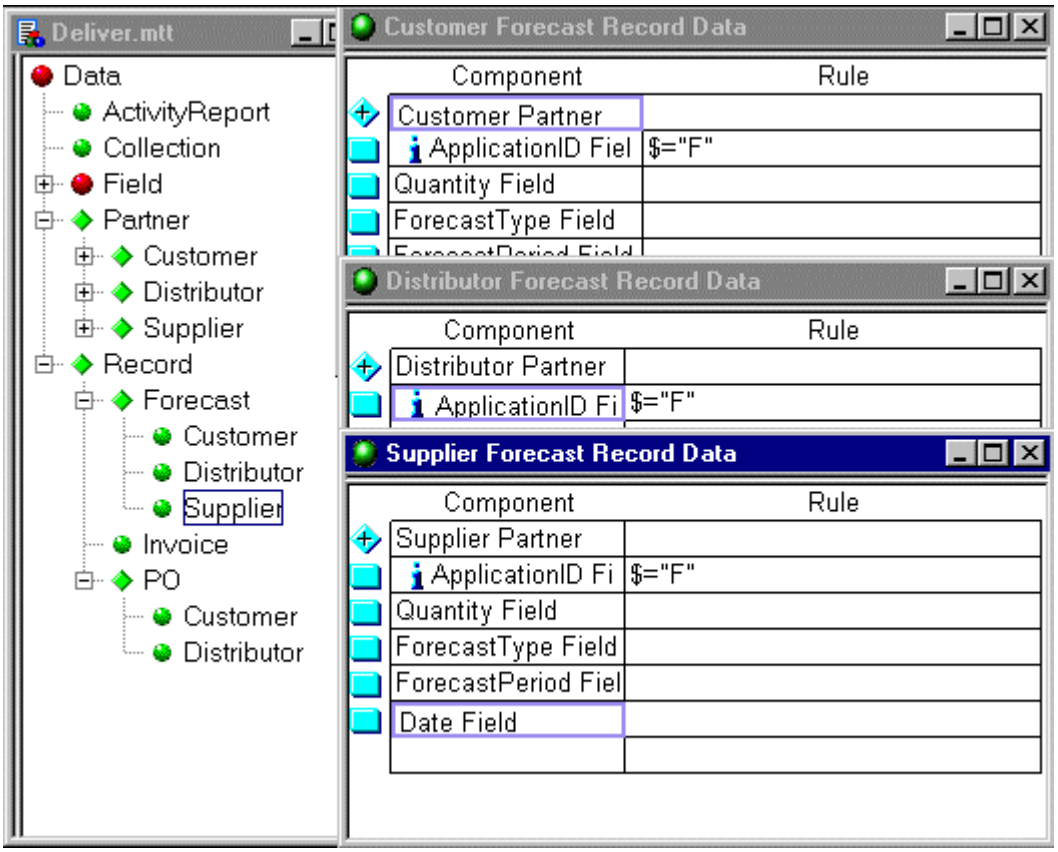
The file is made up of records in a random order. It is defined as the type Collection.

To determine whether a Record is a Forecast, Invoice, or PO record, you can look at the value of the ApplicationID. The Identifier attribute on the component ApplicationID tells Mercator that the components up to, and including, ApplicationID can be used to distinguish record types.

A component rule tells Mercator that the ApplicationID for Forecast is “F,” for Invoice “I,” and for PO “P.”

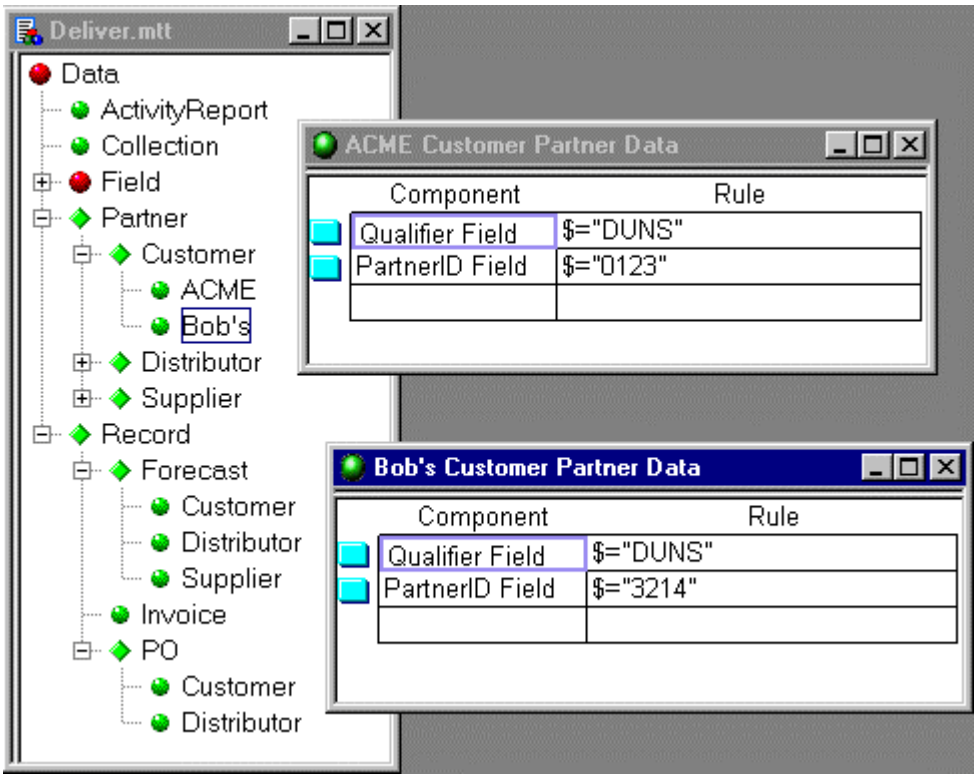


On the next level of partitioning, the types can be distinguished by the first component. For example, the first component of a Customer Forecast record is Customer Partner, but the first component of a Distributor Forecast record is Distributor Partner.



The type Partner is the first component of each record. It is partitioned by Customer, Distributor, and Supplier, and then by specific companies.

The component rules on each company's partner type give the values for each field. In this way, Mercator knows when it is looking at the ACME customer's record, for example.



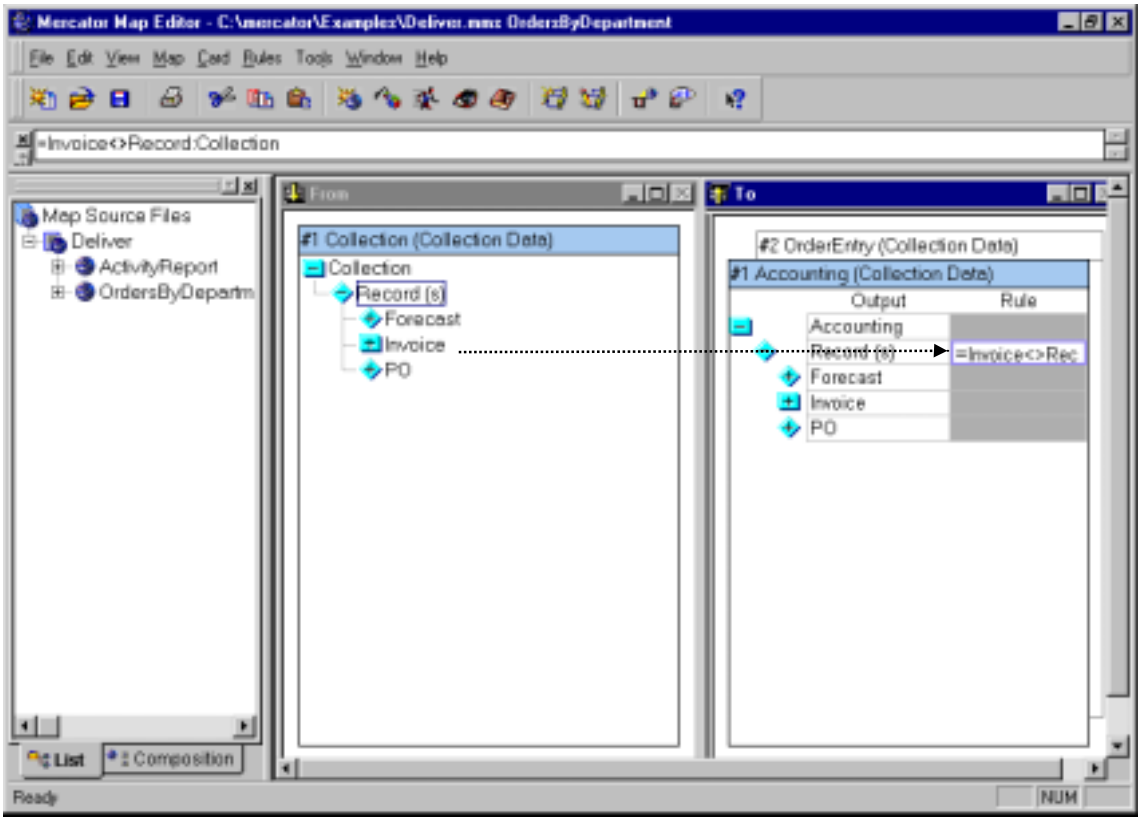
Using the Map Editor

There are two executable maps in this example.

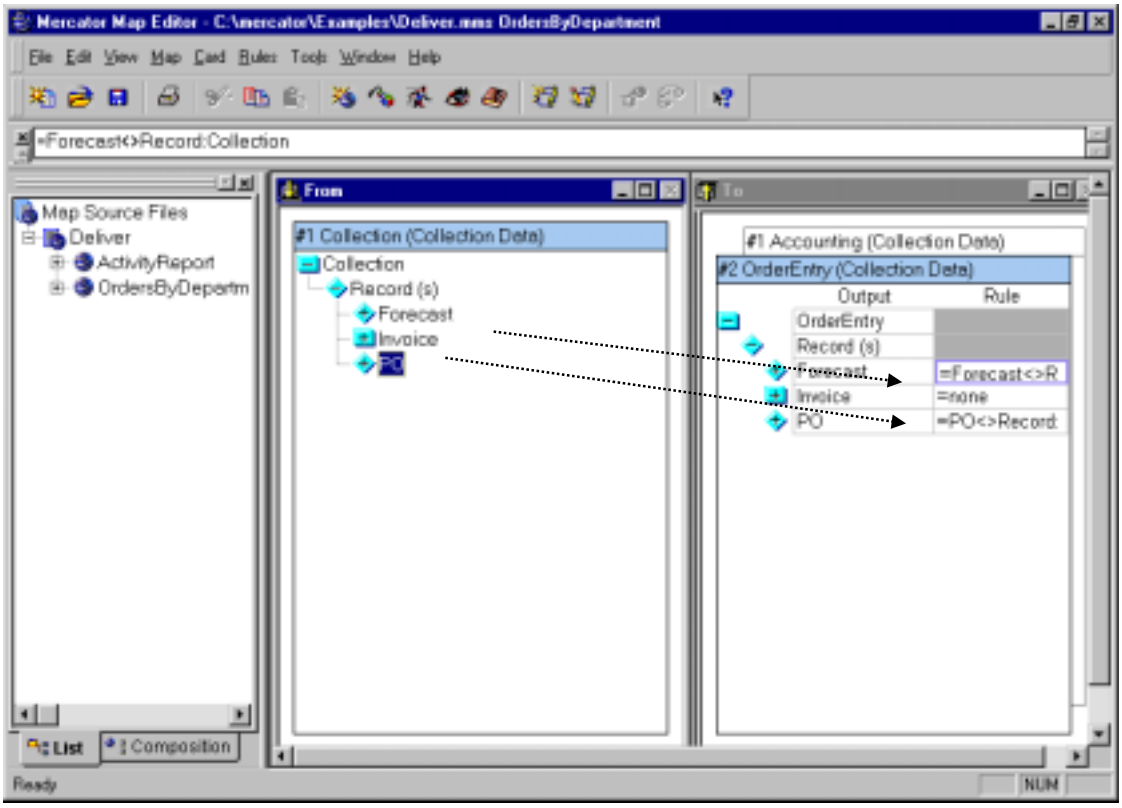
OrdersByDepartment

In the executable map OrdersByDepartment, the input file is mapped to two output files. One output file is for the Accounting department. The other output file is for the Order Entry department. Each file is defined by the same type, Collection.

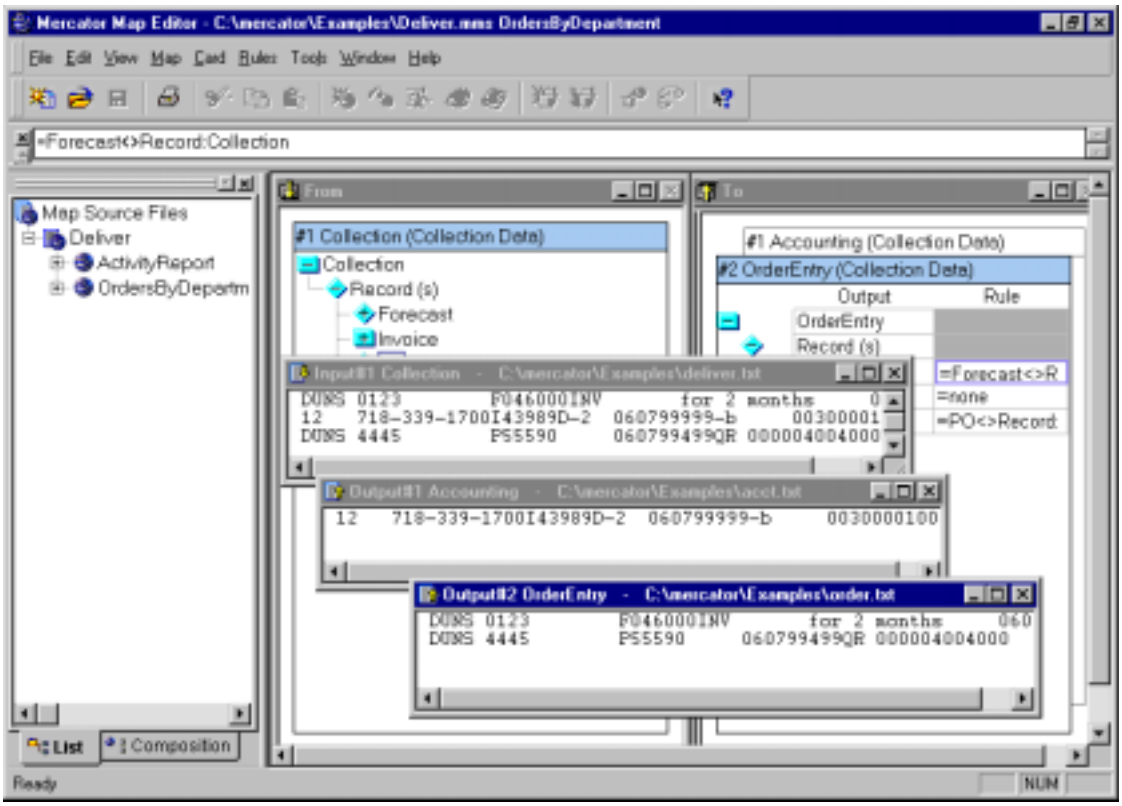
To generate the records in the Accounting output, you want to map the Invoice records from the input. The input Record is the same as the output Record, and Record is partitioned, so you can drag and drop Invoice record from the input to Record in the output.



To generate the records for the OrderEntry output, you drag and drop Forecast to Forecast, and PO to PO. In this file, you do not want any Invoice records, so you put “=NONE” in that rule cell.



You can see how Mercator mapped the records to the appropriate output files.

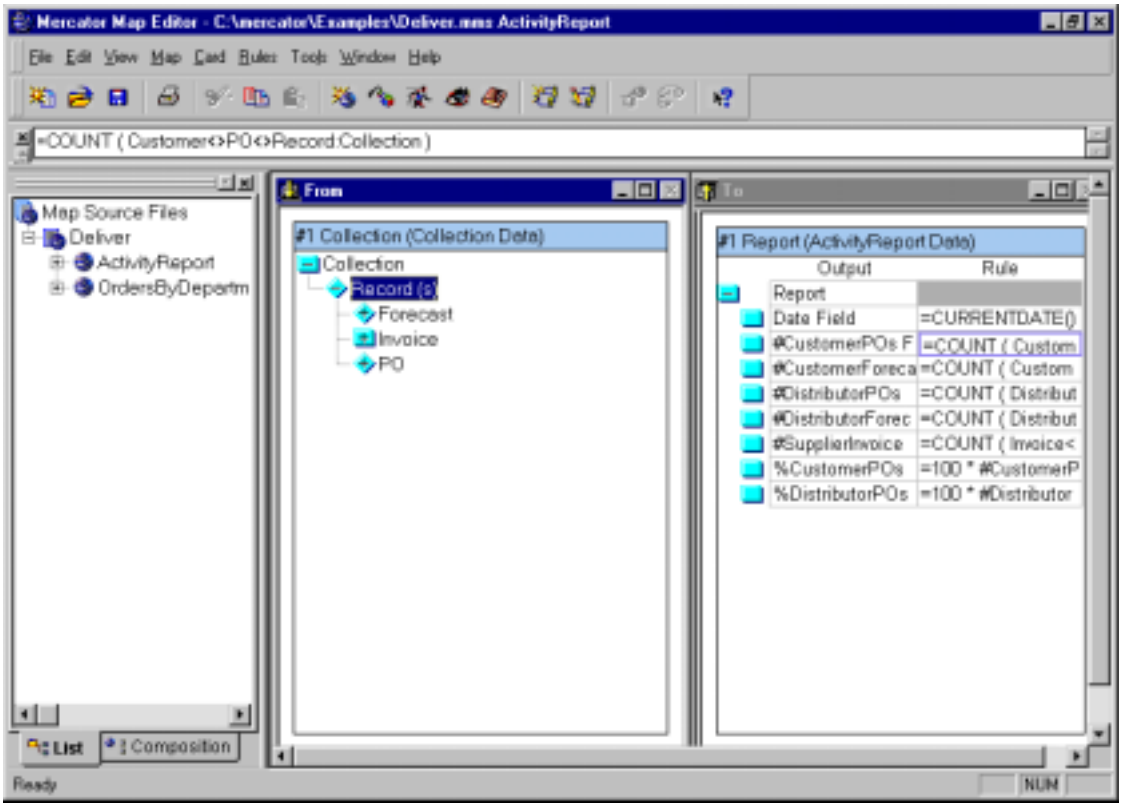


ActivityReport

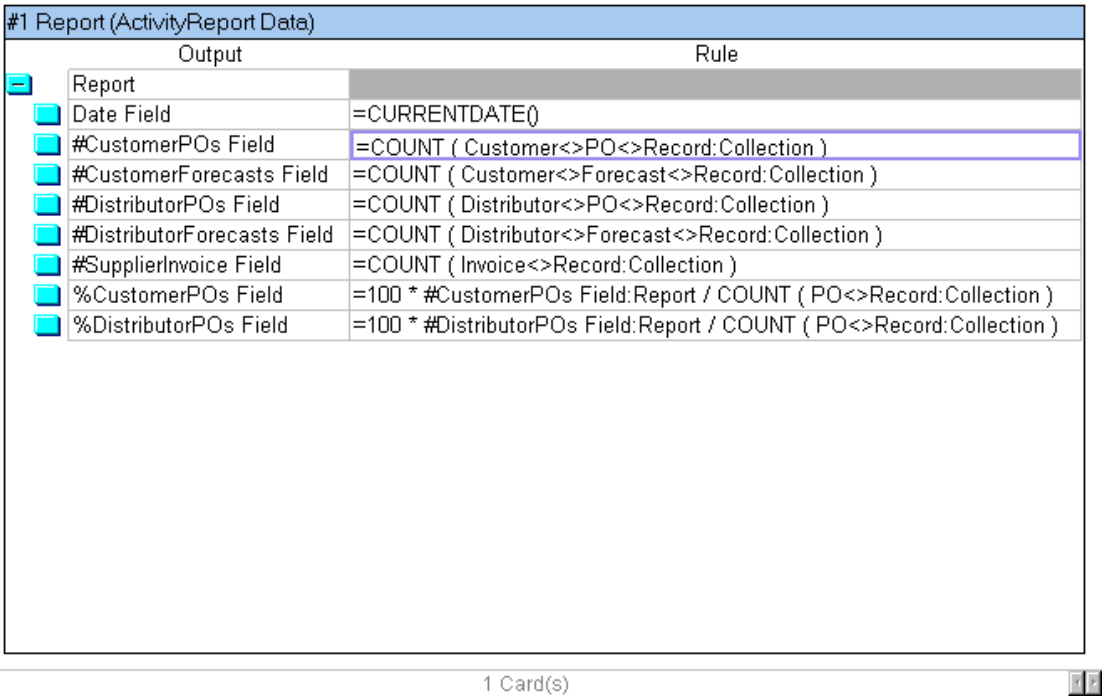
In the executable map ActivityReport, the COUNT function is used to generate some of the output fields. In the last two output fields, the map rules calculate the percentages of Customer and Distributor records in the entire input file.

The COUNT function counts the number of objects in a series. The syntax of the COUNT function is:

```
COUNT (Series of objects you want to count)
```


Mercator Map Editor - C:\mercator\Examples\Deliver.mms ActivityReport
File Edit View Map Card Rules Tools Window Help
=COUNT (Customer<>PO<>Record:Collection)
Map Source Files
Deliver
ActivityReport
OrdersByDepartm
List
Composition
From
#1 Collection (Collection Data)
Collection
Record (s)
Forecast
Invoice
PO
To
#1 Report (ActivityReport Data)
Output
Rule
Report
Date Field
=CURRENTDATE()
#CustomerPOs F
=COUNT (Custom
#CustomerForeca
=COUNT (Custom
#DistributorPOs
=COUNT (Distribut
#DistributorForec
=COUNT (Distribut
#SupplierInvoice
=COUNT (Invoice<
%CustomerPOs
=100 * #CustomerP
%DistributorPOs
=100 * #Distributor
Ready
NUM

Here is a view of the output window maximized. You can see the different kinds of map rules used.

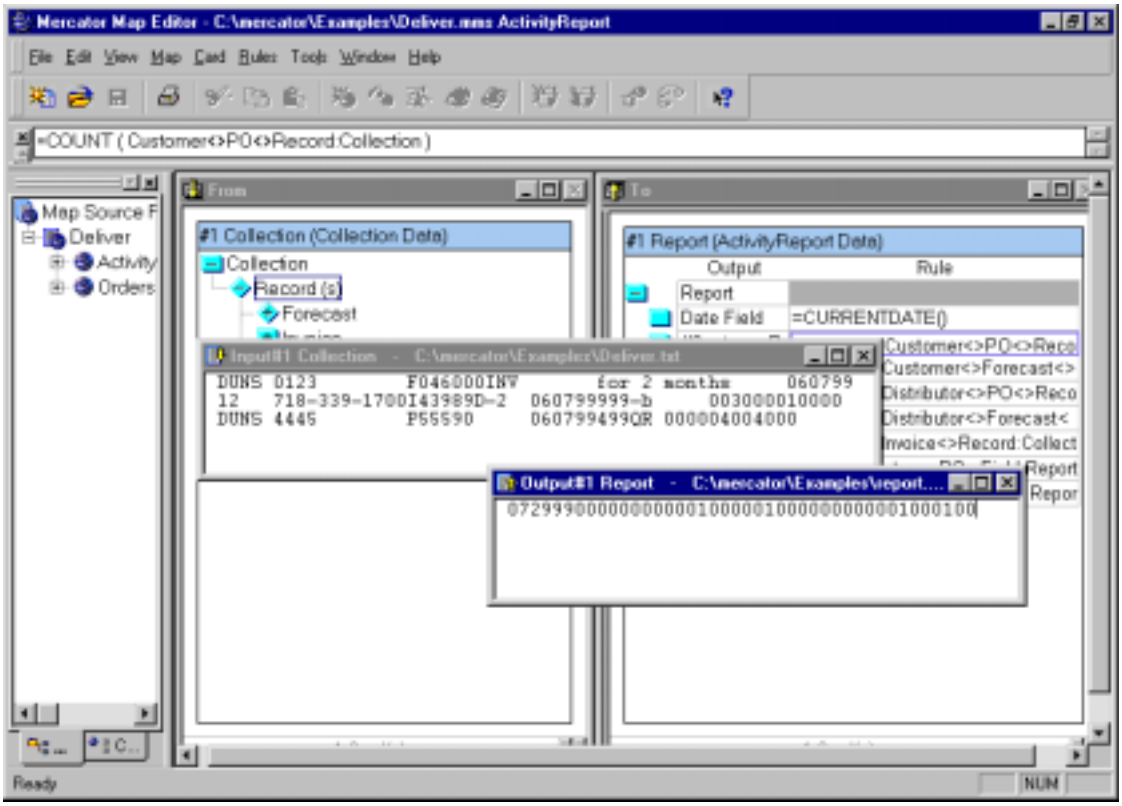


#1 Report (ActivityReport Data)

Output	Rule
Report	
Date Field	=CURRENTDATE()
#CustomerPOs Field	=COUNT (Customer<>PO<>Record:Collection)
#CustomerForecasts Field	=COUNT (Customer<>Forecast<>Record:Collection)
#DistributorPOs Field	=COUNT (Distributor<>PO<>Record:Collection)
#DistributorForecasts Field	=COUNT (Distributor<>Forecast<>Record:Collection)
#SupplierInvoice Field	=COUNT (Invoice<>Record:Collection)
%CustomerPOs Field	=100 * #CustomerPOs Field:Report / COUNT (PO<>Record:Collection)
%DistributorPOs Field	=100 * #DistributorPOs Field:Report / COUNT (PO<>Record:Collection)

1 Card(s)

The resulting output file shows the count of each record type.



Note The value of the first field in your output may be different each time, because it is the current date.

Chapter 9 - Mapping Optional Inputs

This example maps data that has optional data objects.

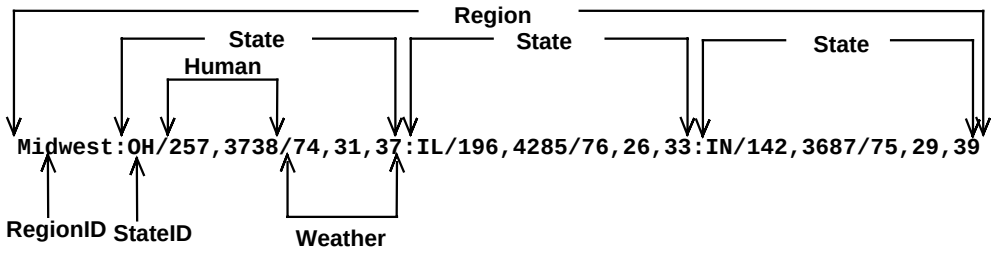
What You Want to Do

You have a data file of statistics on states in the U.S. The statistics for each state may include population density, median household income, average summer temperature, average winter temperature, and average yearly rainfall. Suppose you want to create a new data file that contains only the population statistics.

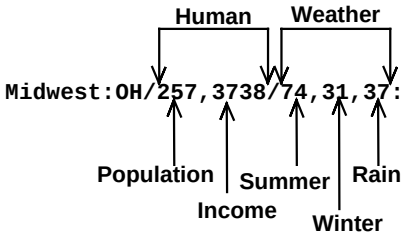
Here is a portion of the input data:

```
Midwest:OH/257,3738/74,31,37:IL/,4285/76,26,33:IN/142,3687/75,29,39
Mountain:MT/5,3130/68,19,11:ID/8,2953/75,29,11:WY/3,3353/70,26,15
Plains:MN/48,3635/73,12,25:IA/,3549/74,20,33:MO/67,3458/78,32,35
```

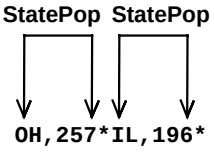
The diagram below shows how the data objects within a Region are organized. A Region is made up of a RegionID, and a series of States. Each State is made up of a StateID, Human, and Weather statistics.



Within each data object, Human, are the statistics for population density and household income. Within each data object, Weather, are the statistics for average summer temperature, average winter temperature, and average yearly rainfall.



You want to extract each StateID and corresponding Population. The output file consists of a series StatePops, containing a StateID and its Population.



How to Do It

Define the input and output data in a type tree.

In the Map Editor, create an executable map. To generate each StatePop in the output, use a functional map.

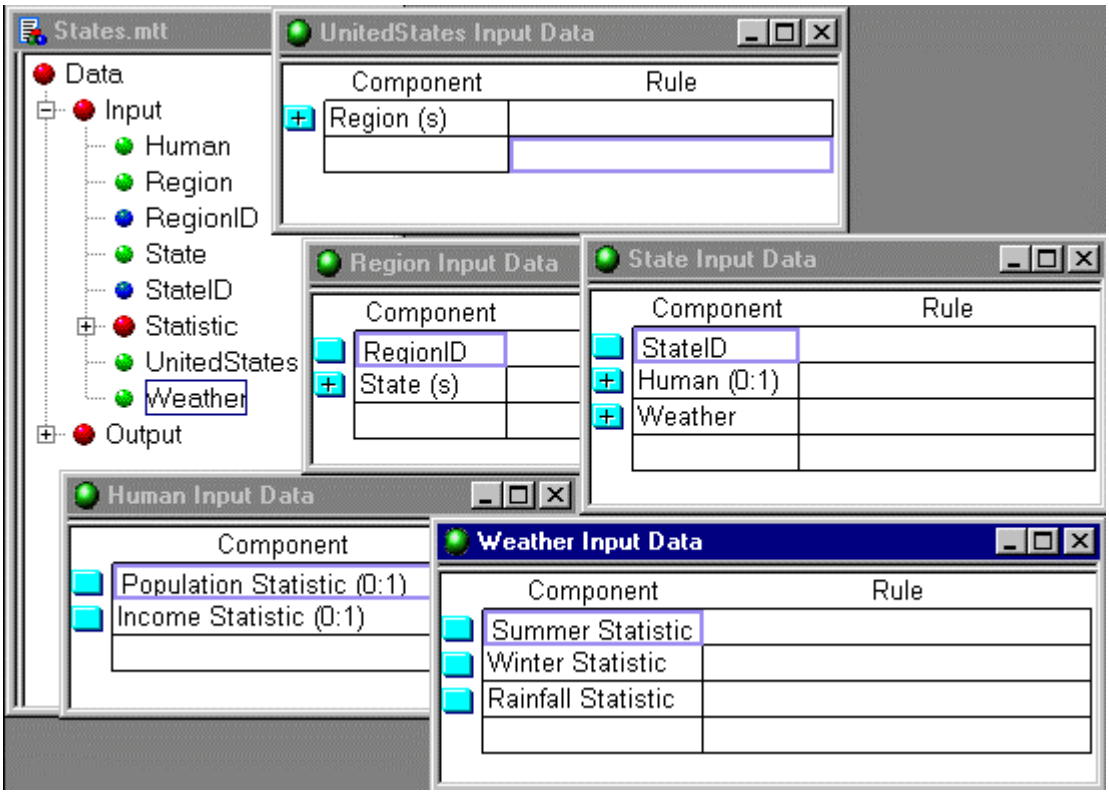
Files Used in this Example

For this example, use these files, which are in your mercator\examples directory.

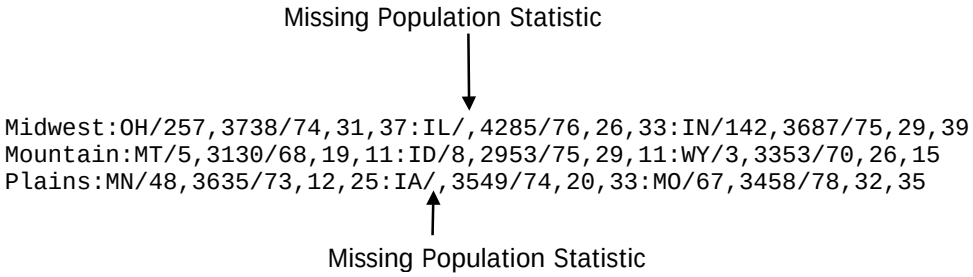
File	Use
sts.txt	Use this file as the input data file.
states.mtt	This type tree defines the data files.
states.mms	This map source file contains the map explained in this example.
output.txt	This output file is created by running the map.

Using the Type Editor

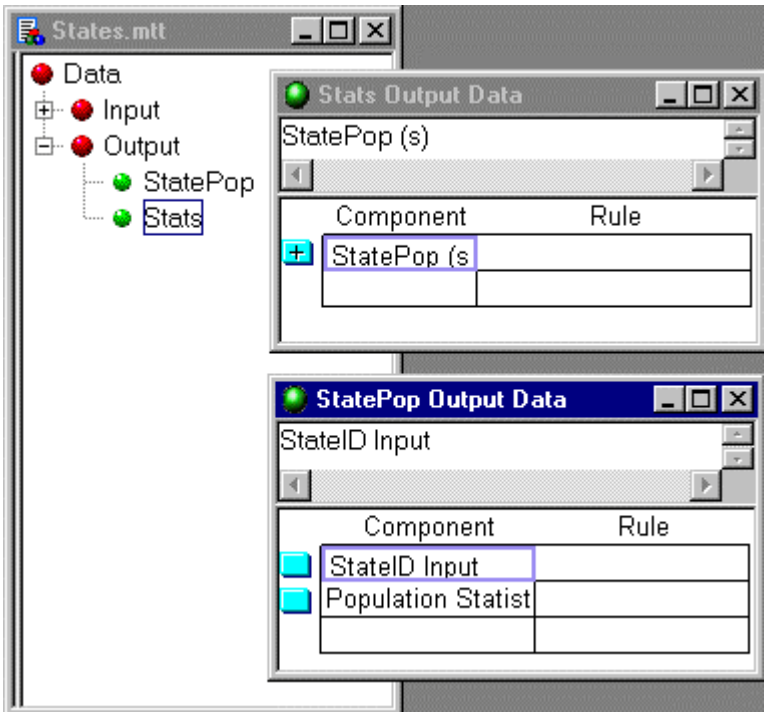
The input types are defined like this:



Notice that the Population Statistic is optional—it has a component range of (0:1). In the data, some Population Statistics are missing.



The output types are defined like this:

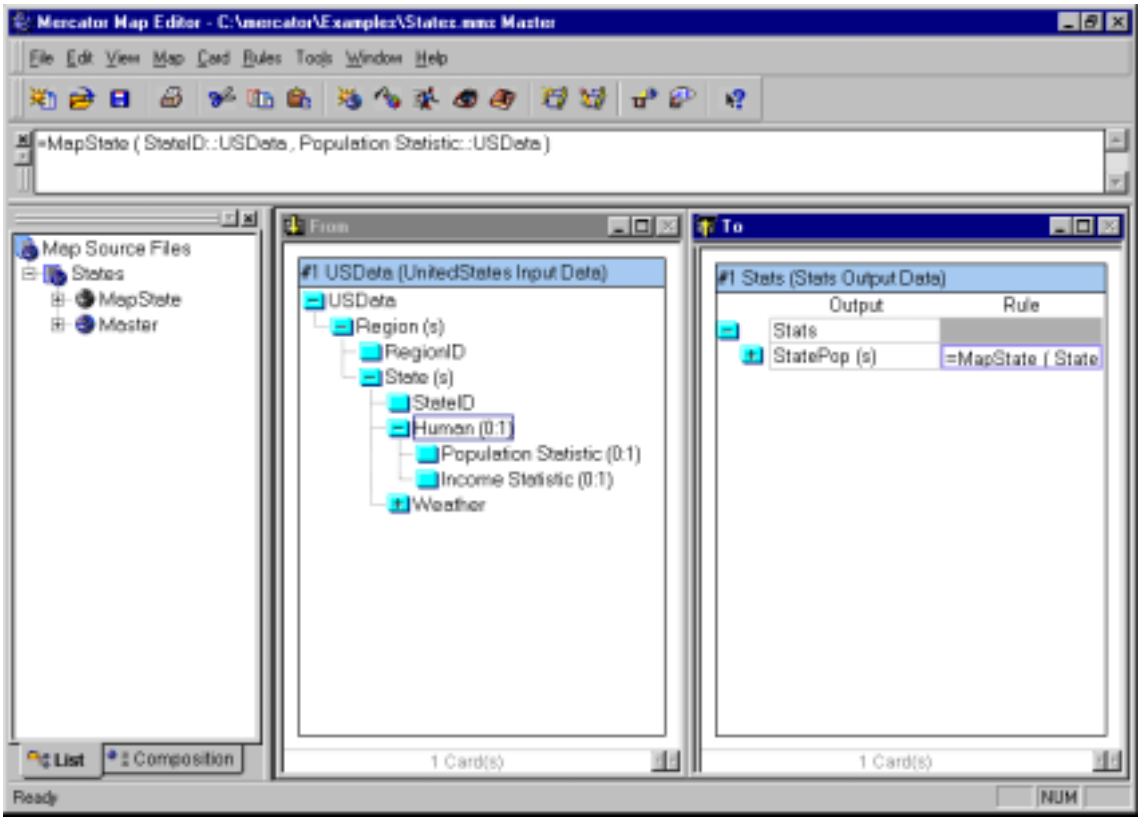


Using the Map Editor

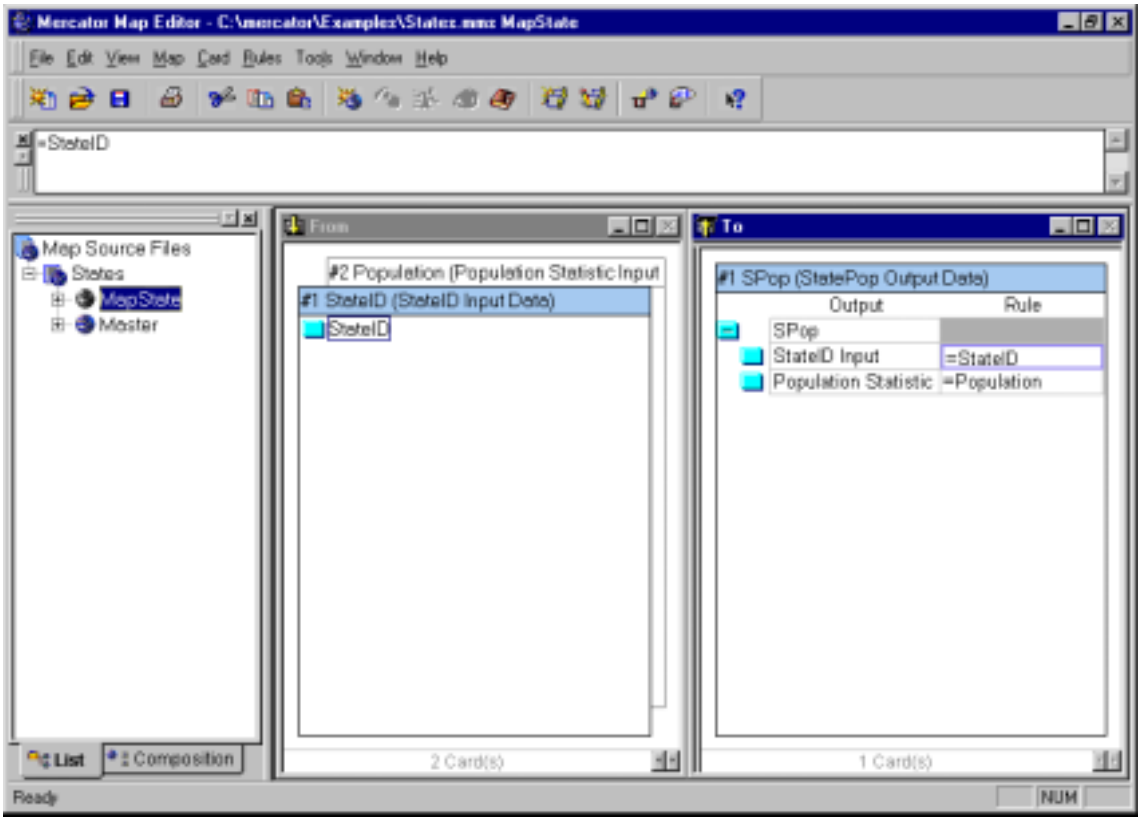
The rule on the output StatePop references a functional map, MapState.

You want to generate a StatePop *only* if Population exists in the input. Therefore, Population Statistic is one of the arguments of the functional map. This ensures that if Population Statistic is missing, the functional map is not called, and the StatePop is not created for that state.

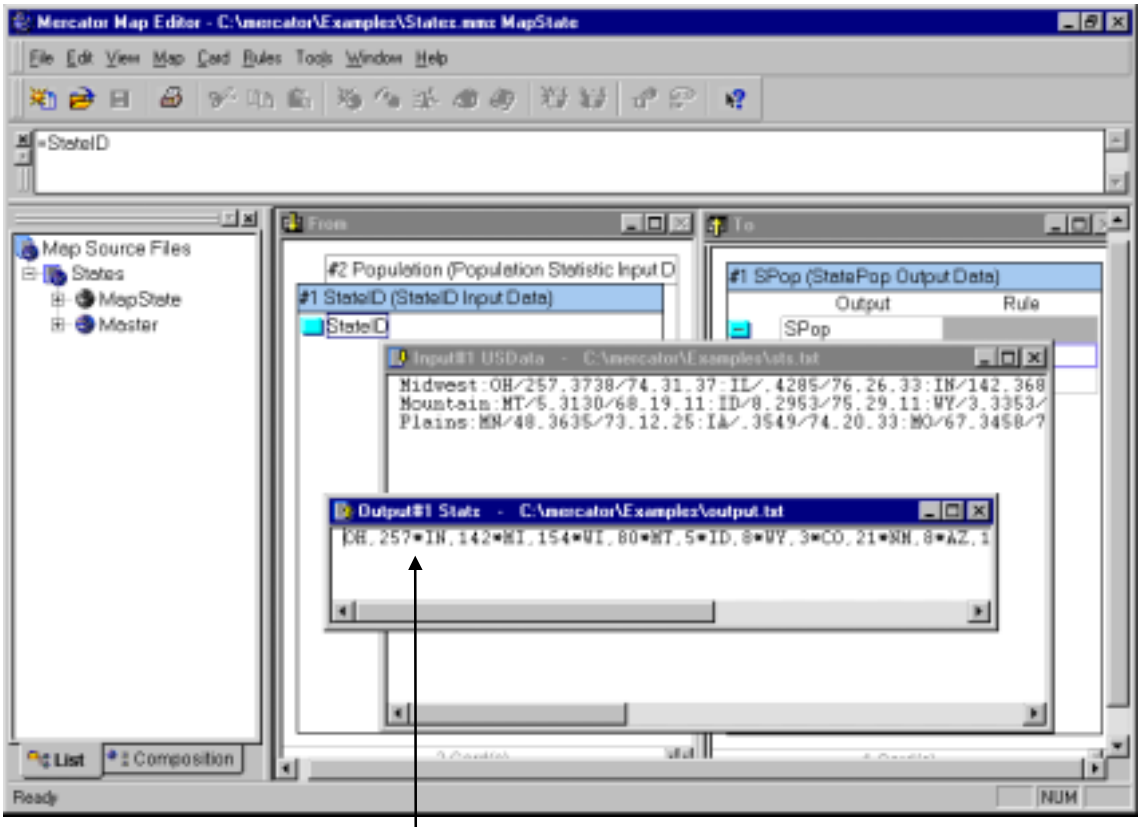
The other argument of the functional map is StateID.



In the map MapState, the StateID and Population Statistic are simply dragged over to the outputs.



The output file shows that a StatePop was not created for the states having no Population Statistic. For example, IL had no Population Statistic, and no IL data was created in the output.



There is no output data for IL

Chapter 10 - Mapping Multiple Files to One File

This chapter includes three examples. Each example maps two input files—one header and one detail file—into one output file of purchase orders.

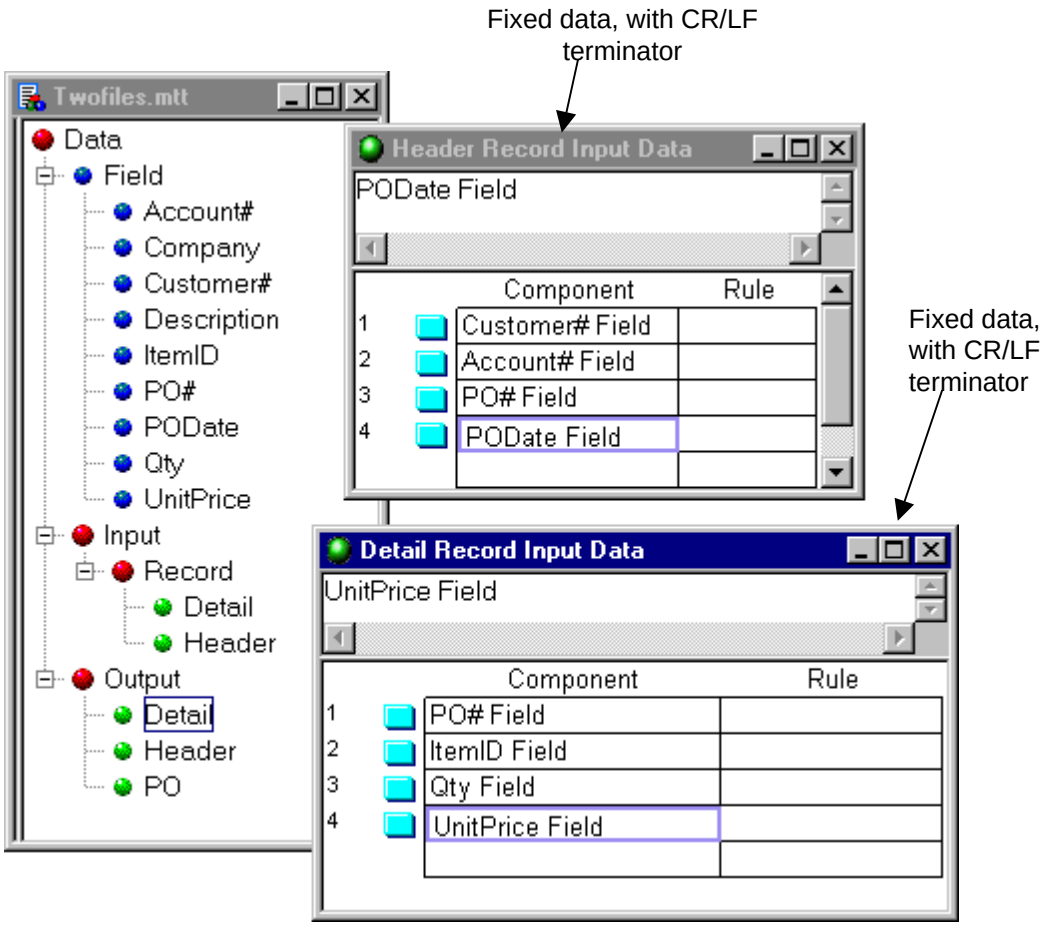
What You Want to Do

You have a header file made up of header records. Each header record includes a customer number, an account number, a PO#, and a date. You have a detail file made up of detail records. A detail record includes a PO#, an item ID, a quantity, and a unit price.

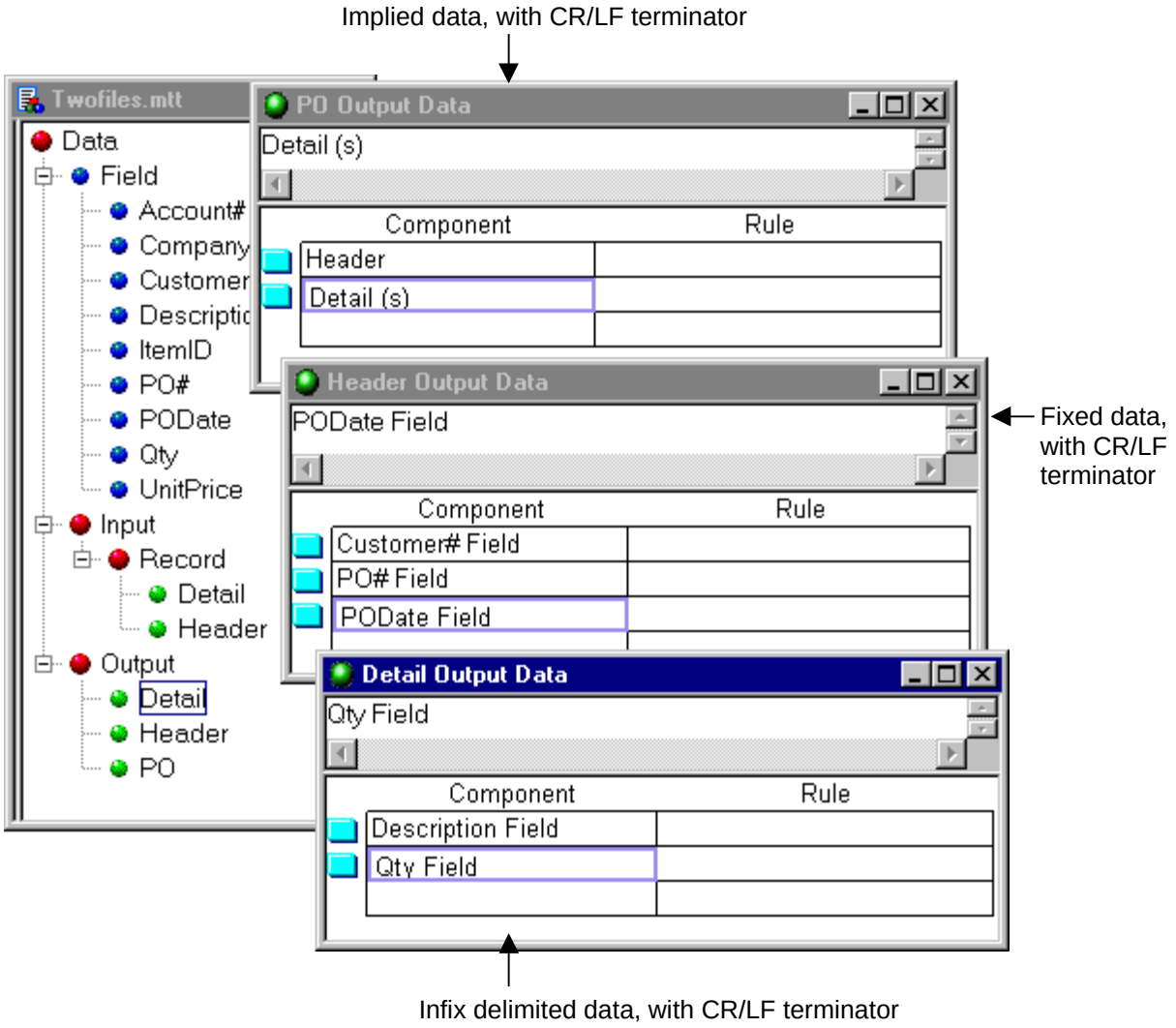
You want to create an output file made up of purchase orders. Each PO has a header and a set of details.

How to Do It

In the Type Editor, define the input header record and detail record.



Define the output PO, Header and Detail.



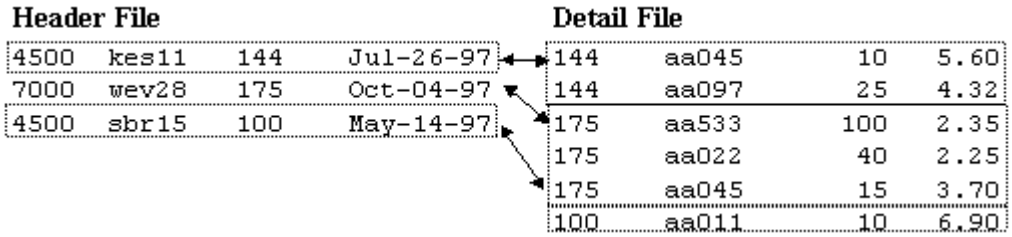
Next, look at how your data is organized to decide how to define the header file, detail file, and PO file, and what map rules to use.

In each of the following examples, the files are defined differently. For each example, create an executable map, using the header and detail files as inputs, and the PO file as the output.

Case 1 – Header and Detail Files in the Same Order

Suppose the detail file has been sorted to correspond with the data in the header file. That is, the first header record goes with the first set of detail records, the second header record goes with the second set of detail records, and so on.

The input data looks like this. The dotted lines indicate how each Header Record corresponds to a particular set of Detail Records:



You want the output data to look like this:

```
4500       PO#144 Jul-26-97
aa045,10
aa097,25

7000       PO#175 Oct-04-97
aa533,100
aa022,40
aa045,15

4500       PO#100 May-14-97
aa011,10
```

Files Used in Case 1

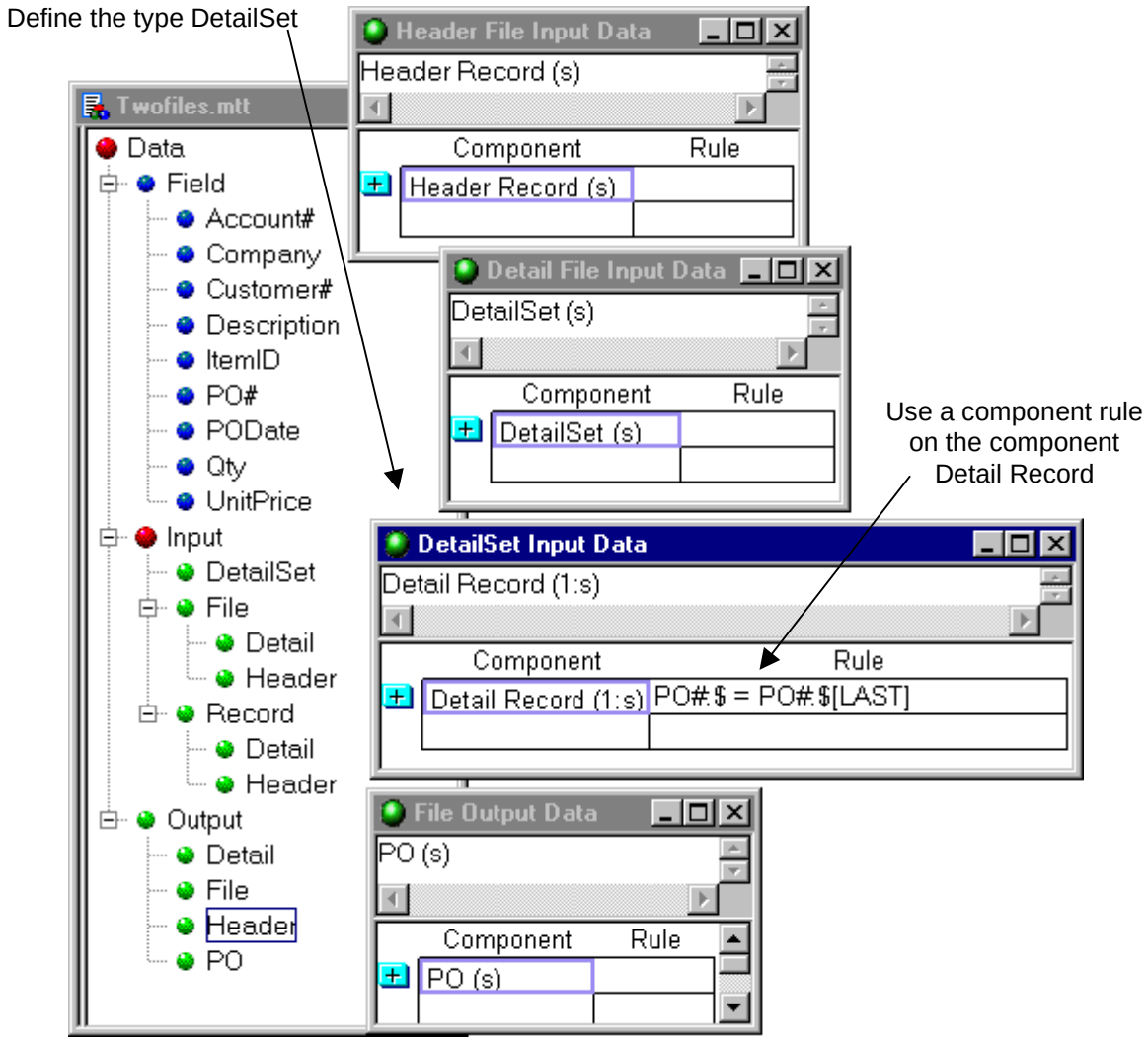
The following table lists the files used in Case 1.

File	Use
header.txt	You create this file to use as an input data file.
detail.txt	You create this file to use as an input data file.
twofiles.mtt	You create this type tree to define the data files.
twofiles.mms	You create this map source file.
output.txt	This output file is created when you run the map.

Using the Type Editor

Define the three files in a type tree. To define the detail file, define the type DetailSet, which is made up of Record(s). Use control-break logic in a component rule to define how the sets are organized—when the PO# changes, a new set begins.

For more information on control-break logic, see Chapter 7.



Using the Map Editor

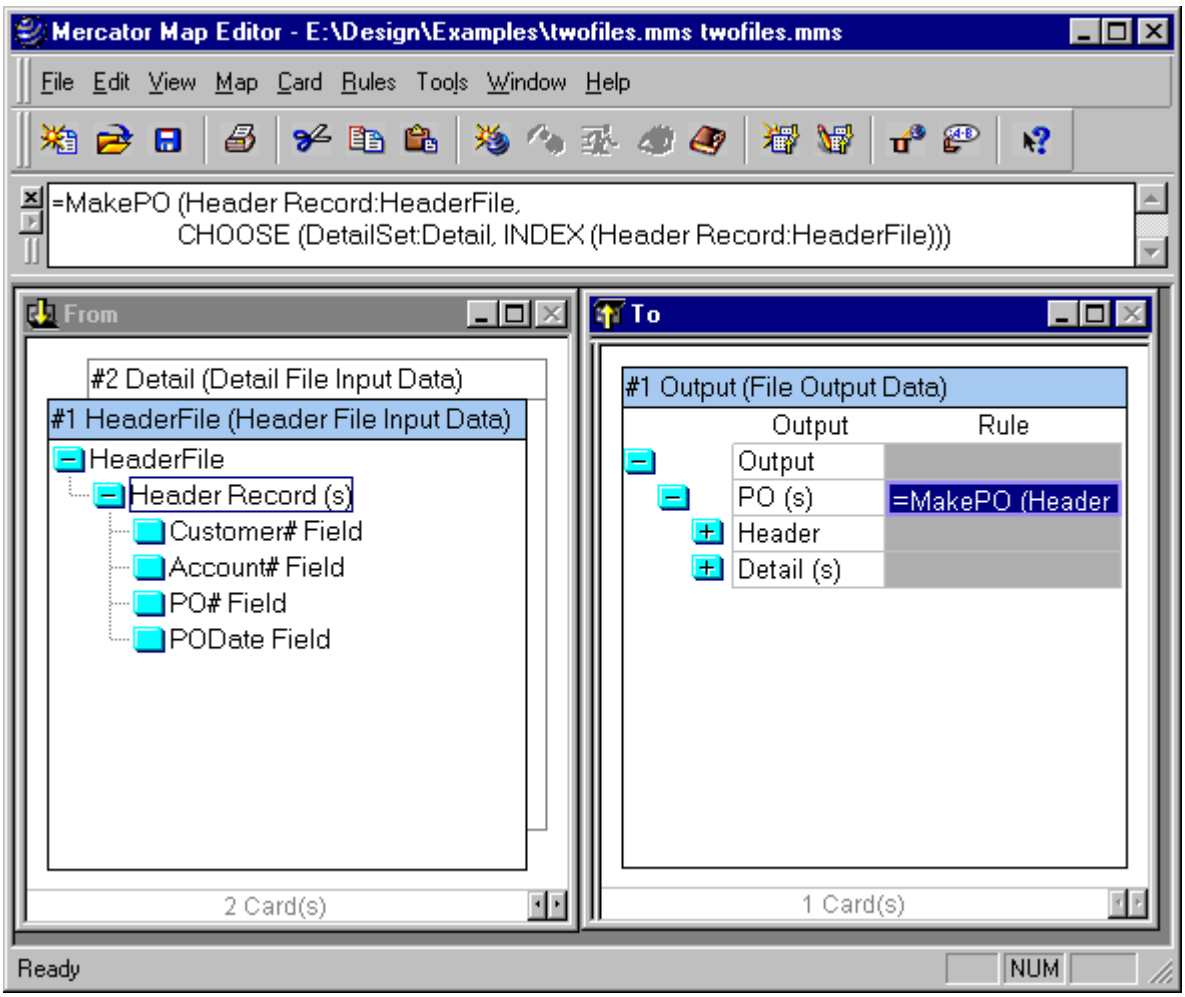
In the executable map, there are two inputs—the header file and the detail file. The output is the PO file.

Use a functional map to generate each PO in the output. The arguments to this map, “MakePO,” are a Header Record, and the corresponding DetailSet. To get the corresponding DetailSet, use the CHOOSE function. The CHOOSE function picks an object at a given index in a series. You want to pick the DetailSet whose index matches Header Record’s index. For example, when Header Record #2 is used, the CHOOSE function retrieves DetailSet #2 as well.

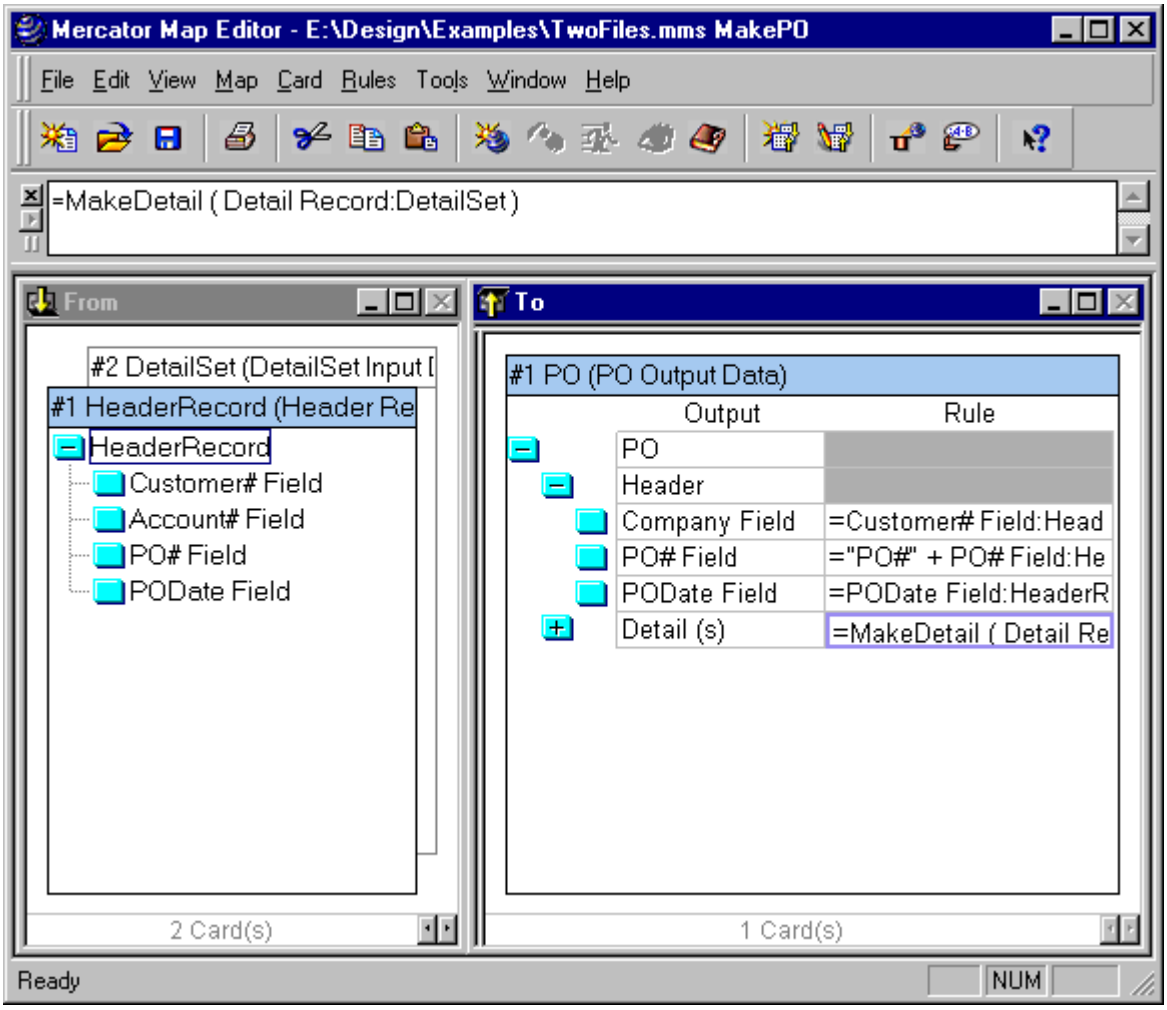
The rule on the PO output is:

```
=MakePO (Header Record:HeaderFile,
     CHOOSE (DetailSet:DetailFile, INDEX (Header Record:HeaderFile)))
```

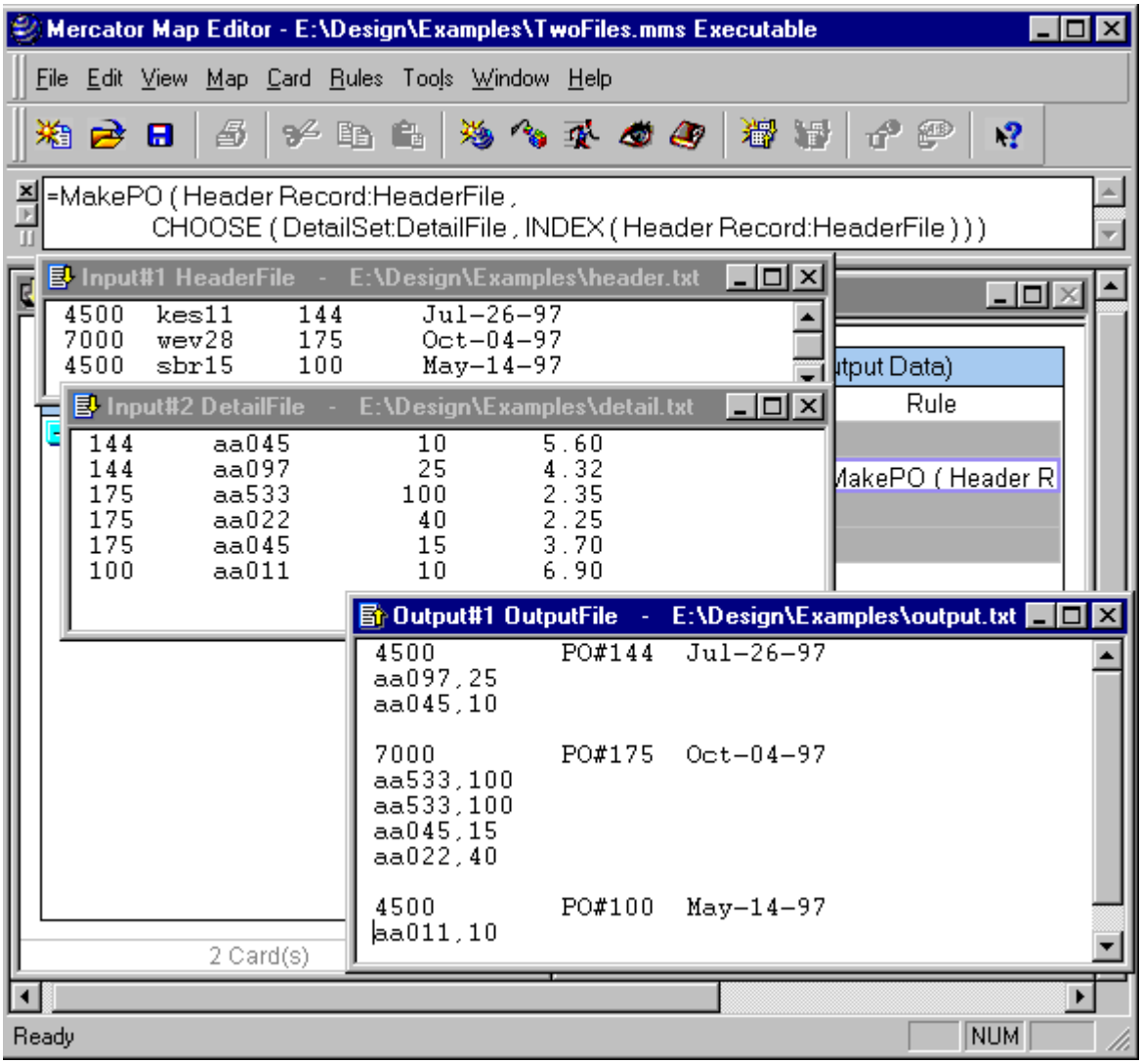
The executable map looks like this:



In the functional map **MakePO**, the Header information in the output is mapped from the Header in the input. The Details are mapped from the Detail Records, by using another functional map—**MakeDetail**.



The final result shows that Mercator matched the header with its corresponding detail set.



Case 2 – The Detail File is Not Sorted by PO

Suppose that the detail file is not organized into sets of detail records. That is, the detail records are in a random order. The detail records for PO# 144, for example, are scattered throughout the file.

Here is the new detail file:

```
175     aa533       100    2.35
175     aa045        15    3.70
100     aa011        10    6.90
144     aa097        25    4.32
175     aa022        40    2.25
144     aa045        10    5.60
```

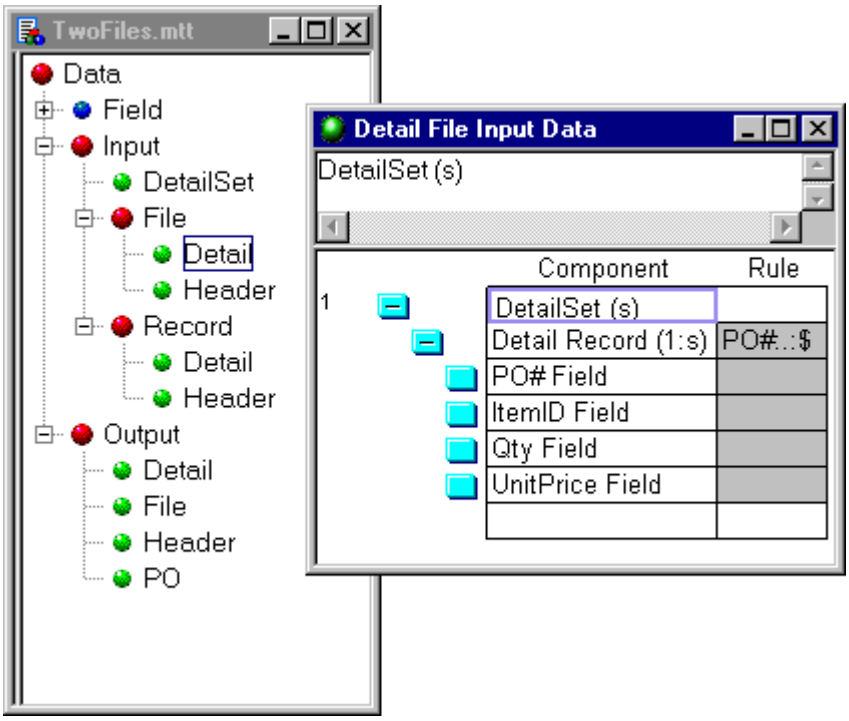
Files Used in Case 2

The following table lists the files used in Case 2.

File	Use
header.txt	Use this file, which was created in Case 1, as an input data file.
detail2.txt	You create this file to use as an input data file.
newdef.mtt	You create this type tree to define the data files.
twofiles.mms	This is a continuation of the map source file you created in Case 1.
output2.txt	This output file is created by running the map.

Using the Type Editor

Define the detail file as made up of detail records.



Using the Map Editor

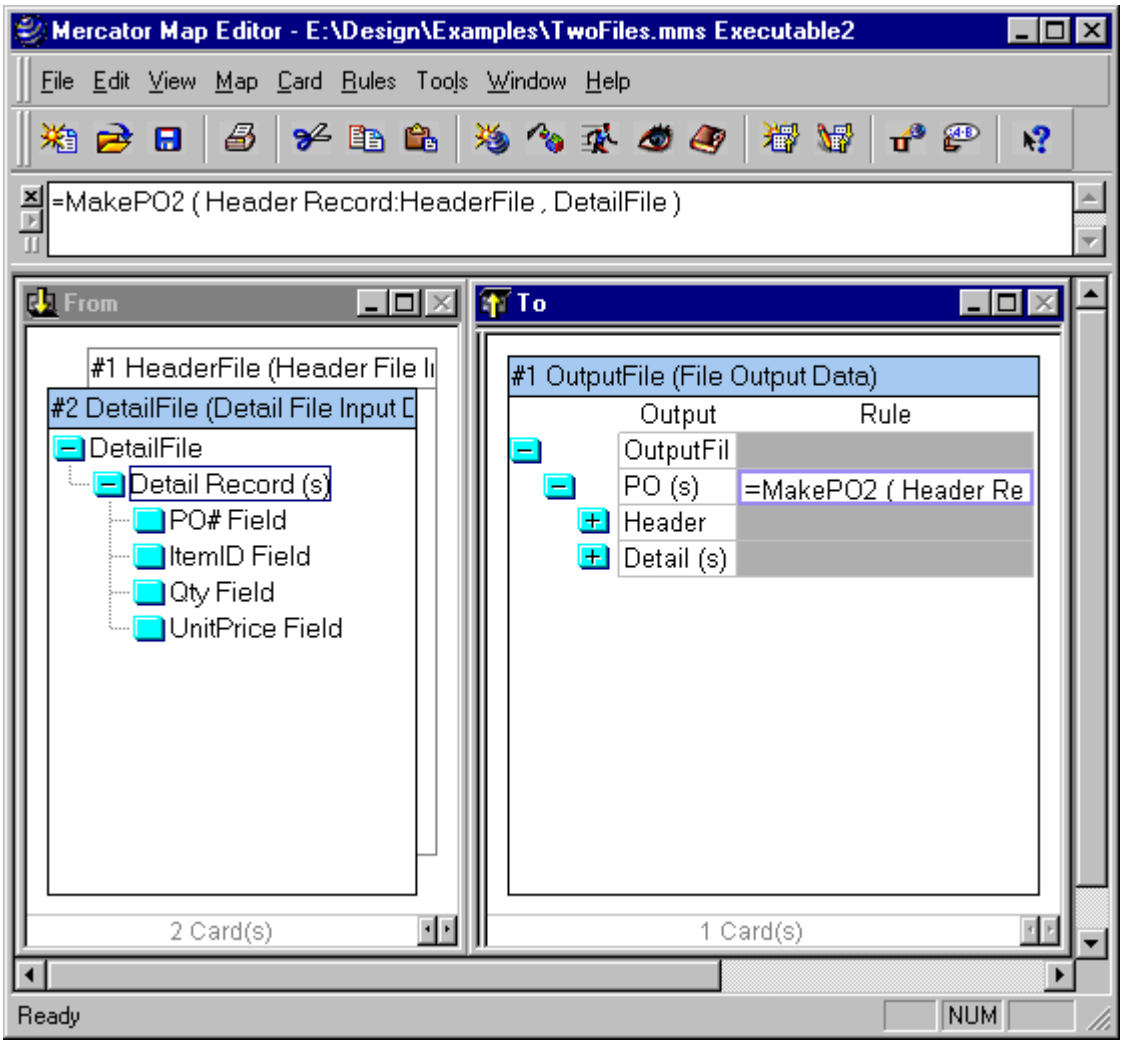
Create an executable map similar to the one in Case 1. The inputs are the header file and detail file. The output is the PO file.

In the map rule for the PO output, a functional map “MakePO2” is referenced. The two arguments to this map are a Header Record, and the entire Detail File.

The map rule for the PO output is:

```
=MakePO2 (Header Record:HeaderFile, DetailFile)
```

The executable map looks like this:

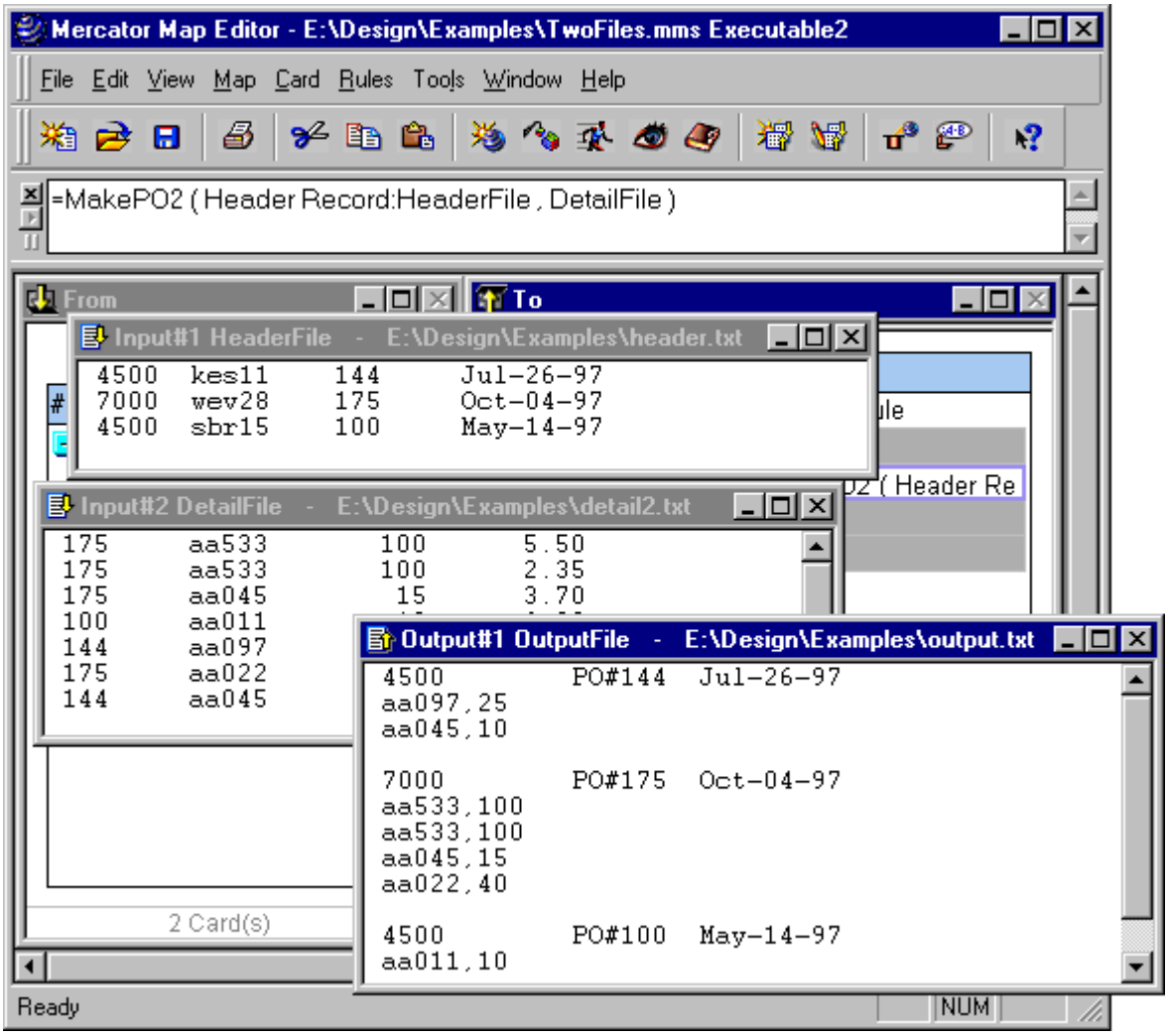


In the functional map **MakePO2**, the rule on **Detail** references another functional map, **MakeDetail2**. The argument to this map is the extract of the **Detail Records** that have the same PO# as the **Header Record**.

The map rule for **Detail** is:

```
=MakeDetail2 (EXTRACT (Detail Record:Detail,
PO# Field:.:Detail = PO# Field:Header))
```

The final results show that, even though the detail records in the input file are not ordered, Mercator still matched the header with the appropriate details.



Case 3 – Organize the POs by Customer

Suppose you want the POs in the output to be sorted by customer. The customer number is the first field in the header of the PO. You define a POSet, made up of POs for the same customer. To generate a POSet per customer, you use the UNIQUE function.

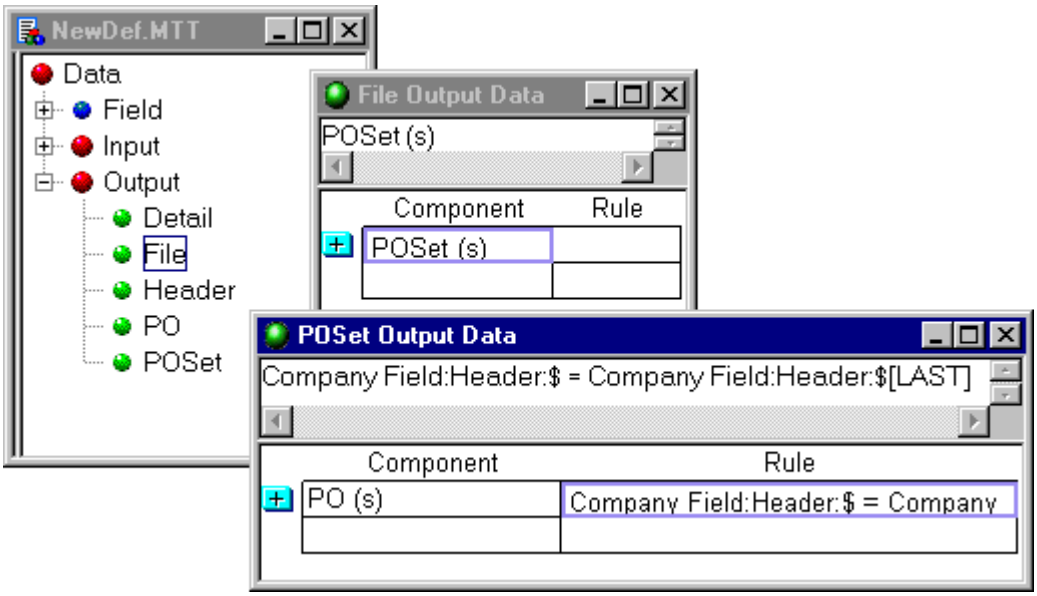
Files Used in Case 3

The following table lists the files used in Case 3.

File	Use
header.txt	Use this file, which was created in Case 1, as an input data file.
detail2.txt	Use this file, which was created in Case 2, as an input data file.
newdef.mtt	You modify this type tree, which was created in Case 2.
twofiles.mms	This is a continuation of the map source file created in Case 1 and modified in Case 2.
output3.txt	This output file is created by running the map.

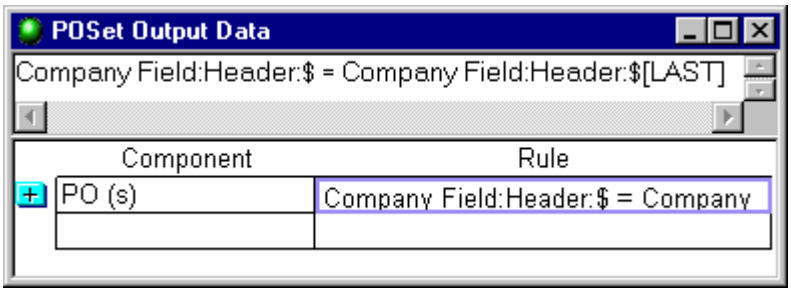
Using the Type Editor

The definitions of the input files are the same as those in Case 2. Change the definition of the output file. It is made up of **POSets**.



A **POSet** consists of a series of POs that have the same **Customer# Field**. Use control break logic in the component rule for PO.

The rule for the PO component of POSet is:



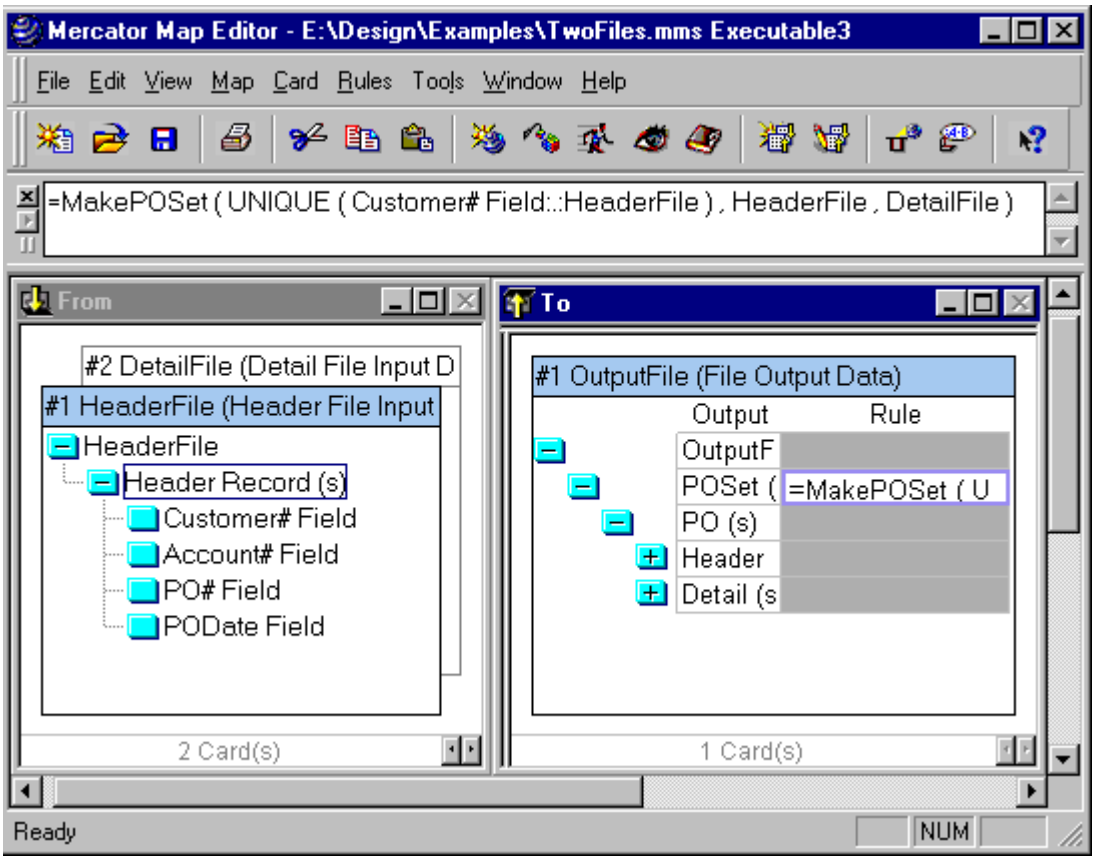
Using the Map Editor

In the executable map, the rule on POSet refers to the functional map MakePOSet. The first argument is the unique Customer# Fields. This generates one POSet per customer. The second and third arguments are the entire header file, and the entire detail file.

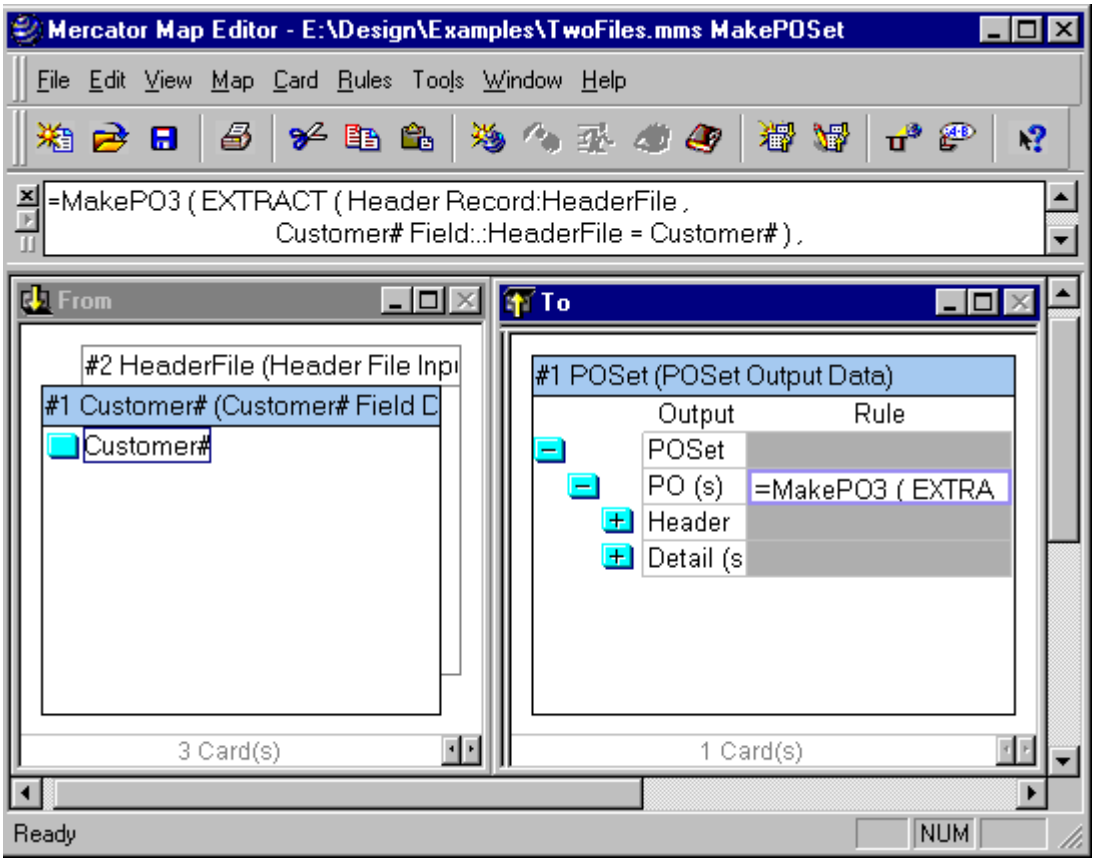
The map rule for POSet is:

```
= MakePOSet (UNIQUE (Customer# Field:.:HeaderFile), HeaderFile,
       DetailFile )
```

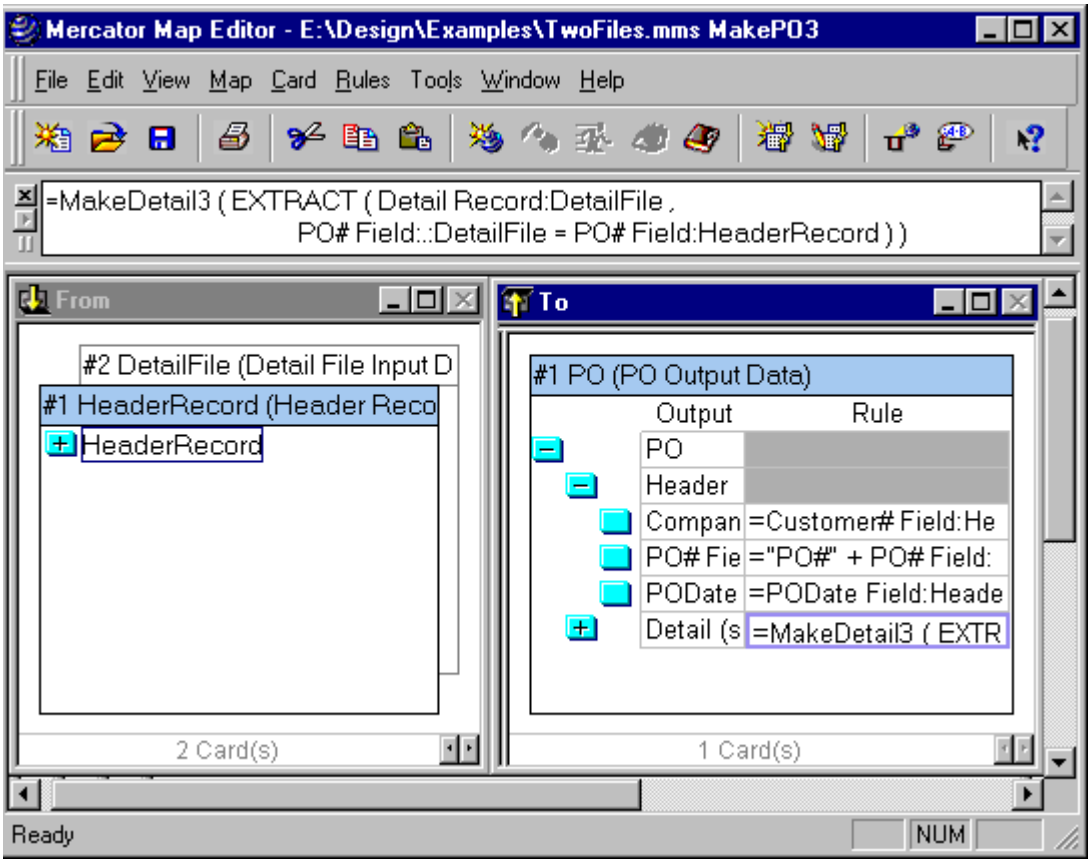
The executable map looks like this:



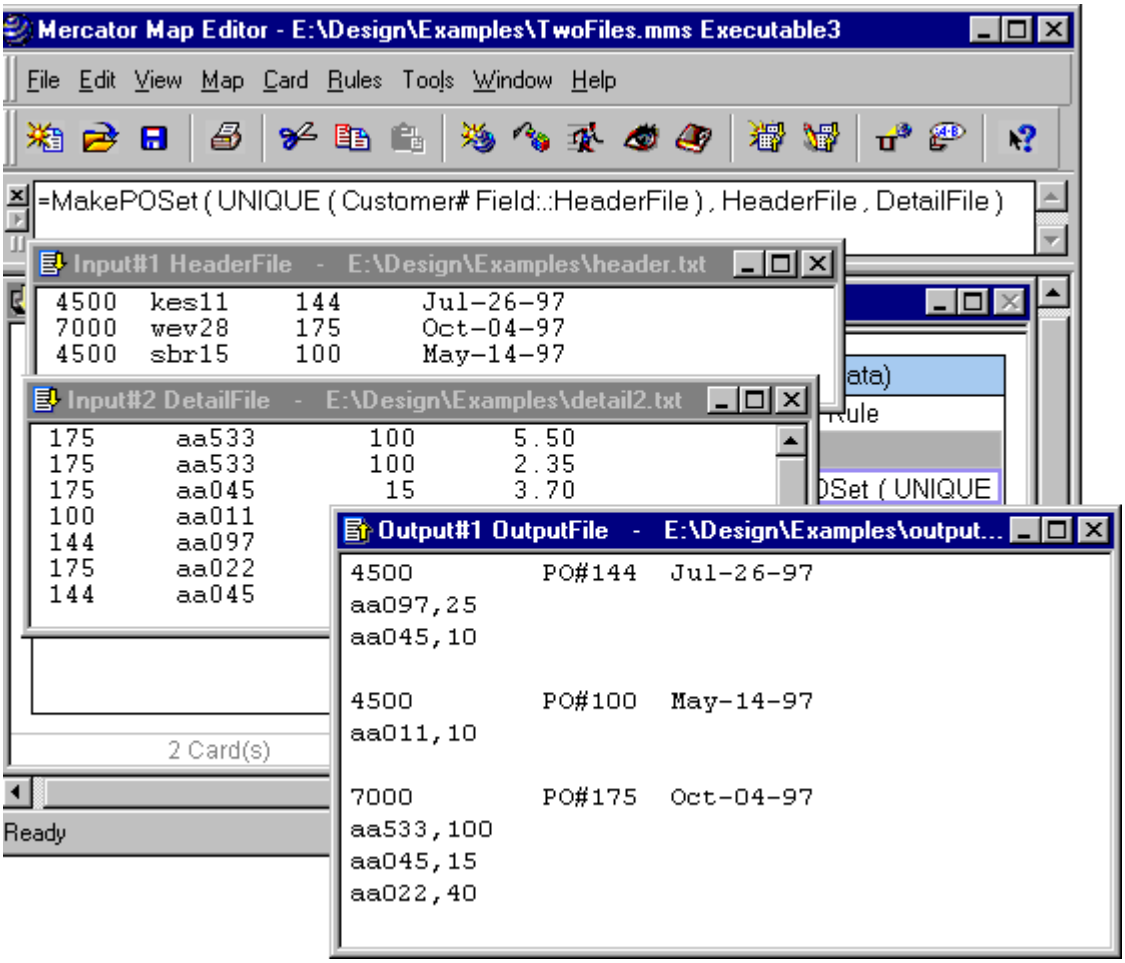
In the **MakePOSet** map, the rule on PO references another functional map, **MakePO3**. The first argument of **MakePO3** extracts the Header Record that has the same **Customer#** as the **Customer#** in input card 1. The second argument is the entire **Detail File**.



In MakePO3, the rule on Detail references the map MakeDetail3. The argument to this map is the Detail whose PO# matches that of Header Record.



In the resulting output file, the first POSet contains POs for customer #4500. This is followed by a POSet for #7000.



Chapter 11 - Mapping Multiple Files to Multiple Files

This chapter includes an example of mapping a file of purchase orders and a cross-reference file, to a header file and a detail file. This map shows how you can add data to the output when it is missing from the main input.

What You Want to Do

You have a file of purchase orders. You want to split the PO file into a Header file and a Detail file. You also have a Cross-reference file of unit prices.

How to Do It

The **Detail** file has a **Unit Price** field, which is not in the input **PO** file. This **Unit Price** field is in a **Unit Price CrossReference** file—an additional input file. Define this file in a type tree.

In the Map Editor, create a map that has the **PO** file and the **CrossReference** file as inputs, and **Header** and **Detail** files as outputs.

Files Used in this Example

The following table lists the files used in this example.

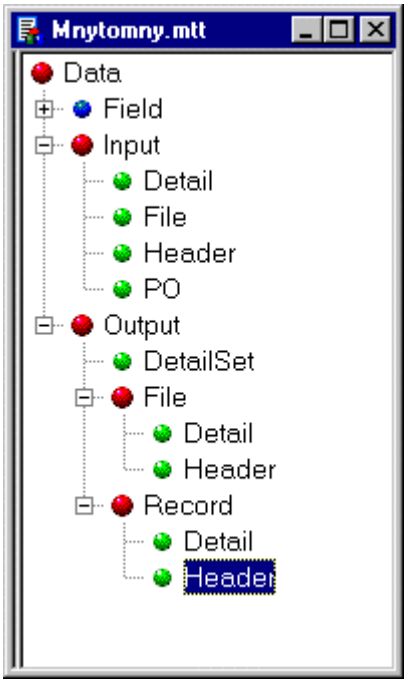
File	Use
pofile.txt	Use this file as an input data file. It is a rename of the output3.txt file created in Chapter 10, Case 3.
untprice.txt	You create this file to use as the cross-reference input data.
untprice.mtt	You create this type tree to define the cross-reference data.
twofiles.mtt	This type tree was created in the examples of Chapter 10.
twofiles.mms	This is a continuation of the map source file you created in Chapter 10.
hdrout.txt	This output file is created by running the map.
dtlout.txt	This output file is created by running the map.

Using the Type Editor

Here is the input **PO** file:

```
4500      PO#144 Jul-26-97
aa097,25
aa045,10

7000      PO#175 Oct-04-97
aa533,100
aa045,15
aa022,40

4500      PO#100 May-14-97
aa011,10
```

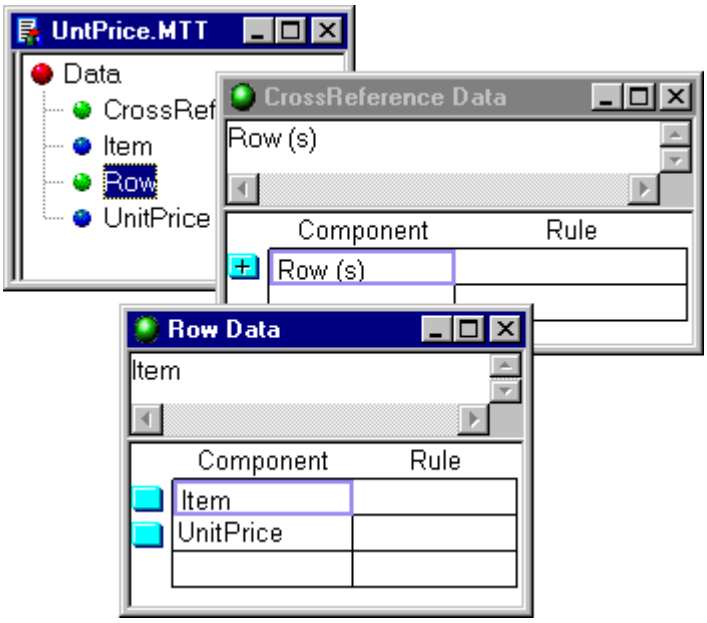
The **PO**, **Header**, and **Detail** files were defined in the type tree **twofiles.mtt**. However, the **PO** file was the output then, and the **Header** and **Detail** were the input. Open **twofiles.mtt** and save it as **mnytomny.mtt**. Then swap the names of the Input and Output categories:



The data for the **CrossReference** file of items and unit prices looks like this:

```
aa045,5.60
aa097,4.32
aa533,2.35
aa022,2.25
aa045,3.70
aa011,6.90
```

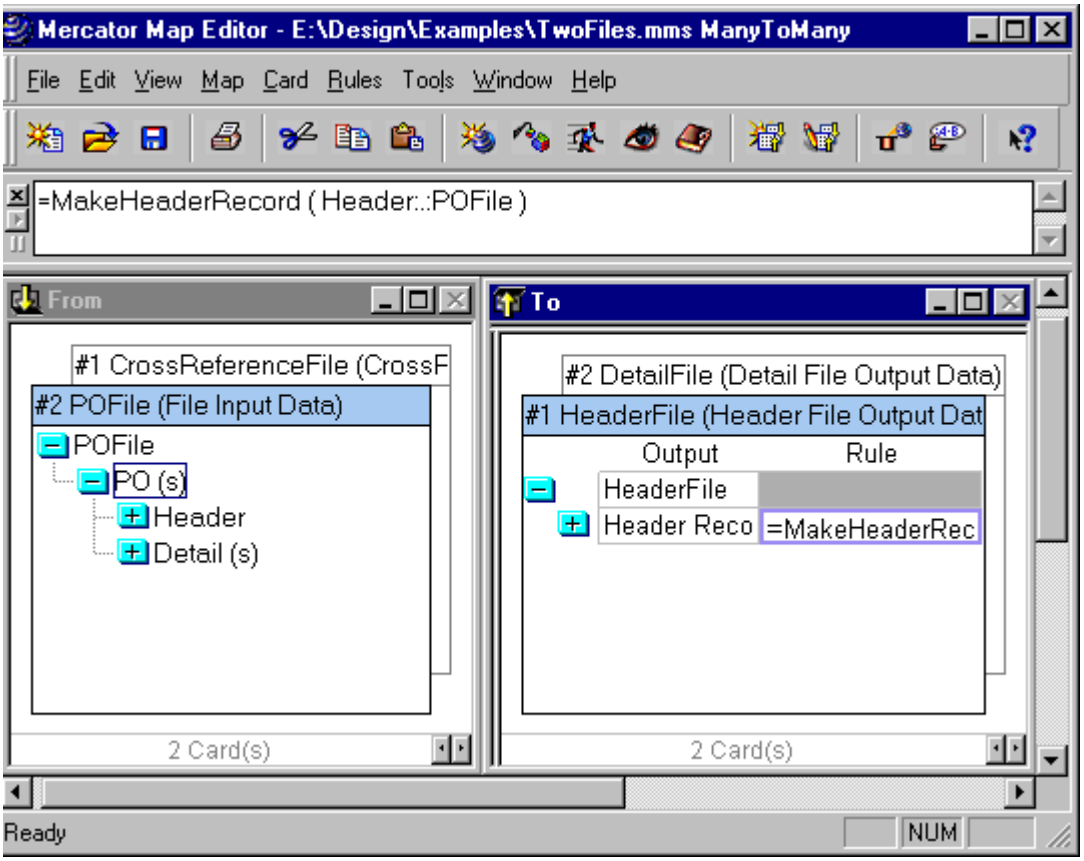
Define the **CrossReference** file in a type tree:



Using the Map Editor

Create an executable map that has two inputs—the **PO** file, and the **Unit Price** file. It has two outputs—the **Header** file and the **Detail** file.

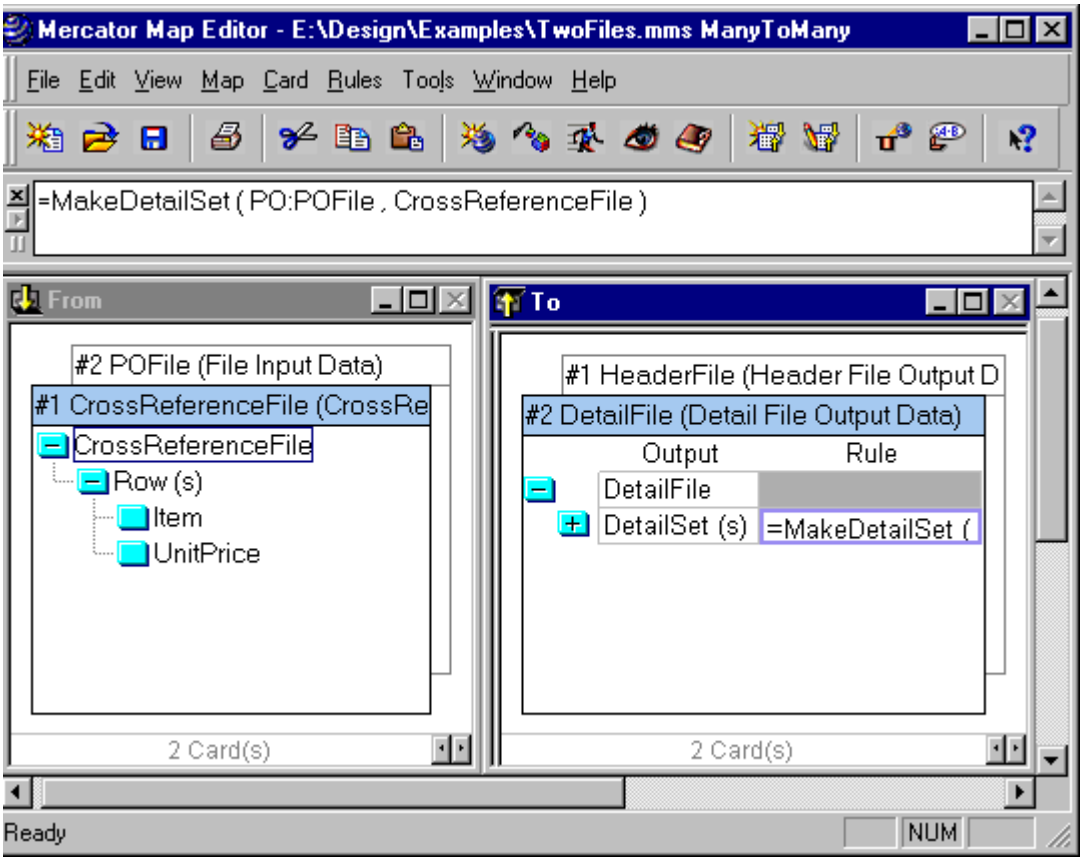
The map rule on **Header Record** references the functional map **MakeHeaderRecord**. Its input is a **Header** from the **PO** file.



The map rule on the output **DetailSet** references the functional map **MakeDetailSet**. You want to generate a **DetailSet** per PO in the input. You also need the unit price information from the **Unit Price** file. Therefore, there are two arguments to **MakeDetailSet**—a PO and the entire **CrossReference** file.

The map rule for **DetailSet** is:

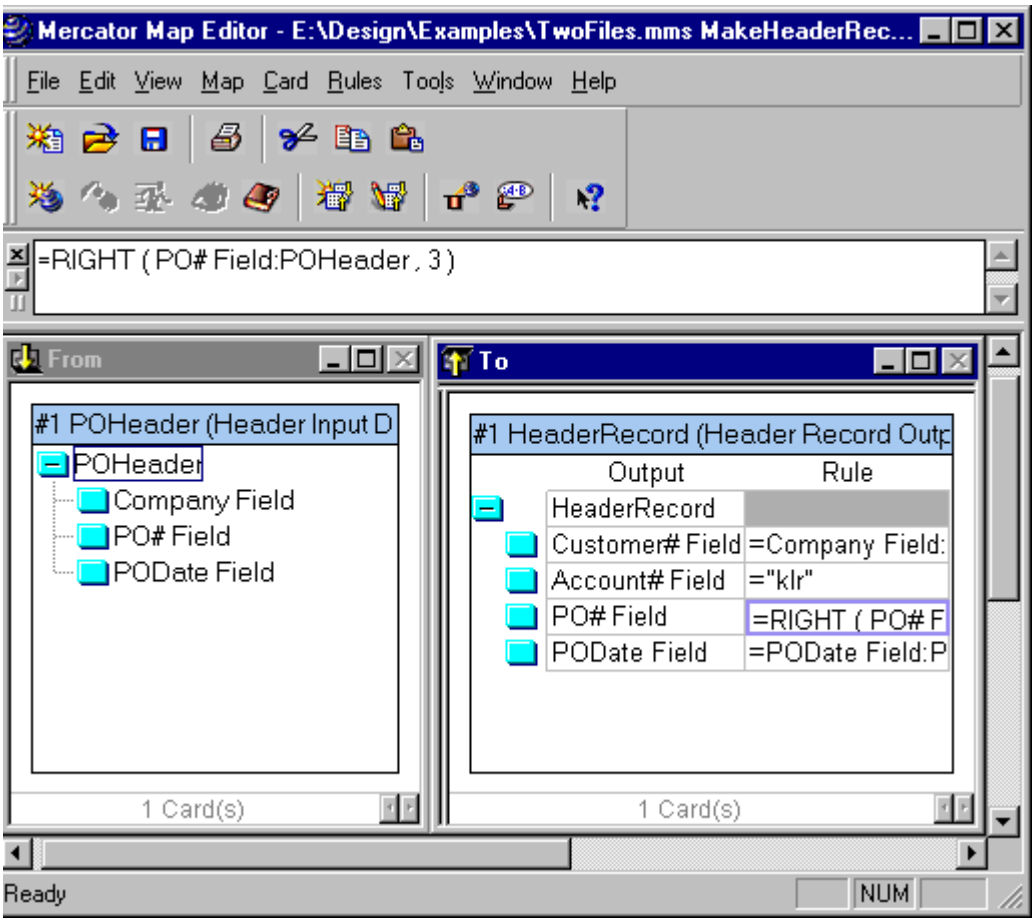
```
= MakeDetailSet (PO:POFile, CrossReferenceFile)
```



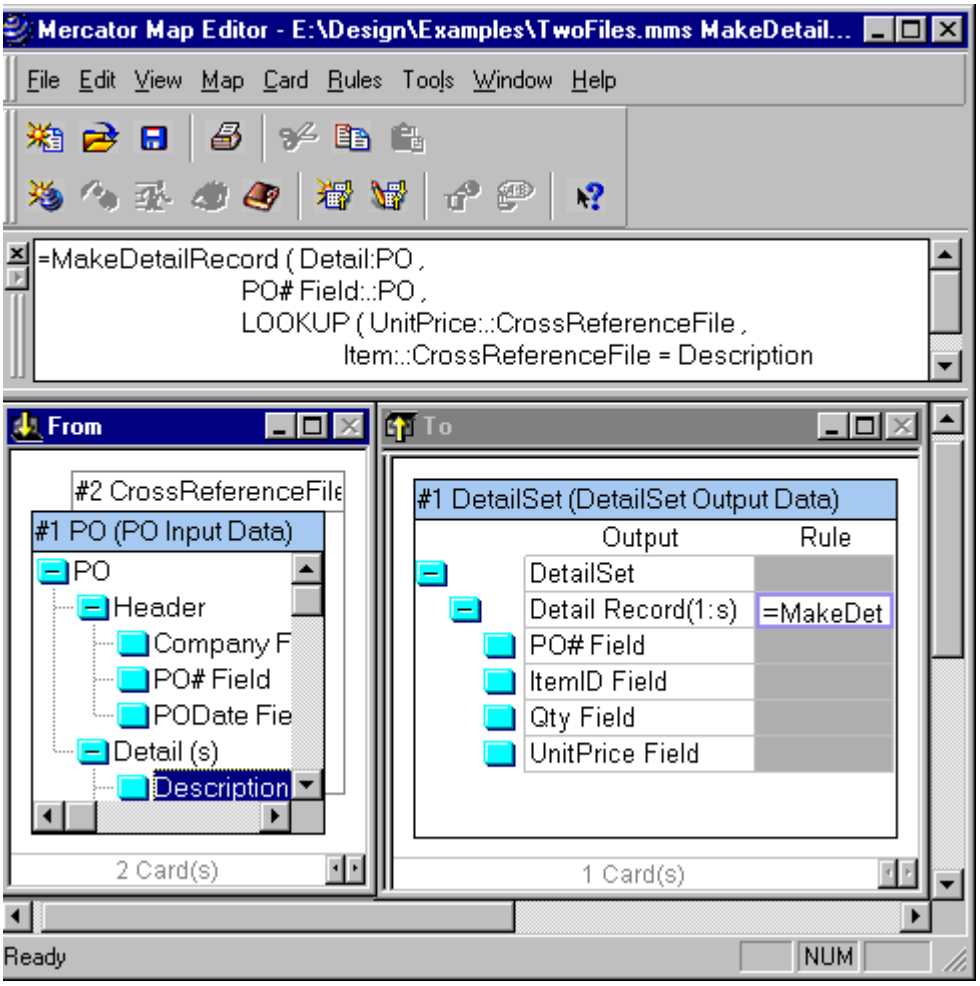
In the functional map **MakeHeaderRecord**, Company is mapped to Customer# and PODate is mapped to PODate. The Account# is the literal `"klr".` The RIGHT function is used to get the right three bytes of the input PO# and map it to the output PO#.

The RIGHT function extracts characters from a text Item, beginning at the rightmost byte of that Item. The second argument specifies how many bytes to extract. The syntax of the RIGHT function is:

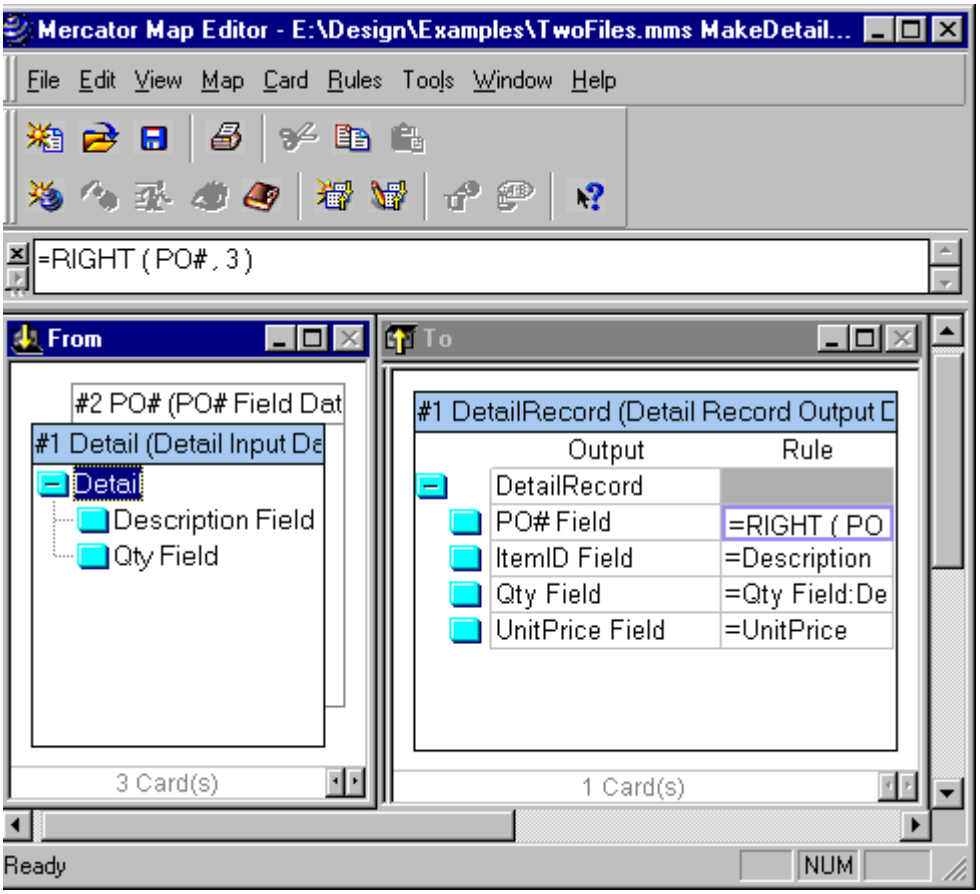
```
RIGHT (Text Item, Number of bytes from right)
```


Mercator Map Editor - E:\Design\Examples\TwoFiles.mms MakeHeaderRec...
File Edit View Map Card Rules Tools Window Help
=RIGHT (PO# Field:POHeader , 3)
From
#1 POHeader (Header Input D
POHeader
Company Field
PO# Field
PODate Field
1 Card(s)
To
#1 HeaderRecord (Header Record Outp
Output
Rule
HeaderRecord
Customer# Field
=Company Field:
Account# Field
="klr"
PO# Field
=RIGHT (PO# F
PODate Field
=PODate Field:P
1 Card(s)
Ready
NUM

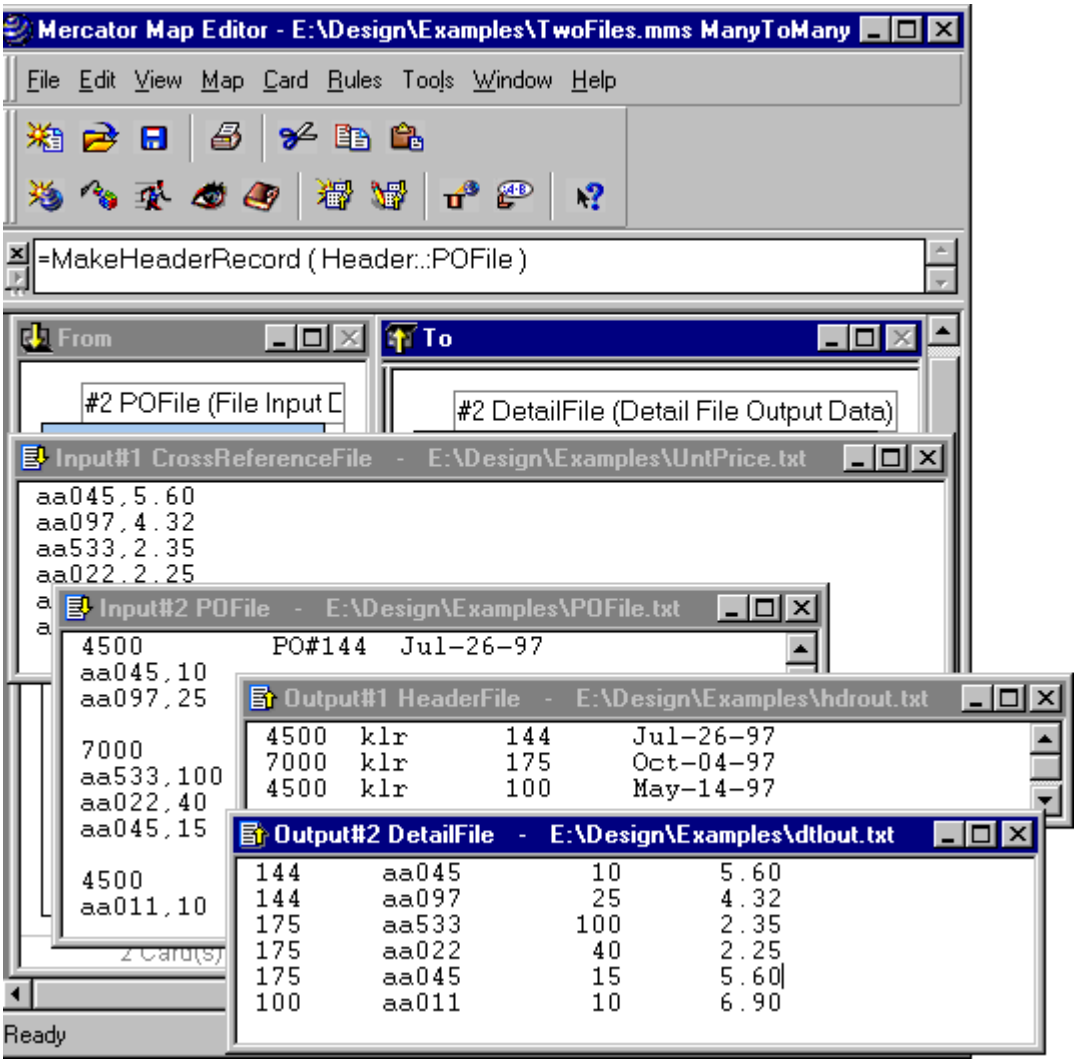
In the functional map **MakeDetailSet**, the map rule on **Detail Record** references the map **MakeDetailRecord**. Its inputs are a Detail, the PO# from the Header, and the unit price for that particular Item.



In the functional map **MakeDetailRecord**, the RIGHT function is used to get the rightmost three bytes of the PO#. The **Description** is mapped to the **ItemID**, the **Quantity** to the **Qty**, and the **UnitPrice** to the **UnitPrice**.



The resulting output files contain all the correct information from the two input files.



Chapter 12 - Arithmetic Functions and Operators

This example uses the arithmetic functions SUM, COUNT, MAX, and ROUND, and the multiplication operator *****.

What You Want to Do

Suppose that you have a **Header** file and a **Detail** file, and you want to map these files to a **Purchase Order** file. In the output **PO**, a trailer includes summary information about the items in the PO.

How to Do It

Define the output file in a type tree.

Create a map that has the **Header** and **Detail** files as inputs, and the **PO** file as the output. To map the trailer, use the arithmetic functions and operators.

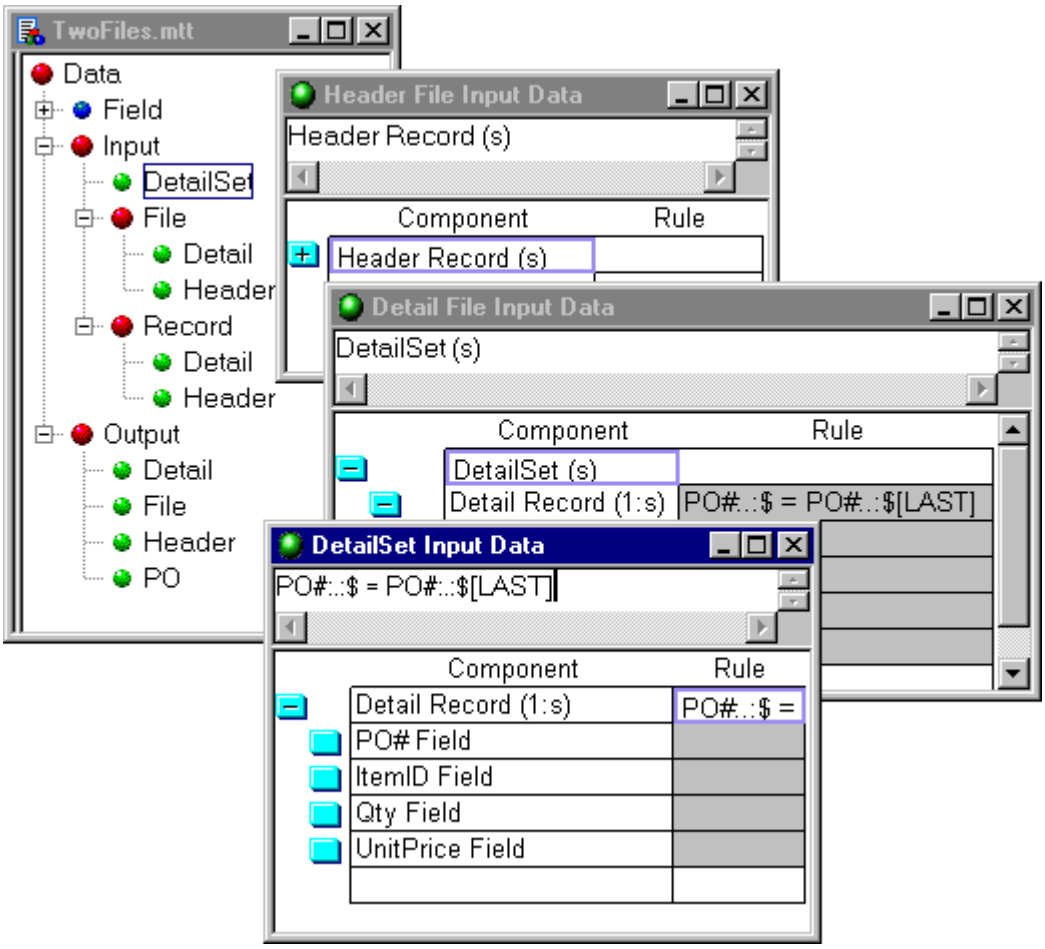
Files Used in this Example

The following table lists the files used in this example.

File	Use
header.txt	Use this data file, which was created in Chapter 10, Case 1, as input.
detail.txt	Use this data file, which was created in Chapter 10, Case 1, as input.
twofiles.mtt	This tree was created in Chapter 10.
math.mtt	You create this type tree to define the output file.
new_pos.mms	You create this map source file.
out_po.txt	This output file is created by running the map.

Using the Type Editor

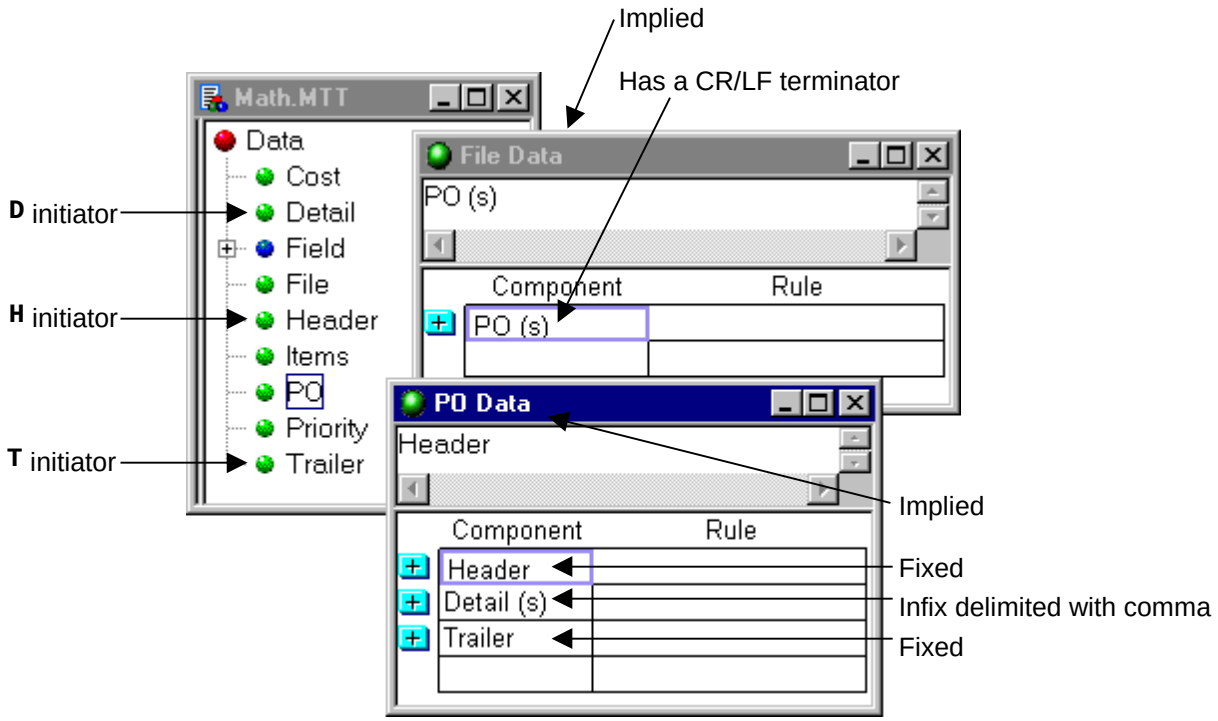
The input files were defined in Chapter 10.



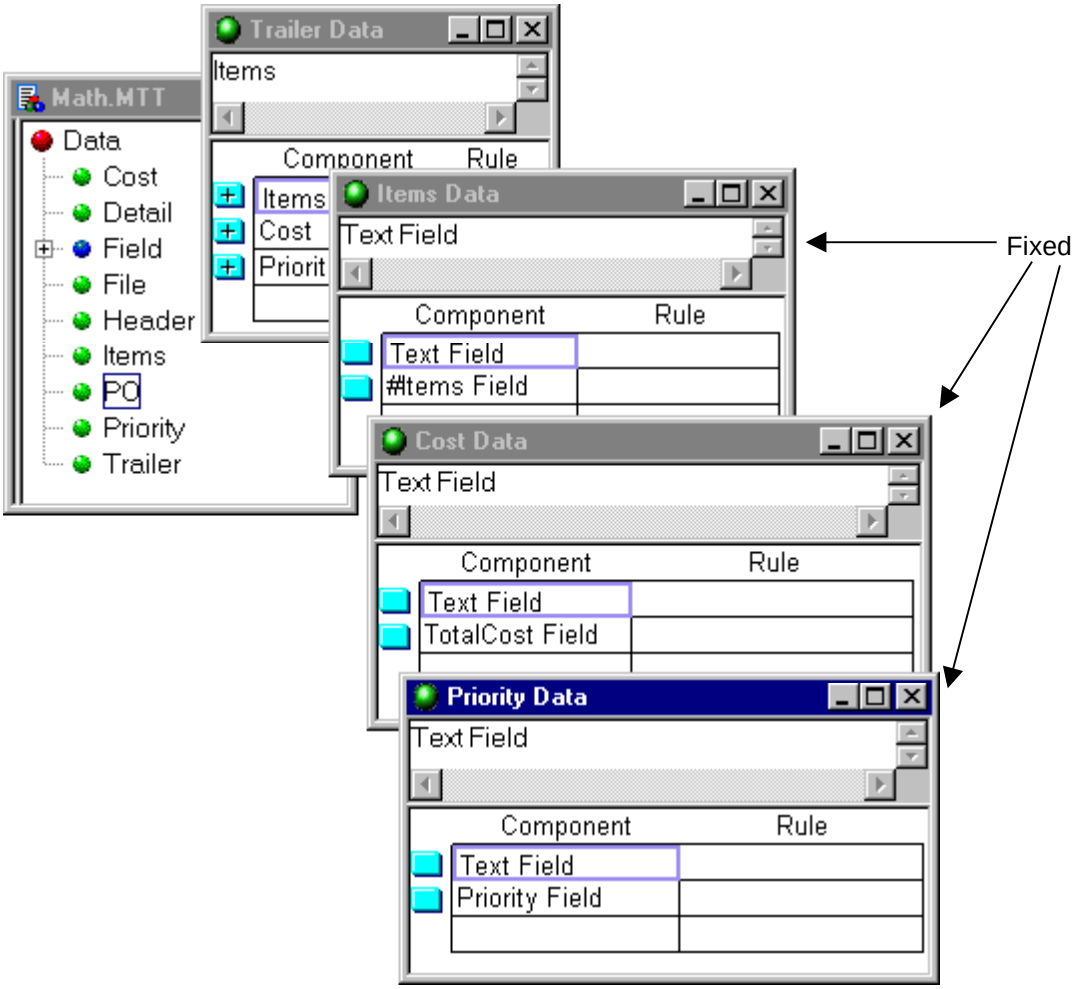
The output file contains POs. Each PO contains a header, a series of details, and a trailer. The output data should look like this:

```
H 4500      144    Jul-26-97
D aa045,10
D aa097,25
T Total items: 2  Total cost: $164.00 Priority:    High

H 7000      175    Oct-04-97
D aa533,100
D aa022,40
D aa045,15
T Total items: 3  Total cost: $384.50 Priority:    Low

H 4500      100    May-14-97
D aa011,10
T Total items: 1  Total cost: $69.00  Priority:    High
```

Define the output data in a type tree. In the PO, the header, detail and trailer begin with “H”, “D”, and “T.” Define these as Initiators.

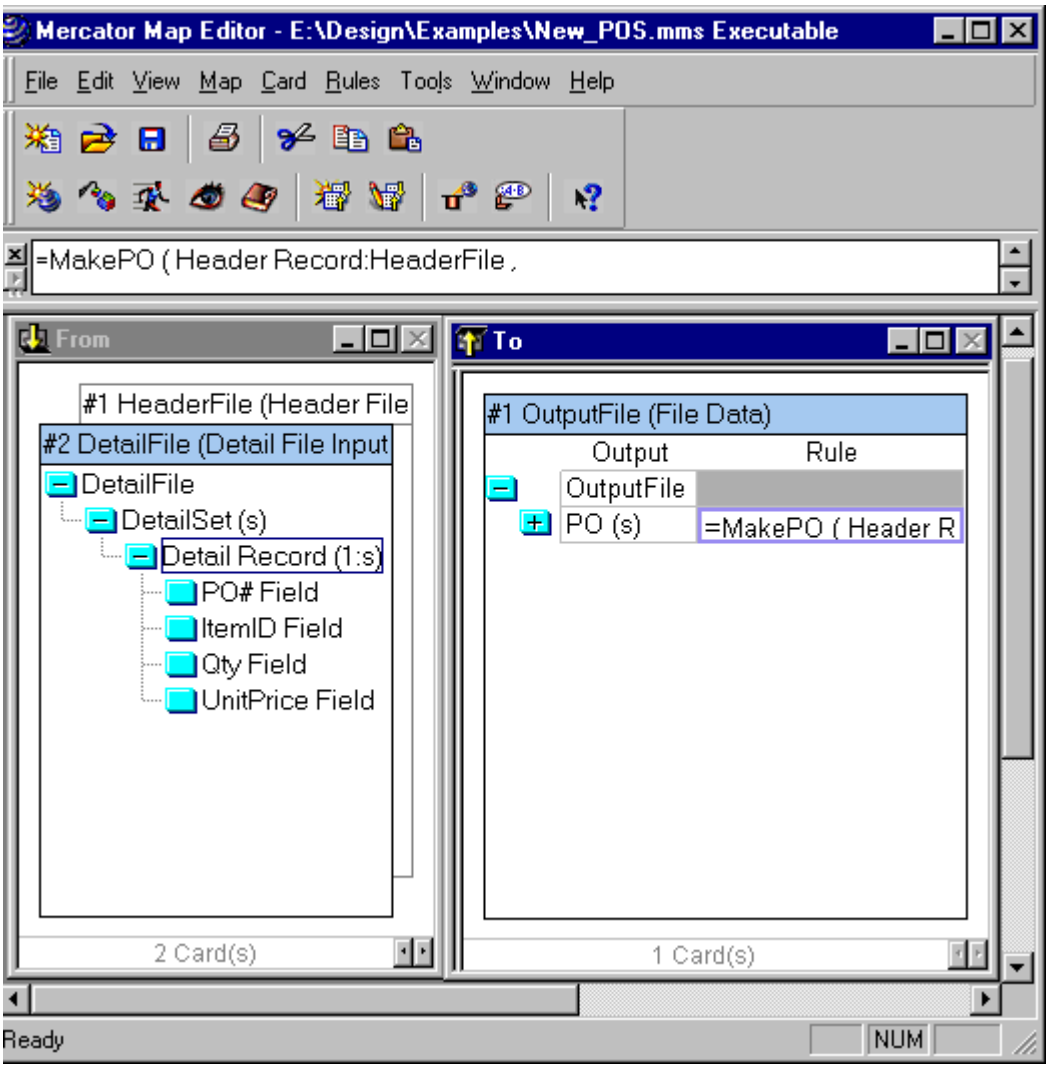


The Trailer is made up the components Items, Cost, and Priority.



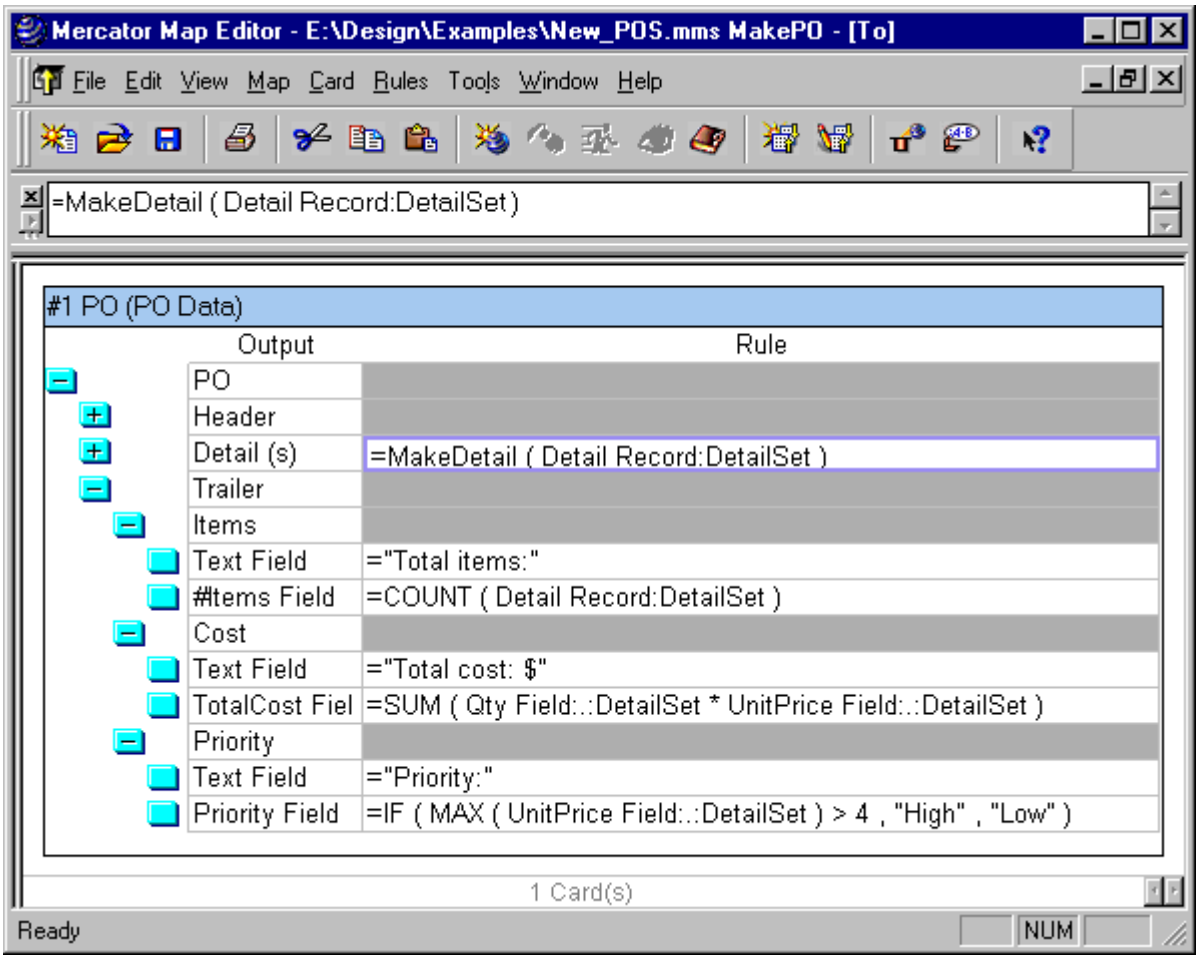
Using the Map Editor

In the executable map, the **Header** and **Detail** files are the inputs, and the **PO** file is the output. The rule on the output PO references the functional map **MakePO**. The arguments to **MakePO** are a Header Record, and the Detail Set whose index matches the index of the Header Record.



In **MakePO**, the **Header** and **Detail** records are mapped as they are in Chapter 10.

Here are the map rules for the components of **Trailer**:



The #Items field is mapped using the COUNT of the **Detail Records**.

The TotalCost field is calculated by taking the SUM of the products of Qty*UnitPrice. Then, it is rounded to two decimal places, with the ROUND function.

The SUM function computes the sum of a series of objects. The syntax is:

```
SUM (Series whose sum you want)
```

The ROUND function evaluates a numeric Item. It rounds the number to the number of decimal places specified in the second argument. The syntax for ROUND is:

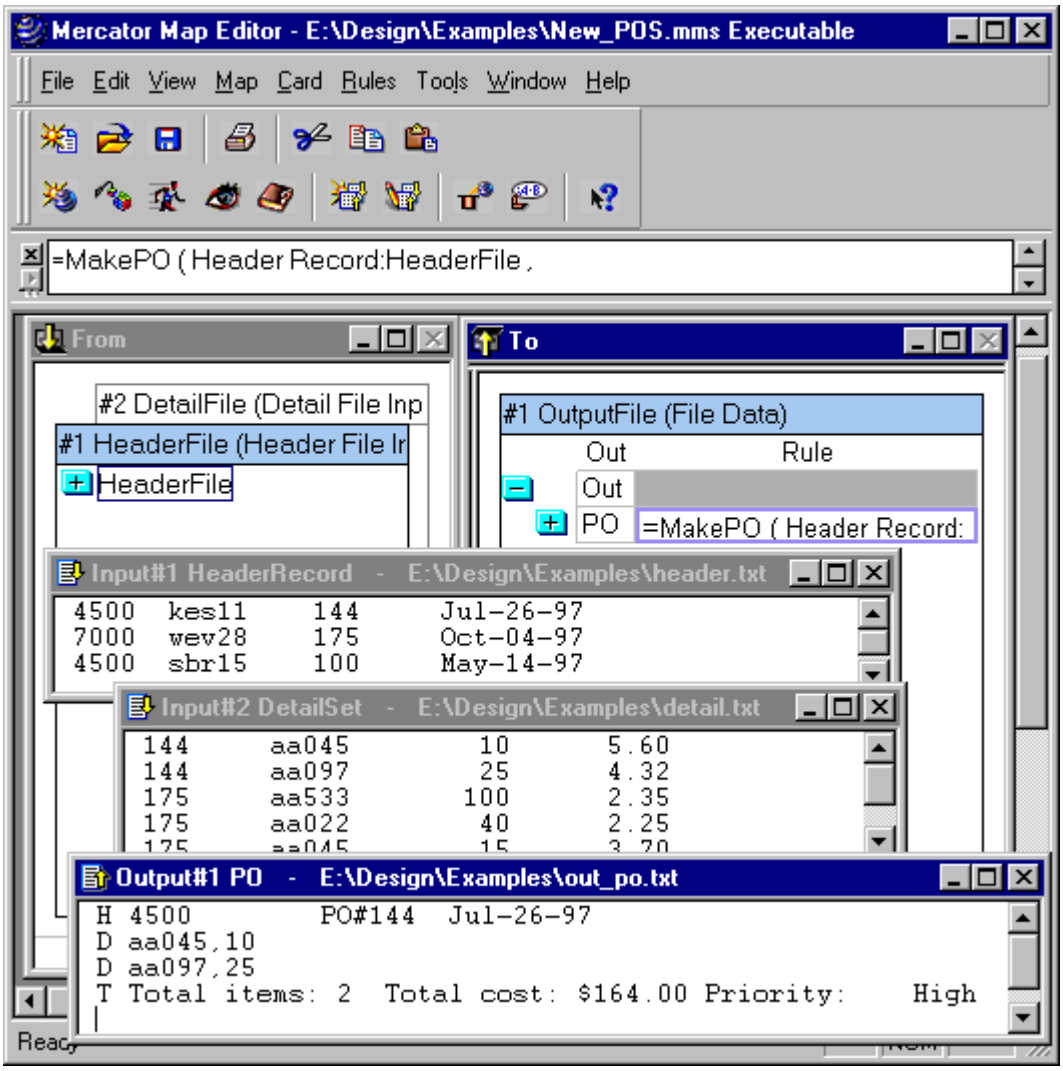
```
ROUND (Numeric Item, Number of decimal places to round to)
```

The rule on the HighPriority field uses the IF function, and the MAX function, to see if the maximum UnitPrice in the PO is greater than 4. If it is, the value for the HighPriority field is "Yes," otherwise, it's "No."

The MAX function returns the maximum value of a series of numeric objects. Its syntax is:

```
MAX (Number series whose maximum value you want)
```

After running the map, here are the results, showing the two input files, and the new output:



Chapter 13 - Ignoring Invalid Data

This example shows how to define your data if you want Mercator to ignore invalid data.

What You Want to Do

You have a file containing your customer contacts. You want to map the customer data to a file that gives a summary of the customer data. If Mercator finds an invalid customer record, you want Mercator to continue mapping the data.

How to Do It

Define the input customer data in a type tree.

Define the output customer data in a type tree.

Create a map that maps the input customer data to the output customer data.

Files Used in this Example

The following table lists the files used in this example.

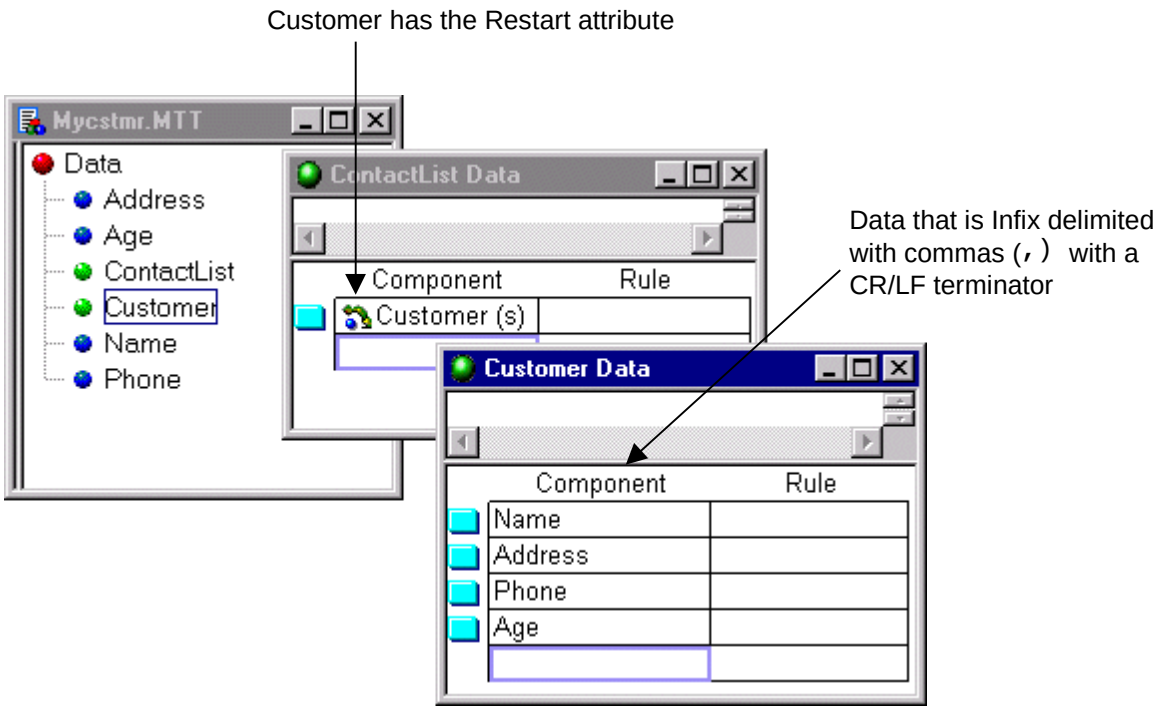
File	Use
mycstmrs.mtt	You create this type tree to define your input customer data.
cust_out.mtt	You create this type tree to define your output customer data.
mycstmrs.mms	You create this map source file to map the input data to the output data.
names.txt	Use this file as the input data file.
cust_out.txt	This output file is created by running the map.

Using the Type Editor

You have an input file of customer contacts. The third customer record is invalid. The Age field is supposed to be an integer, but it is a decimal number.

```
Gisela,1252 S. Broward/Ft. Lauderdale,523-2622,22
Dede,1513 Palatine Rd./Boca Raton,252-6560,86
Lewis,74099 S. 67th Ave/Ft. Lauderdale,332-8665,373
Eric,6933 Main St./South Miami Beach,291-7281,56
```

Bad data



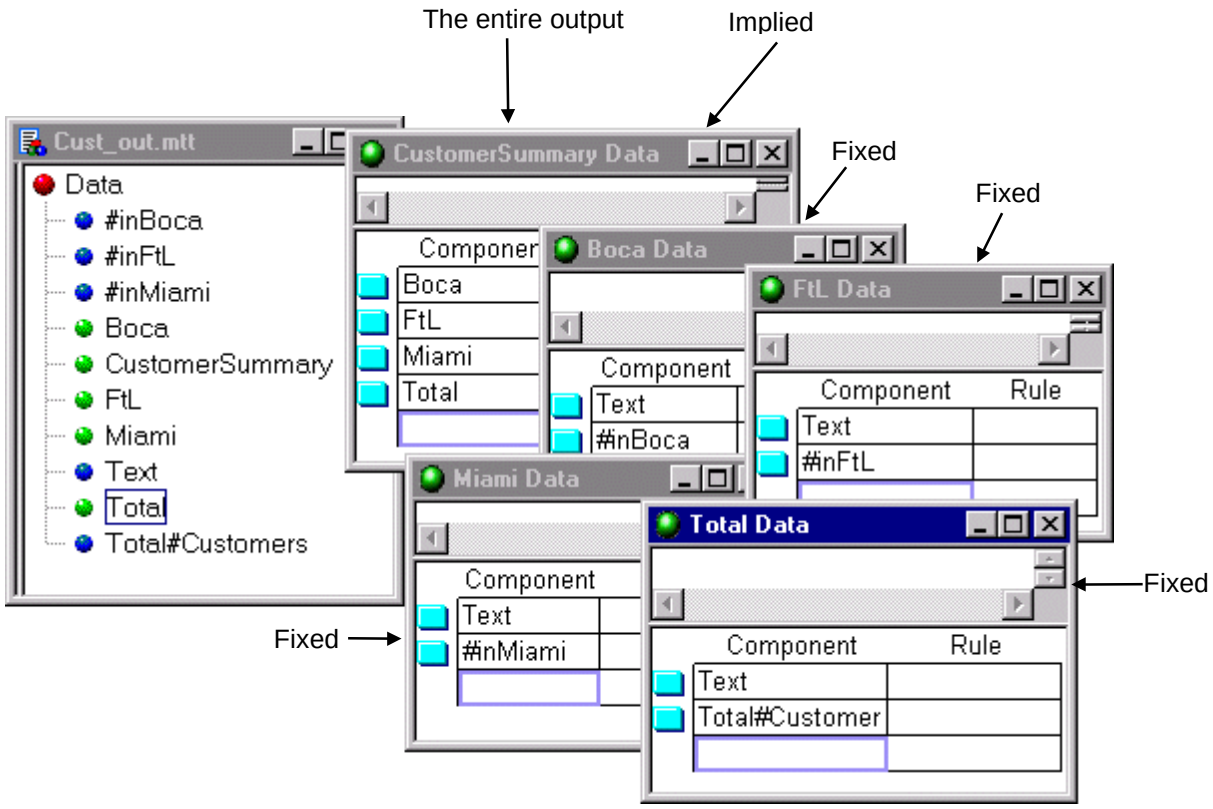
When you want Mercator to ignore an invalid object, you assign the Restart attribute to that component. You want Mercator to ignore any invalid Customer, so you assign the Restart to it.

To assign the Restart attribute to a given component, select the component, and then choose Restart from the Component menu.

You want the output data to look like this:

```
# customers in Boca:    1
# customers in Ft. L:   1
# customers in Miami:   1
Total # customers:      3
```

Define the output in a type tree:



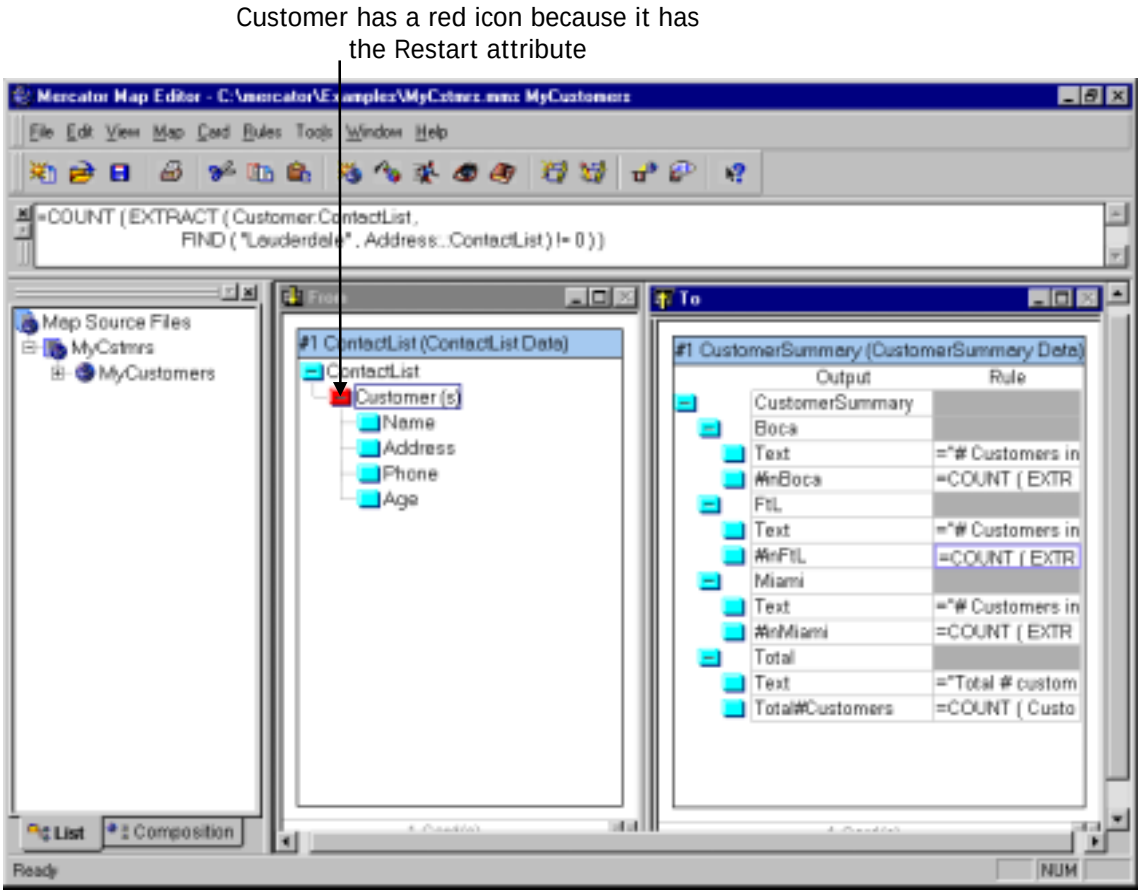
Using the Map Editor

In the Map Editor, any component that has the Restart attribute has a red icon. There is a red icon on Customer. The rule for calculating the number of customers in each city uses the EXTRACT and FIND functions. For example, the rule on the output #inBoca counts the occurrences of Customers where the text “Boca” is in the Address field:

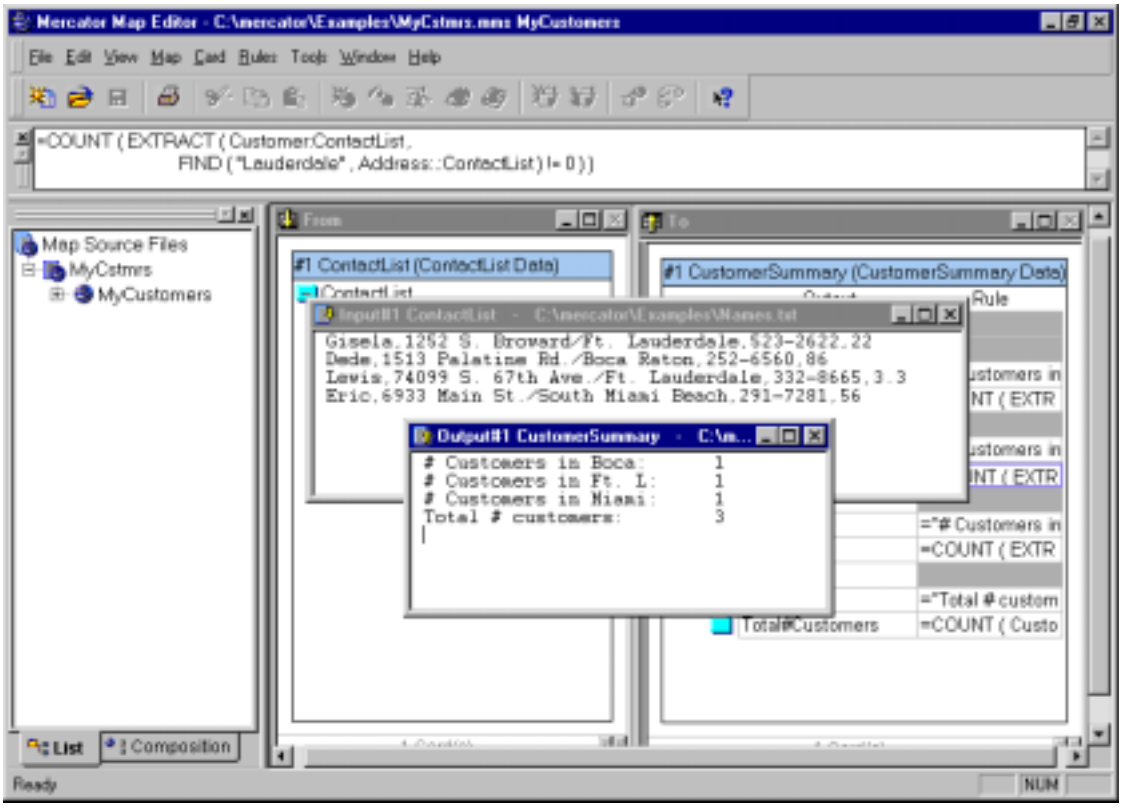
```
= COUNT (EXTRACT (Customer:ContactList,
FIND ("Boca", Address:.:ContactList) !=0))
```

The rule on the output Total#Customers is simply the COUNT of the Customers in the input:

```
= COUNT (Customer:ContactList)
```



Because Customer has the Restart attribute, when Mercator comes across the invalid third Customer record, Mercator will continue to validate the data.



Notice that, even though there are two customers from Ft. Lauderdale, the third Customer—from Ft. Lauderdale—is invalid. It is not included in the calculations in the output rules. For example, the total number of customers is 3—because there were only three valid Customer records.

Chapter 14 – Mapping Invalid Data

This chapter discusses the REJECT function, and shows you how to map invalid data to an Error file. The example is a continuation of the example in Chapter 13.

What You Want to Do—Mapping Invalid Data to a File

You want to map the invalid data in the input Customer file to an Error Eeport.

How to Do It

In the Type Editor, define the Error Eeport. In the Map Editor, create an additional output card, to map the invalid data. Use the REJECT function.

Files Used in this Example

The following table lists the files used in this example.

File	Use
mycstmrs.mtt	This type tree was created in Chapter 13.
cust_out.mtt	This type tree was created in Chapter 13.
errors.mtt	You create this type tree to define the error file.
mycstmrs.mms	This map source file was created in Chapter 13.
names.txt	Use this file as input data. It was used in Chapter 13.
cust_out.txt	This output file is created by running the map.
errors.txt	This output file is created by running the map.

Using the Type Editor

An error file is simply an output file. To define the invalid data, create a text Item that has a minimum size of 0, and no maximum size.

The input data was defined in Chapter 13. Here is the input data. The third Customer is invalid:

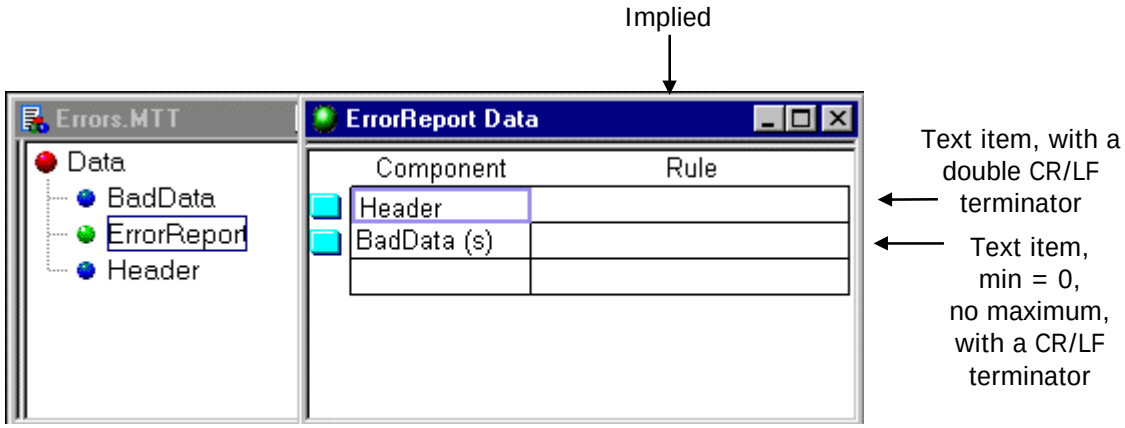
```
Gisela,1252 S. Broward/Ft. Lauderdale,523-2622,22
Dede,1513 Palatine Rd./Boca Raton,252-6560,86
Lewis,74099 S. 67th Ave/Ft. Lauderdale,332-8665,3.3
Eric,6933 Main St./South Miami Beach,291-7281,56
```

Bad data

Suppose you want the error file to look like this:

```
These are the invalid customers:

Lewis,74099 S. 67th Ave/Ft. Lauderdale,332-8665,3.3
```

Define the error file in a type tree. The Item "BadData" can be used to map an invalid customer. It is defined as a text Item, with a minimum size of 0, and no maximum size. You define a series of BadData(s) as a component of the ErrorReport—this way, each invalid customer will appear in the error report. In this example, there is just one invalid customer.



Using the Map Editor

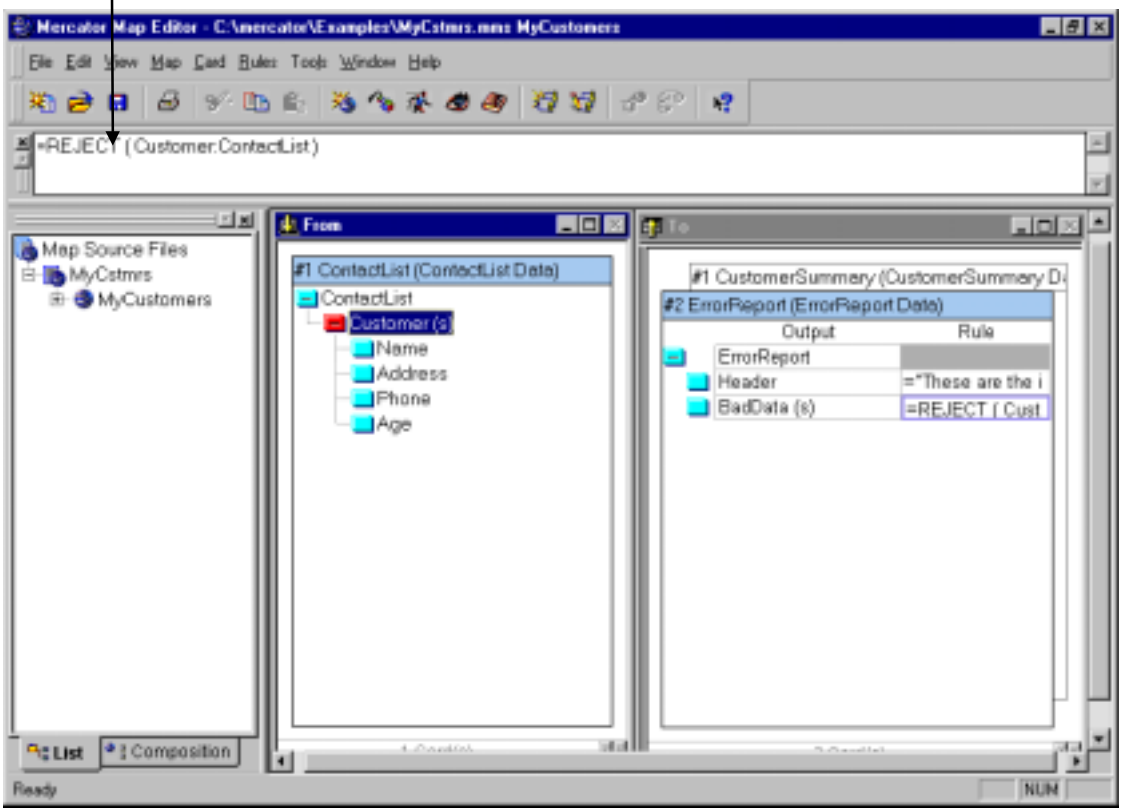
To map the invalid data, you can simply add another output card to the executable map from Chapter 13.

To map the invalid customers to the output BadData, use the REJECT function.

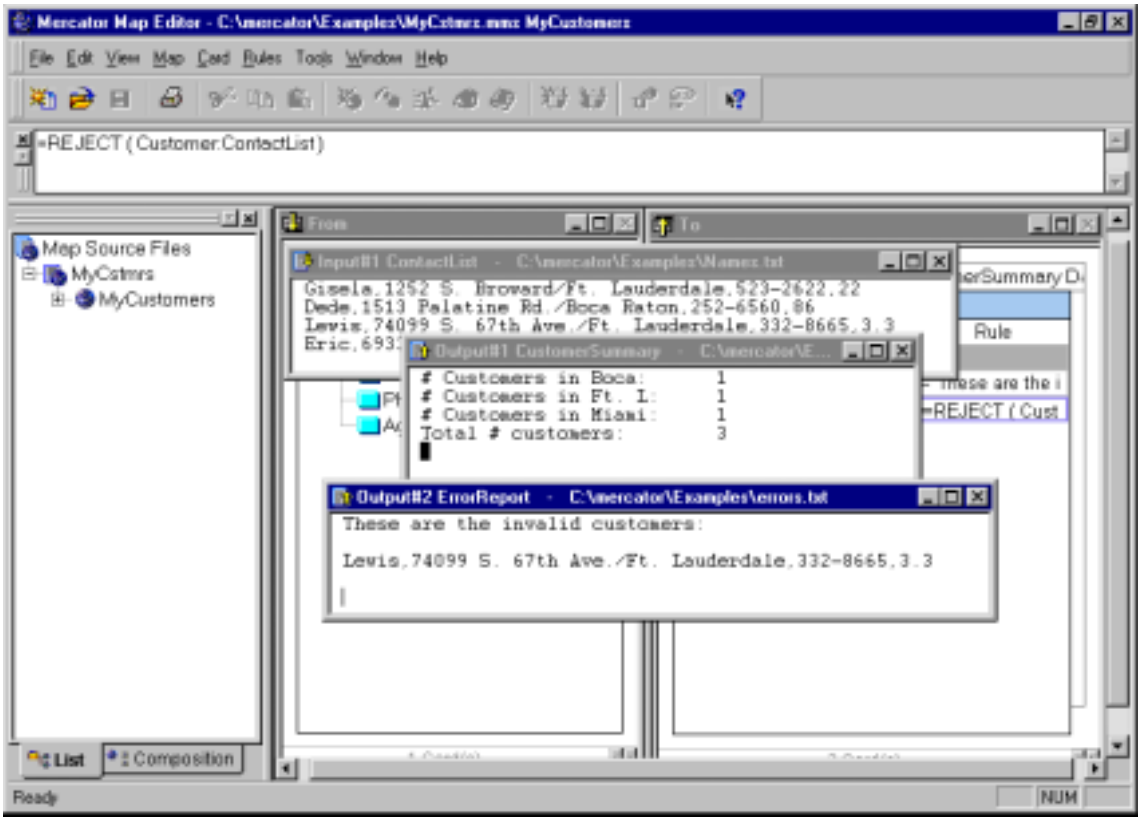
The REJECT function evaluates a series of objects, of a type that has the Restart attribute. It returns a series of text Items—each of which is an invalid object. The syntax is:

REJECT (Series that has the Restart attribute)

Use the REJECT function to map the invalid Customers



The resulting error file includes the invalid customer:


Mercator Map Editor - C:\mercator\Examples\MyCstmrs.mmc MyCustomers
File Edit View Map Card Rules Tools Window Help
=REJECT (Customer.ContactList)
Map Source Files
MyCstmrs
MyCustomers
List
Composition
From
To
Input#1 ContactList - C:\mercator\Examples\Names.txt
Gisela,1252 S. Broward/Ft. Lauderdale,523-2622,22
Dede,1513 Palatine Rd./Boca Raton,252-6560,86
Lewis,74099 S. 67th Ave./Ft. Lauderdale,332-8665,3.3
Output#1 CustomerSummary - C:\mercator\E...
Customers in Boca: 1
Customers in Ft. L: 1
Customers in Miami: 1
Total # customers: 3
Output#2 ErrorReport - C:\mercator\Examples\errors.txt
These are the invalid customers:
Lewis,74099 S. 67th Ave./Ft. Lauderdale,332-8665,3.3
Rule
=REJECT (Cust
Ready
NUM

Chapter 15 – Using Logical Functions

There are four examples in this chapter. Each example uses a combination of the logical functions OR, ALL, IF, and EITHER.

Case 1 – Using the OR Function

The OR function evaluates a boolean expression about a series of objects. The OR function returns the boolean TRUE if *any* evaluation of the expression comes out to TRUE, or the boolean FALSE if *none* of the evaluations come out to TRUE. The syntax of the OR function is:

```
OR (Boolean expression about a series of objects)
```

What You Want to Do

You have a file that lists the number of phone calls that were received at your company's stores. You have another file that lists the stores you are interested in collecting data for.

You want to generate a file that contains the calls for just the selected stores, and gives the total number of calls for these stores.

How to Do It

Define the two input files and the output file in a type tree.

Create a map that has the Calls file and the Stores file as inputs, and the Summary file as the output.

Files Used in Case 1

Use these files, which you create as you work through this example:

File	Use
calls.txt	You create this file to use as an input data file.
stores.txt	You create this file to use as an input data file.
stores.mtt	You create this type tree that defines the data

files.

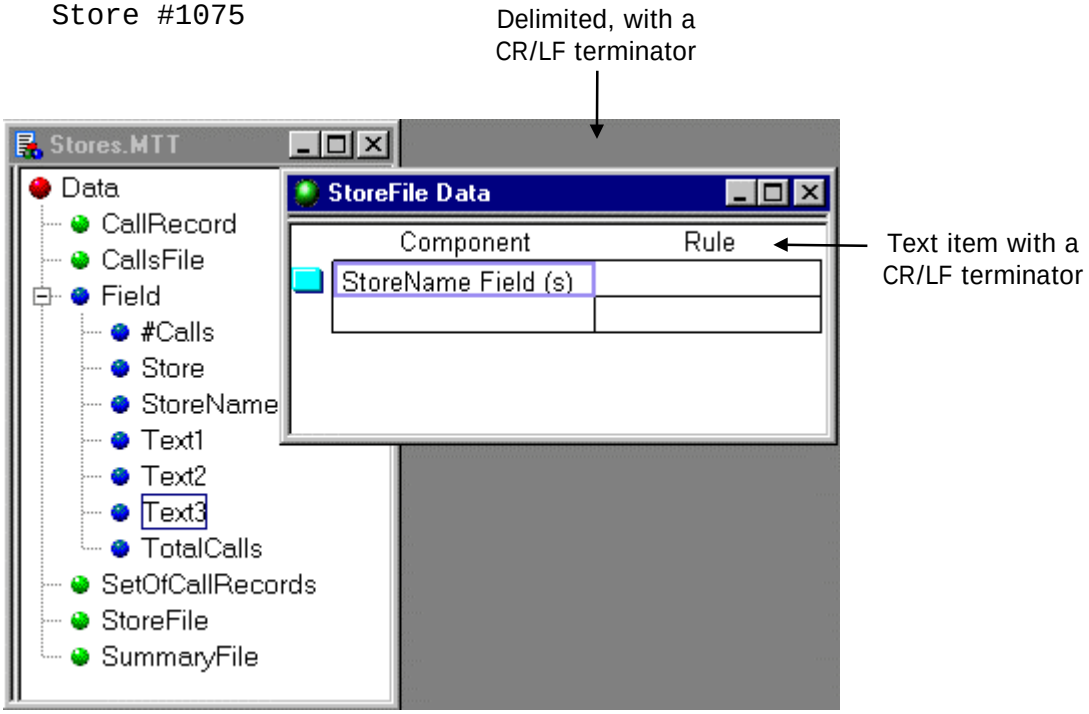
stores.mms You create this map source file that contains the map explained in this example.

summary.txt This output file is created by running the map.

Using the Type Editor

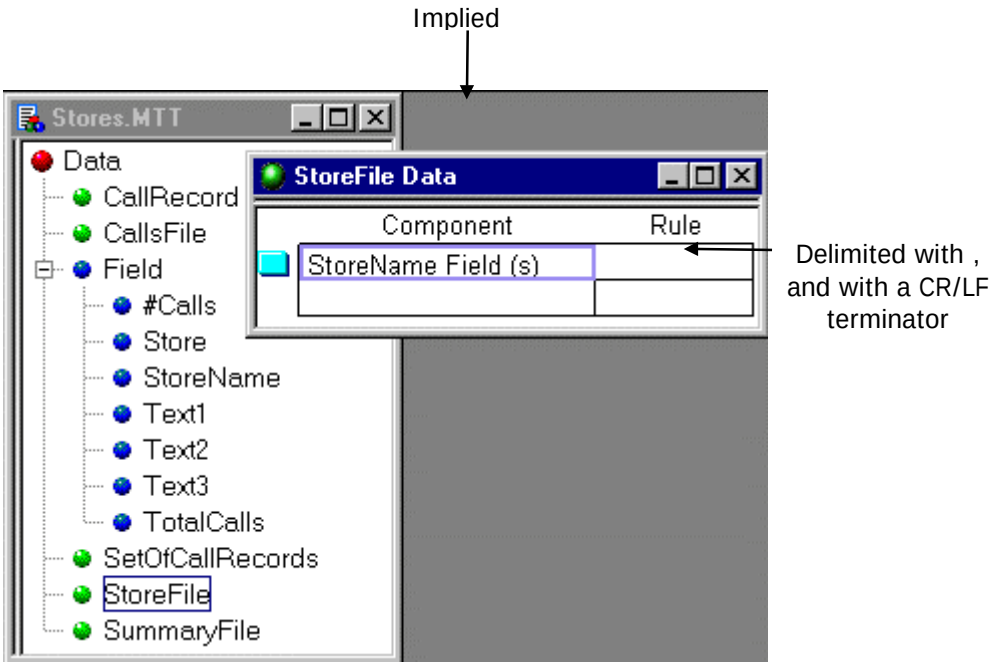
The stores data looks like this:

```
Store #1431
Store #1492
Store #1939
Store #1075
```



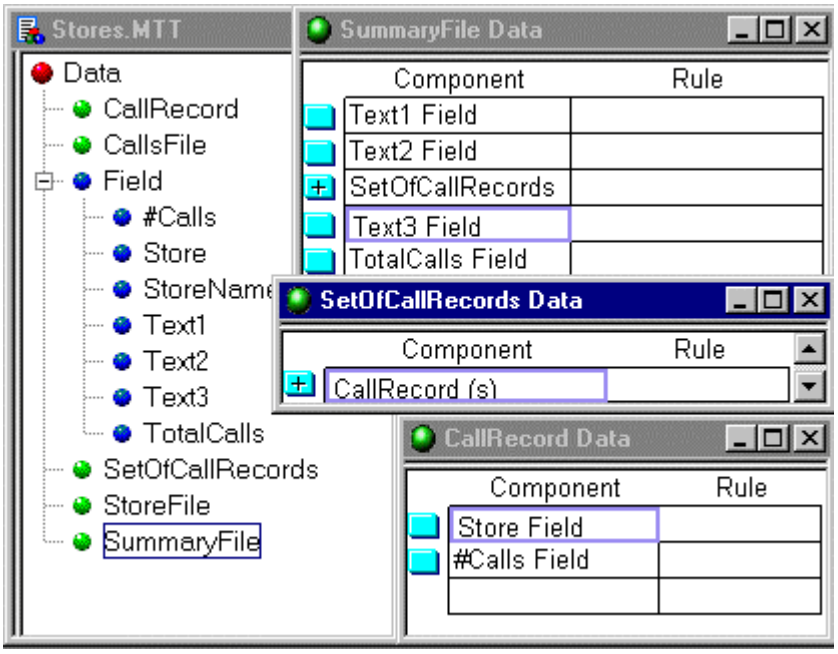
In the Calls data, the first field in each record is the store number, and the second field is the number of calls. The Calls data looks like this:

```
Store #1208,500
Store #1939,1020
Store #1488,536
Store #1431,750
```



The output data should look like this:

```
Calls at Selected Stores
************************

Store #1939,1020
Store #1431,750

TOTAL NUMBER OF CALLS:  1770
```

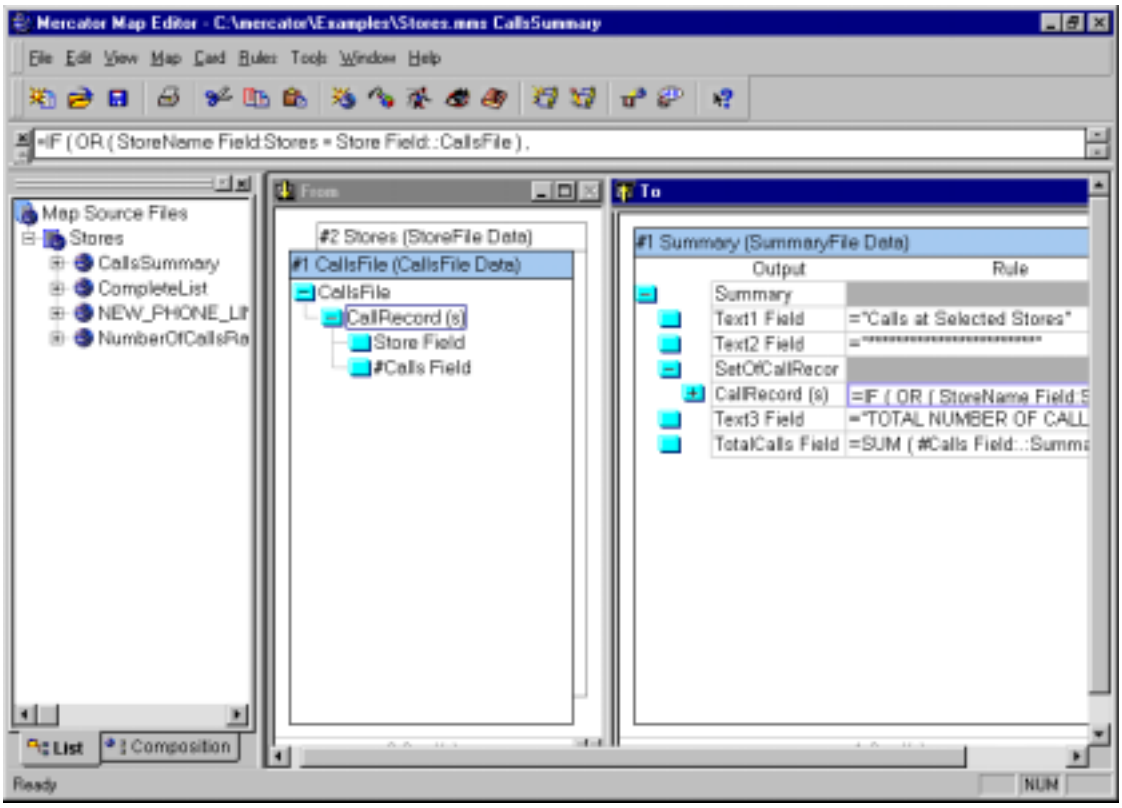


Using the Map Editor

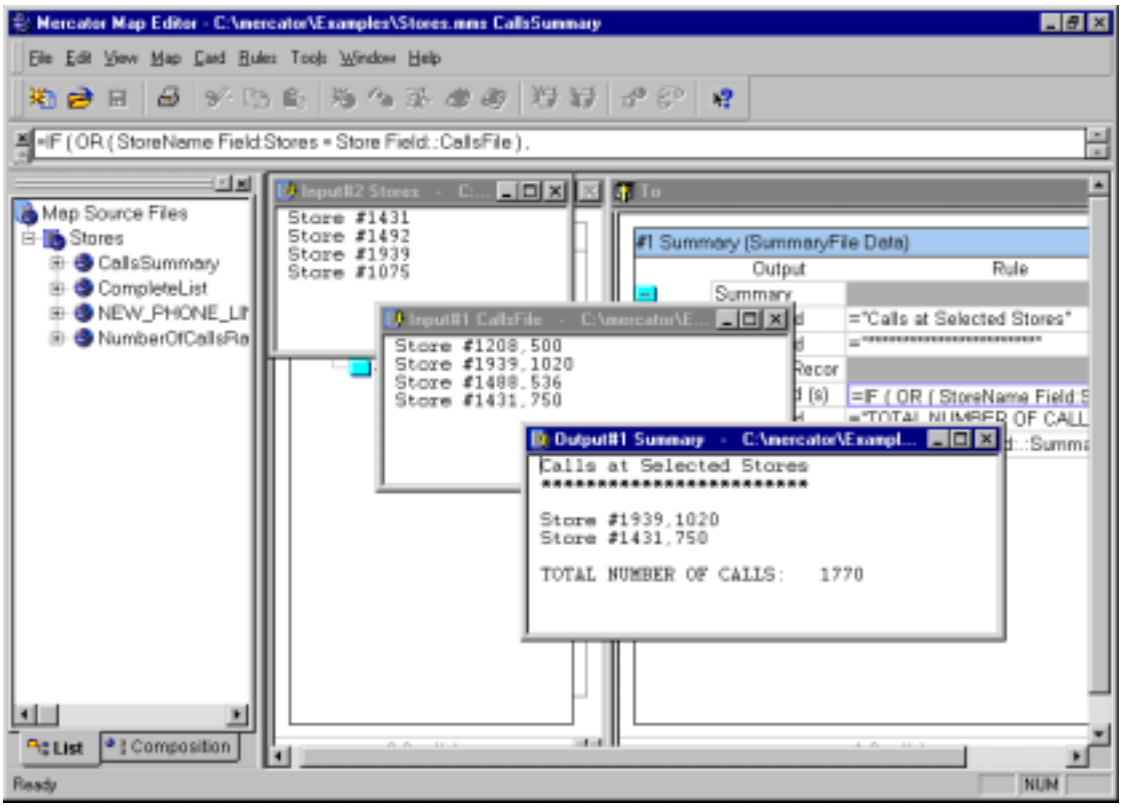
You do not need a functional map, because the CallRecord in the input is exactly the same type as the CallRecord in the output. In the map rule for CallRecord, use the OR function, to map only the CallRecords whose Store field matches the StoreName field in the stores file.

The map rule for CallRecord is:

```
= IF (OR (StoreName Field:Stores = Store Field:.:CallsFile),
CallRecord:CallsFile,
NONE)
```


Mercator Map Editor - C:\mercator\Examples\Stores.mms CallsSummary
File Edit View Map Card Rules Tools Window Help
=IF (OR (StoreName Field:Stores = Store Field::CallsFile) ,
Map Source Files
Stores
CallsSummary
CompleteList
NEW_PHONE_LI
NumberOfCallsRe
List
Composition
From
#2 Stores (StoreFile Data)
#1 CallsFile (CallsFile Data)
CallsFile
CallRecord (s)
Store Field
#Calls Field
To
#1 Summary (SummaryFile Data)
Output
Rule
Summary
Text1 Field
="Calls at Selected Stores"
Text2 Field
SetOfCallRecor
CallRecord (s)
=IF (OR (StoreName Field:S
Text3 Field
="TOTAL NUMBER OF CALL
TotalCalls Field
=SUM (#Calls Field::Summa
Ready
NUM

The results include only the stores that are found in the Stores file:



Case 2 – Using the ALL Function

The ALL function evaluates a boolean expression about a series of objects. The ALL function returns the boolean TRUE if *all* evaluations of the expression come out to TRUE, or the boolean FALSE if *any* evaluation of the expression comes out to FALSE. The syntax of the ALL function is:

ALL (Boolean expression about a series of objects)

What You Want to Do

Suppose you have a file that rates the volume of phone calls to the different stores in your company, over a number of different dates. The volume for each store is rated low, medium, or high.

You may need to install new phone lines in a store, if the volume of calls is consistently high. So, you want to generate a file that lists the stores that have a rating of “high” for each date.

How to Do It

Define the input and output file in a type tree. Create a map that maps the input file to the output file. To map a store name to the output, use the ALL function, to determine if the store's volume was consistently high.

Files Used in Case 2

Continue working with the map source file created in Case 1. You need to create the input files, containing the data described in the following topic, "Using the Type Editor."

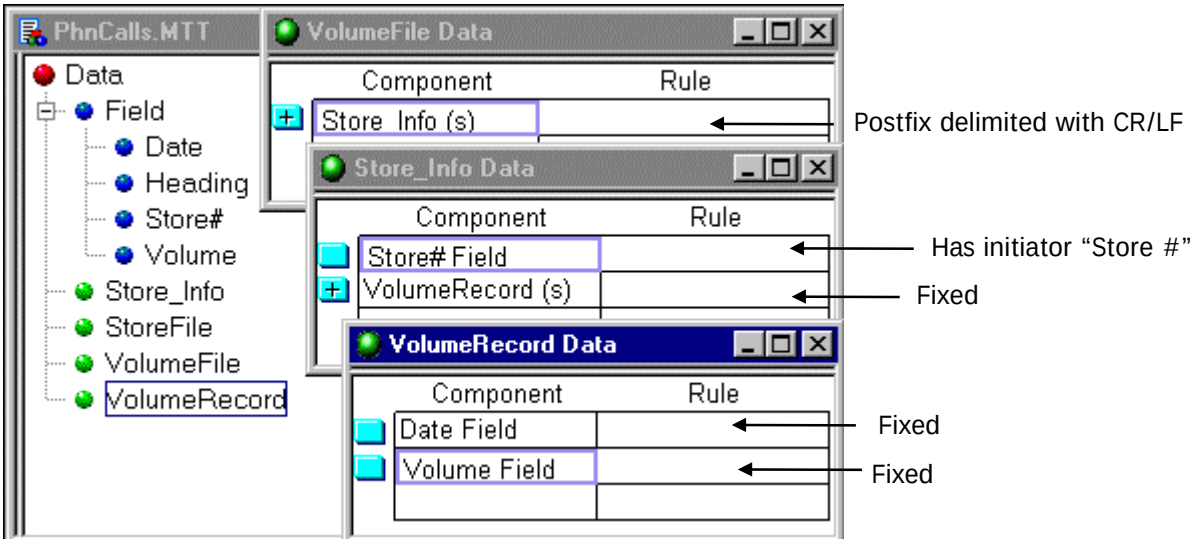
File	Use
volume.txt	You create this file to use as an input data file.
phncalls.mtt	You create this type tree that defines the data files.
stores.mms	You modify this map source file, which was created in Case 1, to include the map for this example.
newlines.txt	This output file is created by running the map.

Using the Type Editor

The input data looks like this:

```
Store #1075
09/07/97    high
09/22/97    high
09/30/97    medium
Store #1939
09/10/97    high
09/23/97    high
09/29/97    high
Store #1208
09/01/97    medium
09/12/97    low
09/26/97    medium
09/30/97    low
Store #1005
09/02/97    high
09/07/97    high
```

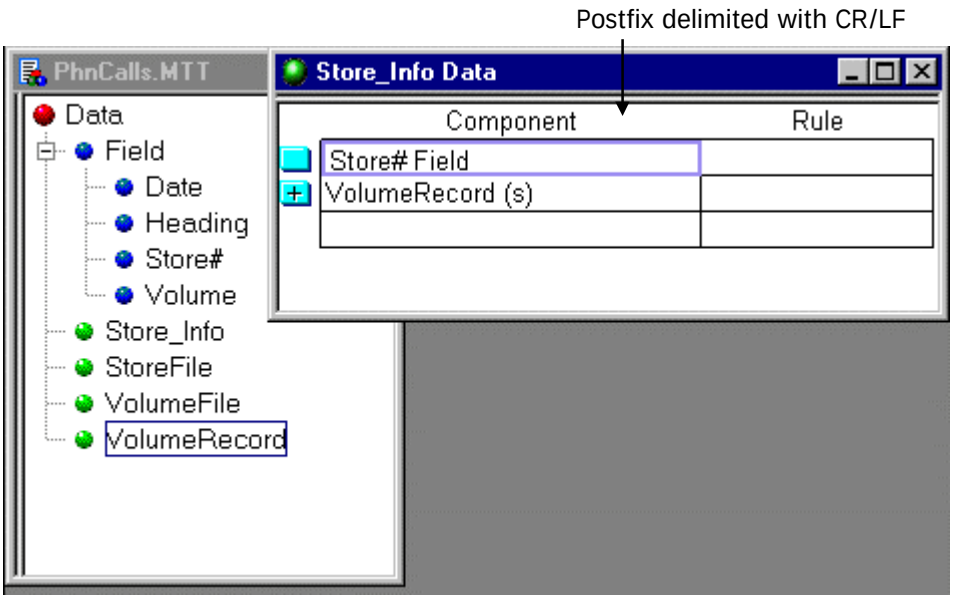
This Volume file is made up of a series of StoreInfos.



Your output should look like this:

```
We need new phone lines in the following stores:
Store #1939
Store #1005
```

The Store Info file is made up of a Store# field and Volume records.



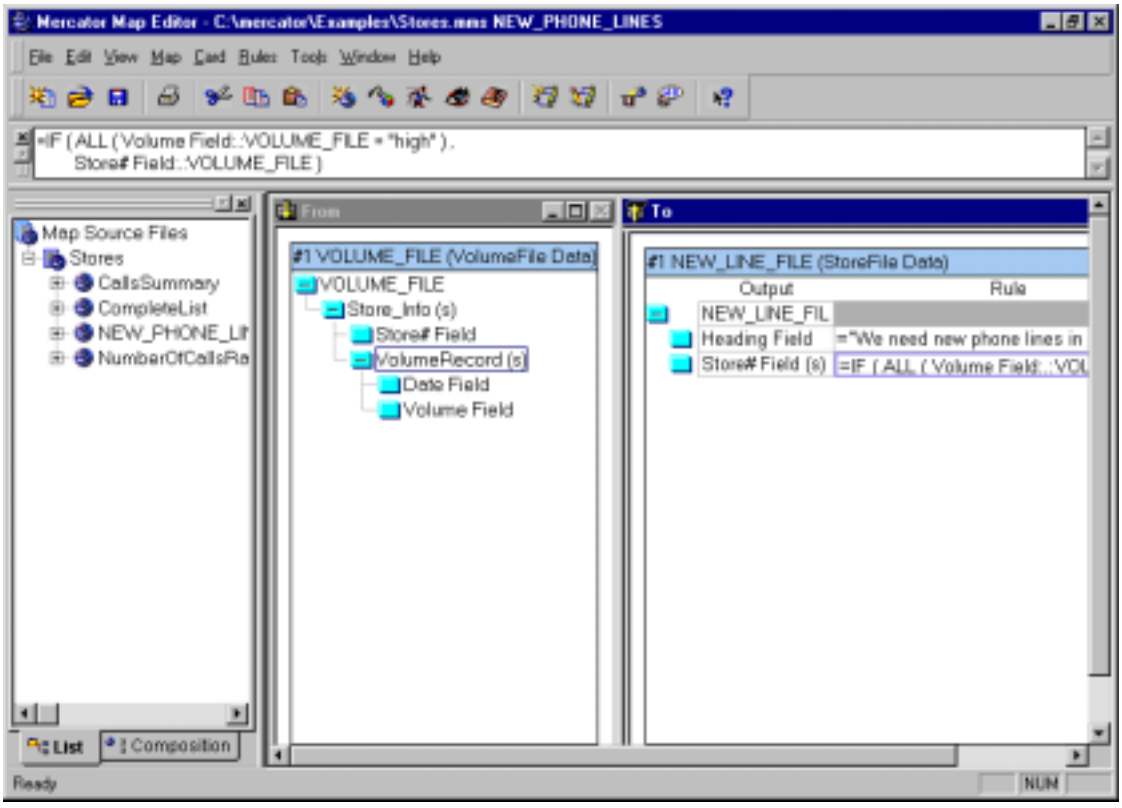
Using the Map Editor

In the map, you want to map the Store# field if all of the Volume fields for that store contain the value “high.” To do this, use the ALL function.

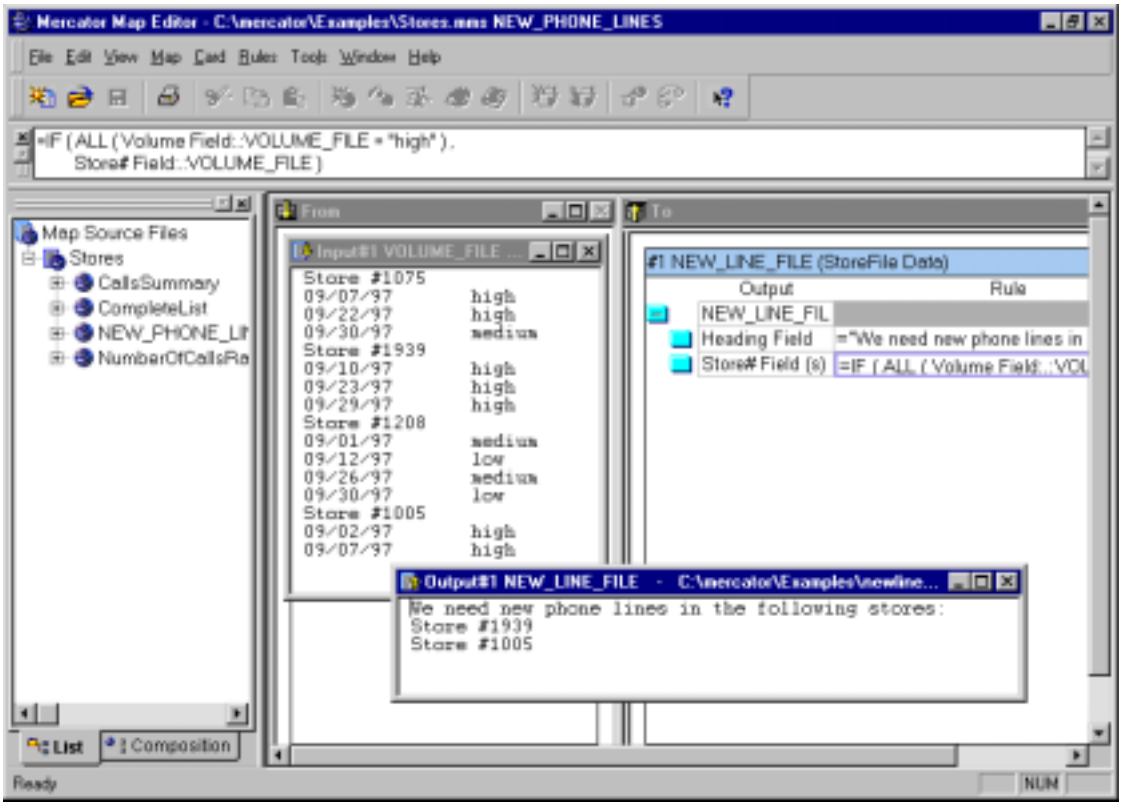
The map rule for Store# field is:

```
= IF (ALL (Volume Field:.:VOLUME_FILE = "high"), Store#
Field:.:VOLUME_FILE, none)
```

The executable map looks like this:



The results show that Mercator mapped only the stores that had the rating “high” in every Volume field.



Case 3 – Using the EITHER Function

This example uses the EITHER function to map different data if an object evaluates to NONE.

What You Want to Do

You want to generate a file that lists all of the stores in the Calls file, and indicates whether or not the store is listed in the Stores file.

How to Do It

Define the output data in a type tree. Create a map that maps the stores and calls data to the output file. Use the EITHER and LOOKUP functions to determine if a given store appears in the Stores file. If the store does not appear, put the text “Not in our stores list.”

Files Used in Case 3

This example uses the map source file, the type tree, and the input data files created in Case 1.

File	Use
stores.txt	Use this file, created for Case 1, as an input data file.
calls.txt	Use this file, created for Case 1, as an input data file.
stores.mtt	Use this type tree, created in Case 1, for both input and output types.
stores.mms	You modify this map source file, which was created in Case 1 and modified in Case 2, to include the map for this example.
list.txt	This output file is created by running the map.

Using the Type Editor

The output data should look like this:

```
Store #1208 Not in our stores list
Store #1939
Store #1488 Not in our stores list
Store #1431
```

This example shows how you can reuse types for many sources and/or destinations. The output type is the same as the type of one of the input cards—StoreFile.

Using the Map Editor

You want to map a store name if it is present in the Stores file. If it is not present in the Stores file, you want to map it, and add the text “Not in our stores list.” To do this, use the EITHER function.

The EITHER function has two arguments. If the first argument evaluates to a value other than NONE, the result is the first argument. If the first argument evaluates to NONE, the result is the second argument. The syntax of the EITHER function is:

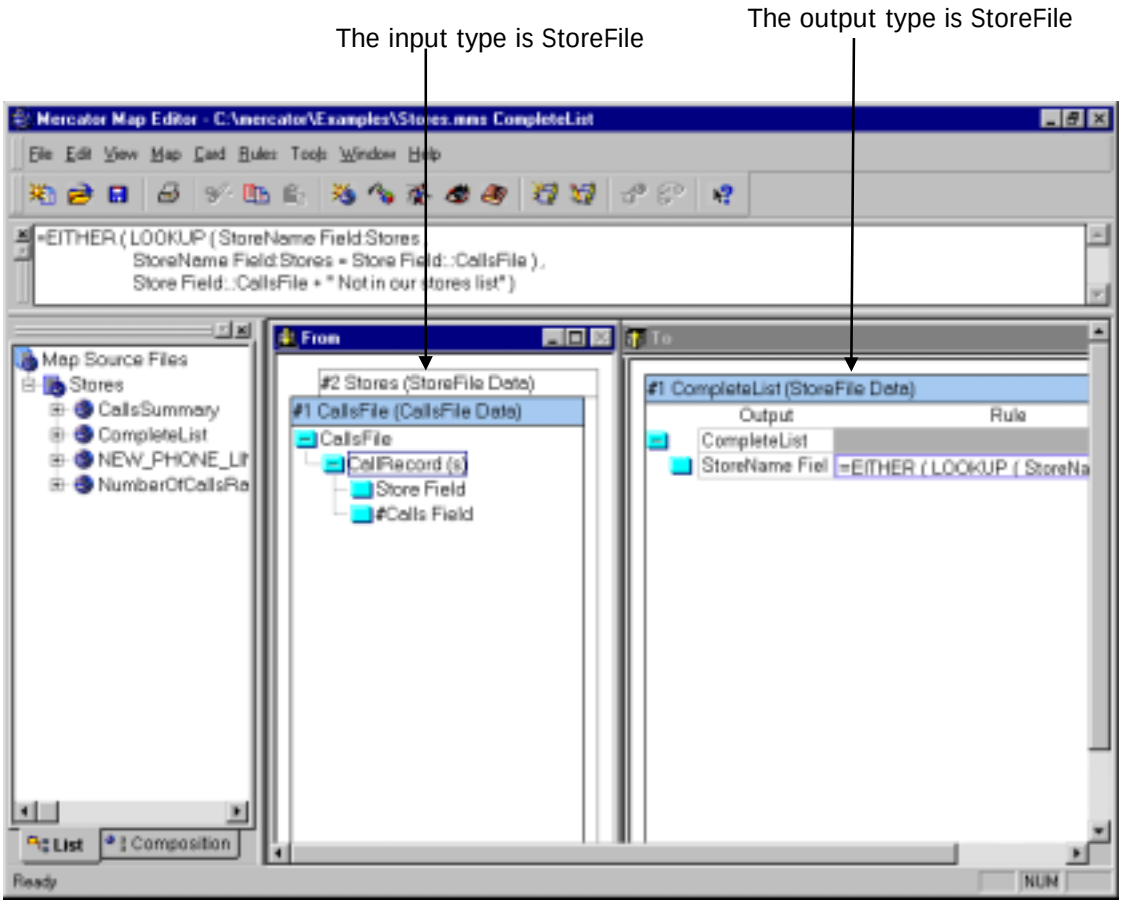
EITHER (Expression, Expression)

The map rule for Store field is:

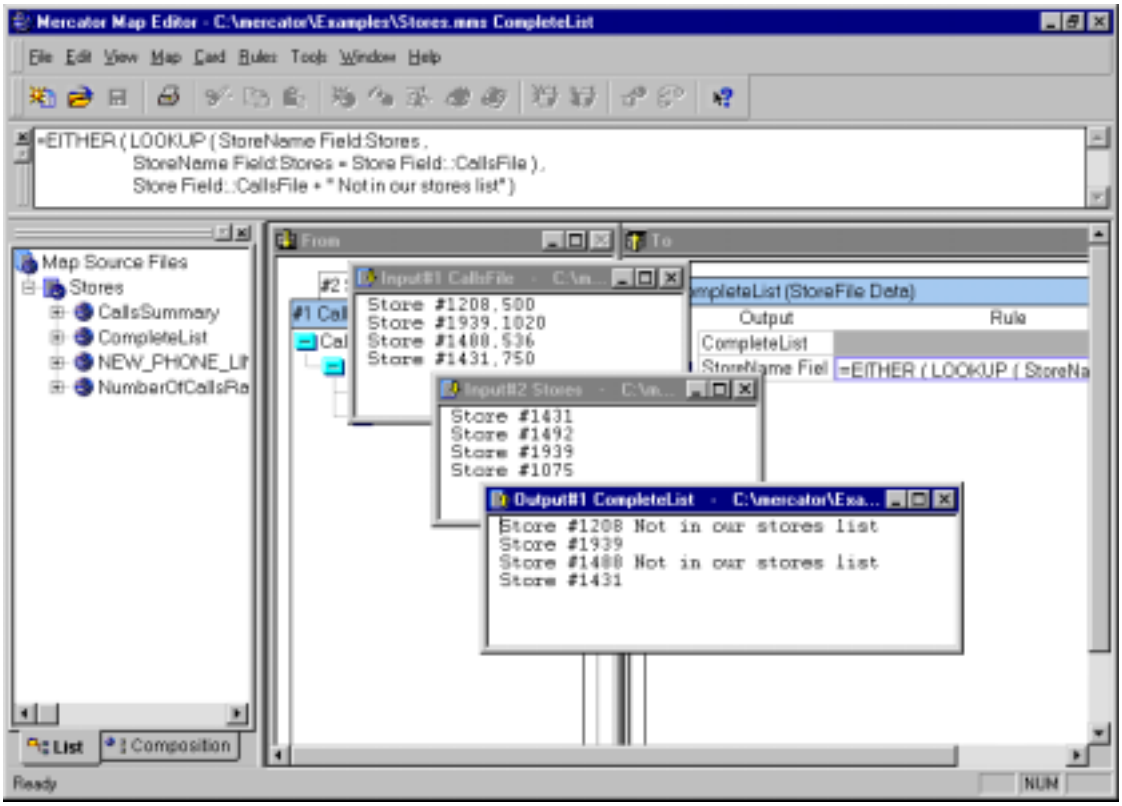
```
= EITHER (LOOKUP (StoreName Field:Stores,
```

```
StoreName Field:Stores = Store Field:.:CallsFile),
Store Field:.:CallsFile + " Not in our stores list")
```

The map looks like this:



In the output data, the stores that are not in the Stores file are indicated as such.



Case 4 – Using Nested IF Functions

In this example, you use nested IF functions to determine what data should be generated.

What You Want to Do

You want to generate a file that tells whether the number of calls at a store is low, moderate, or high. The Calls file is the input.

How to Do It

Use the type CallsFile as the input file, and StoreFile as the output file.

Create a map that maps the Calls data to the StoreFile. To map the rating of the number of calls—not many calls, a moderate amount, or a high number of calls—use nested IF functions.

Files Used in Case 4

This example uses map source file, the type tree, and the input data files created in Case 1.

File	Use
calls.txt	Use this file, created for Case 1, as an input data file.
stores.mtt	This type tree was created in Case 1.
stores.mms	You modify this map source file, which was created in Case 1 and modified in Cases 2 and 3, to include the map for this example.
ratings.txt	This output file is created by running the map.

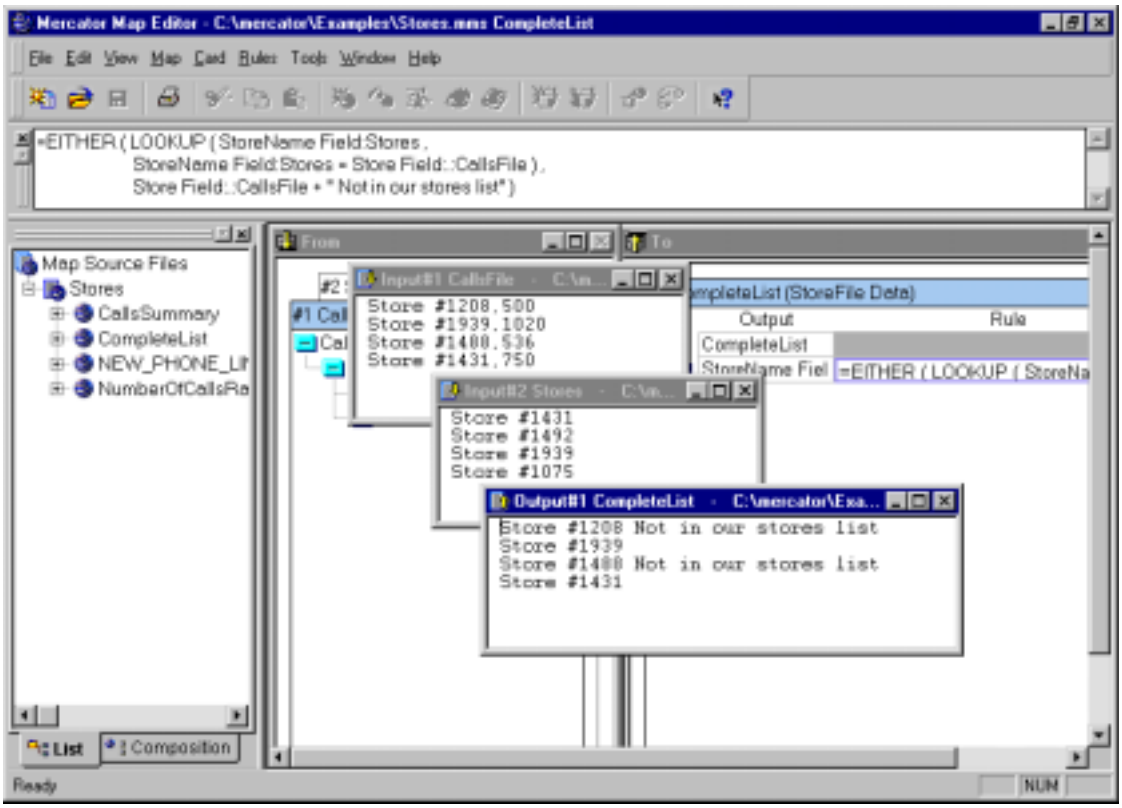
Using the Type Editor

Your output data should look like this:

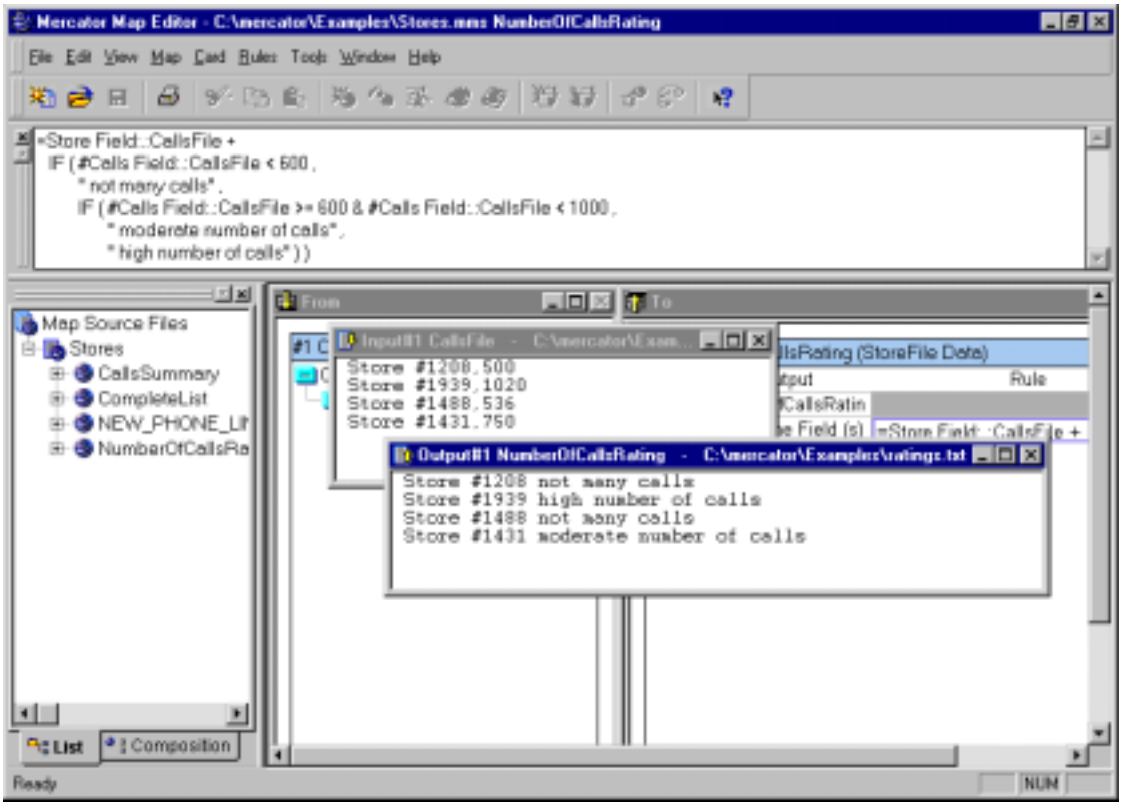
```
Store #1208 not many calls
Store #1939 high number of calls
Store #1488 not many calls
Store #1431 moderate amount of calls
```

Using the Map Editor

In the map enter a rule for StoreName field. Map the Store field from the input, and then use nested IF functions to indicate “not many calls,” “moderate amount of calls,” or “high number of calls.”


Mercator Map Editor - C:\mercator\Examples\Stores.mms CompleteList
File Edit View Map Card Rules Tools Window Help
=EITHER (LOOKUP (StoreName Field:Stores ,
StoreName Field:Stores = Store Field::CallsFile),
Store Field::CallsFile + " Not in our stores list")
Map Source Files
Stores
CallsSummary
CompleteList
NEW_PHONE_LI
NumberOfCallsRe
List
Composition
From
To
Store #1208,500
Store #1939,1020
Store #1400,536
Store #1431,750
Store #1431
Store #1492
Store #1939
Store #1075
Output#1 CompleteList - C:\mercator\Exa...
Store #1208 Not in our stores list
Store #1939
Store #1400 Not in our stores list
Store #1431
Output
Rule
CompleteList
=EITHER (LOOKUP (StoreNa
Ready
NUM

In the output, you can see that each store has its appropriate rating.



Chapter 16 – Incrementing Output Data

This chapter shows three methods of incrementing the position of an output object. Each method uses one of the following—the INDEX function, the COUNT function, and the index [LAST].

What You Want to Do

You have a Header file and a Detail file, which you want to map to a PO file. In the header of each PO in the output, there is a field whose value increments by one for each PO. In the first PO, it has the value 1, in the second PO, it has the value 2, and so on.

Case 1 – Using the INDEX Function

In the first example, you use the INDEX function to index the output.

How to Do It

In the type tree, add the IndexOfPO field to the definition of the header.

In the map rule for PO, use either the INDEX function, the COUNT function, or the index [LAST] to increment the output.

Files Used in Case 1

This example uses input data files created in Case 1 of Chapter 10. The type tree file and map source file must be created for this example.

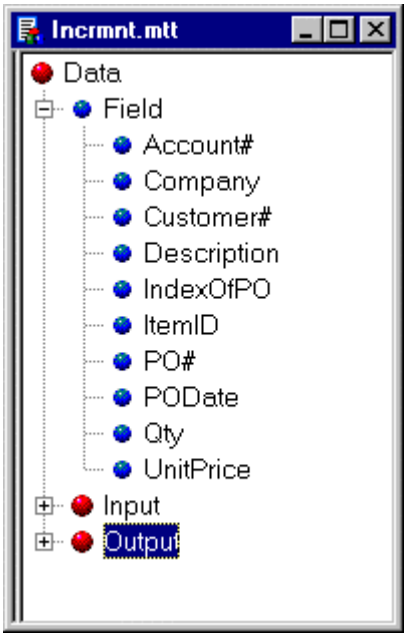
File	Use
header.txt	Use this file, created for Case 1 of Chapter 10, as an input data file.
detail.txt	Use this file, created for Case 1 of Chapter 10, as an input data file.
incrmnt.mtt	Open the type tree twofiles.mtt, created in Chapter 10, and save it as incrmnt.mtt.
incrmnt.mms	You create this map source file for use in the examples in this chapter.

incrmnt.out This output file is created by running the map.

Using the Type Editor

Open the type tree twofiles.mtt, created in Chapter 10, and save it as incrmnt.mtt. Then, add a field called IndexOfPO, and define it as an integer. Make it the last component of Header.

The field types and components in your type tree should look like the following:

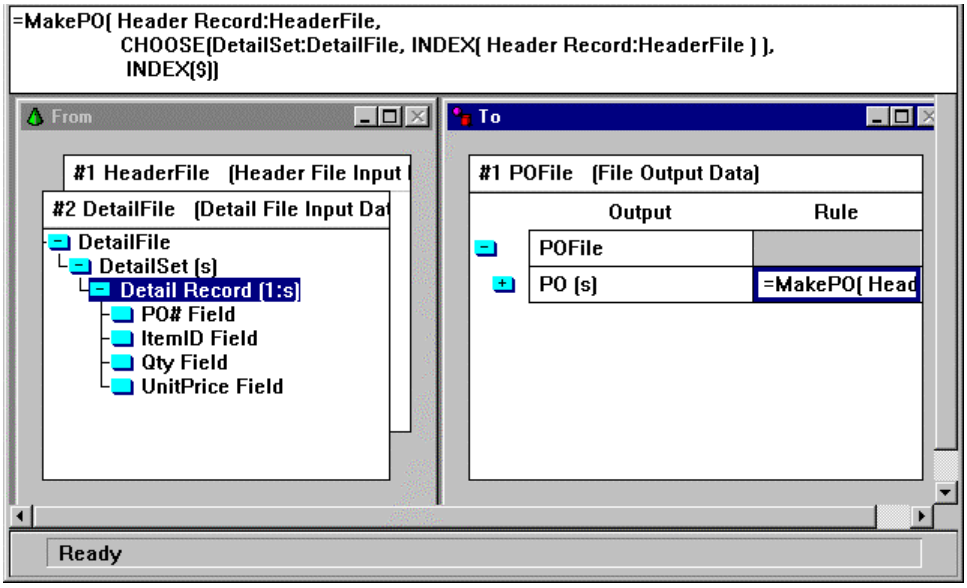


Using the Map Editor

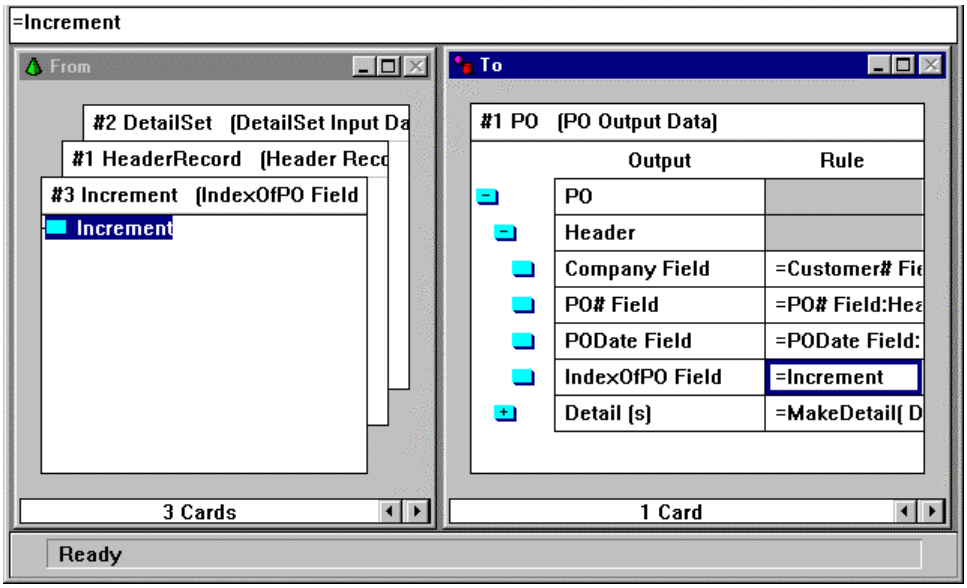
The INDEX function returns the position of an object in a series. For example, the first object has the index of 1. The second object has the index of 2. The syntax of the INDEX function is:

INDEX (Object whose index you want)

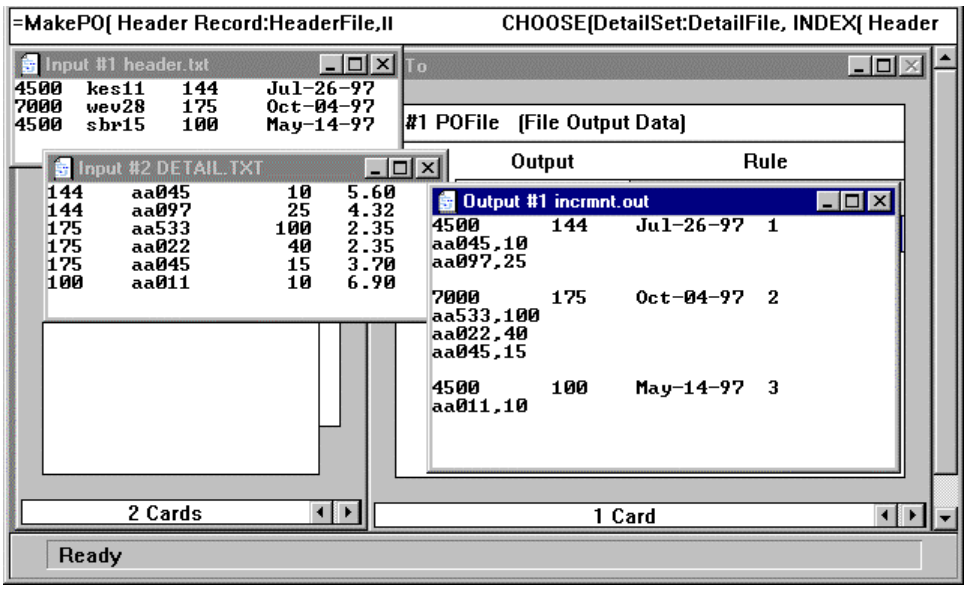
In the map rule for the output PO, you reference the functional map MakePO. There are three arguments to this map. The first two are the same as they are in Chapter 10. The first is the Header Record, the second is the DetailSet that has the same index as the Header Record. The third argument is the index of the current output, that is, the index of PO. The index of the first PO is 1, the index of the second PO is 2, and so on. To indicate the output PO, use the shorthand notation “$”.



In the functional map MakePO, the third input card is mapped to the IndexOfPO output.



In the output file, the IndexOfPO increments by one for each PO.



Case 2 – Using the COUNT Function

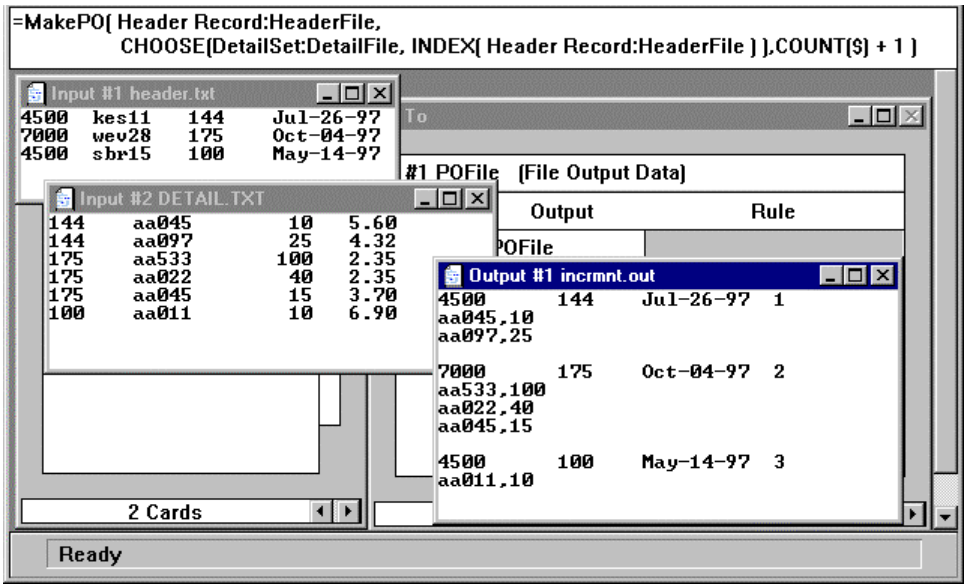
You can use the COUNT function instead of the INDEX function. If you count the number of POs that have already been created in the output, you have to add 1, to include the current one. When Mercator is creating the first PO, the count of PO is 0, so you add 1 to make it 1. When Mercator is creating the second PO, the count of PO is 1, so you add 1 to make it 2, and so on.

Files Used in Case 2

Use the same type tree, map source file and input data files as those used in Case 1.

Using the Map Editor

The only difference between this case and the previous case is the third argument of the functional map MakePO, which is COUNT($) + 1.



Case 3 – Using the Index [LAST]

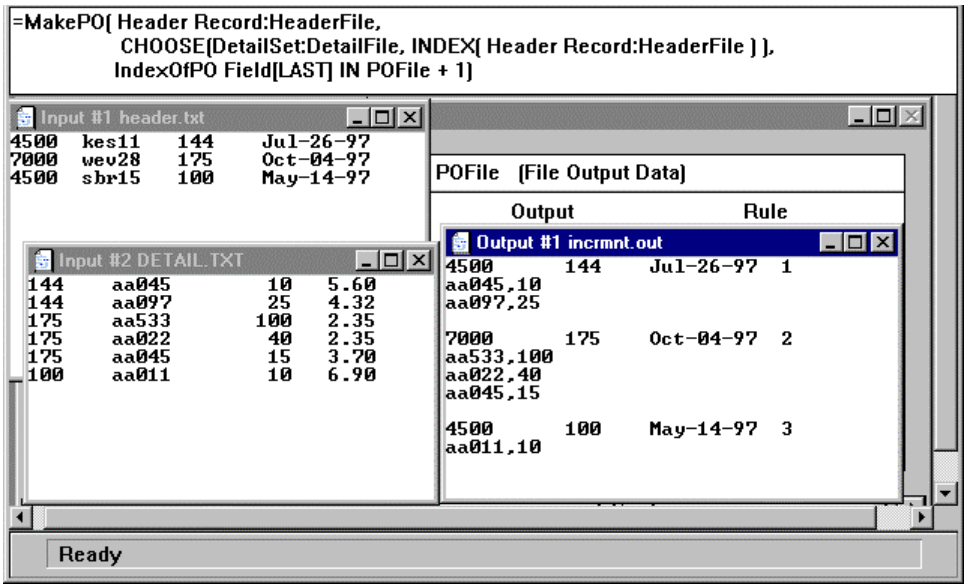
Another way to increment data is to take the index of the last IndexOfPO in the output, and add 1. When Mercator is creating the first PO, there is no IndexOfPO yet, so the index of the last one is 0. You add 1 to make it 1. When Mercator is creating the second PO, the index of the last IndexOfPO is 1. When you add 1, it is 2.

Files Used in Case 3

The files used are the same as those used in Case 2.

Using the Map Editor

The output IndexOfPO is nested within the PO. When referring to IndexOfPO, you do not have to enter the full object name. You can refer to the field in the entire card. The third argument of the functional map MakePO is IndexOfPO[LAST] IN POFile+ 1.


=MakePO(Header Record:HeaderFile,
CHOOSE(DetailSet:DetailFile, INDEX(Header Record:HeaderFile)),
IndexOfPO Field[LAST] IN POFile + 1)
Input #1 header.txt
4500 kes11 144 Jul-26-97
7000 wev28 175 Oct-04-97
4500 sbr15 100 May-14-97
Input #2 DETAIL.TXT
144 aa045 10 5.60
144 aa097 25 4.32
175 aa533 100 2.35
175 aa022 40 2.35
175 aa045 15 3.70
100 aa011 10 6.90
POFile (File Output Data)
Output
Rule
Output #1 incrmnt.out
4500 144 Jul-26-97 1
aa045,10
aa097,25
7000 175 Oct-04-97 2
aa533,100
aa022,40
aa045,15
4500 100 May-14-97 3
aa011,10
Ready

Chapter 17 – Retrieving Information from Other Applications

Using the EXIT Function

The EXIT function sends a text or bytestream Item to a function defined outside of Mercator, and returns a text or bytestream Item. The EXIT function is platform-specific. For the Windows engine, the EXIT function requires the first argument to be the name of a DLL, the second to be the name of a function in the DLL, and the third to be the input Item. The example that follows uses AnsiUpper—a function that receives and returns only a text Item.

It is important to remember that the EXIT function requires that the DLL function you call have only one argument. It returns a text Item. The DLL must be written to specific requirements. For an explanation of these requirements, see the description of the EXIT function in the *Functions & Expressions Reference Guide*.

Files Used in this Example

In this example you use a DLL that comes with Windows (either user.dll or user32.dll), and a test DLL (mydll.dll or mydll32.dll). So, if you have a 16-bit environment (Windows 3.1), you use mydll.dll and user.dll. If you have a 32-bit environment (Windows 95 or NT), you use mydll32.dll and user32.dll.

The files for this example can be found in your mercator\examples\general\exit directory. User.dll or user32.dll is in your system directory (or folder).

File	Use
mydll.dll or mydll32.dll	This DLL, which is the first argument of one EXIT function, converts the text to alternating upper and lowercase letters.
user.dll or user32.dll	This DLL, which is the first argument of the other EXIT function, converts text to all uppercase characters.
exit.mtt	This type tree file defines the output data of the map.
exit.mms	This map source file contains the map that uses the EXIT function to generate data.
exittest.out	This output file is generated by running the

map.

Understanding the Map

The map example has one output card, and no input cards. The output has two text Item components. The meaning for the EXIT function in a Windows environment is:

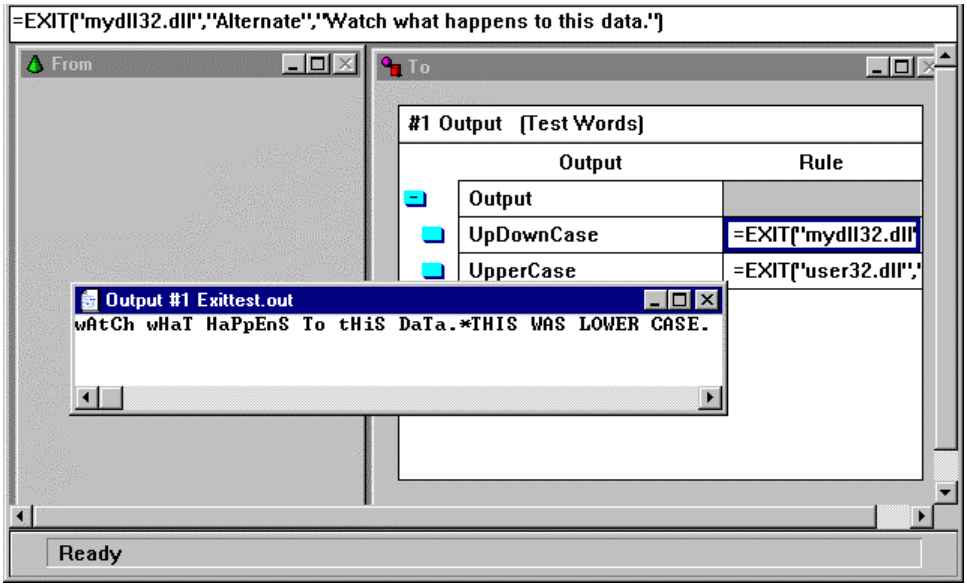
EXIT*(application_name, application_function_name, input_to_the_function)*

Note The map to run for the 16-bit Windows environment is Exit_Example. For the 32-bit Windows environment, the map is Exit_Example32. If you attempt to run the map for the other environment, you get errors.

For the 32-bit environment, the map rules on the two components are:

```
=EXIT(“mydll32.dll”, “Alternate”, “Watch what happens to this
data.”)

=EXIT(“user32.dll”, “AnsiUpper”, “this was lower case.”)
```

The rule on the first component calls mydll32.dll, which converts the text to alternating upper and lowercase letters. The rule on the second component calls user32.dll, which converts the text to uppercase.



Using the DDEQUERY Function

The DDEQUERY function allows you to interface to other DDE enabled programs (Excel, Trading Partner PC, Word for Windows, etc.). When you use DDEQUERY, you tell Mercator to ask a particular program to provide information from a specific area of a selected file. Each program with this ability requires that you ask for the information in a specific way.

In general, the DDEQUERY function uses these arguments:

```
DDEQUERY (program, topic, item )
```

For example, a DDEQUERY function to retrieve data from Excel may look like this:

```
DDEQUERY (“excel”,”[MyData.xls]Sheet1”,”R1C2:R2C5”)
```

A request to Trading Partner PC could look like this:

```
DDEQUERY (“tppc”,”PartnerX”,”BGyourEDICode”)
```

In both cases you supply the name of the program that you want to communicate with, a topic, and an item. The topic, which is the second argument, is the topic of the “conversation”. In the Excel example you see that the topic is some file, MyData.xls and, more specifically, sheet 1 of that file.

Note The executable program file *must* be in your DOS path. You *cannot* enter the full path name of the program for the first argument of the DDEQUERY function.

The software you are retrieving information from determines the syntax of the second argument. It should be found in the documentation for that software.

The third argument is the specific information that you want to obtain. You are asking Excel to provide all the data in the block R1C2 to R2C5 from Sheet1 of MyData.xls. R1C2 stands for Row 1, Column 2, and R2C5 stands for Row 2, Column5, or the block B1:E2. From Trading Partner PC (the program tppc), you are requesting the BG EDI Code from Partner X.

DDEQUERY is a powerful function—enabling you to retrieve exactly the data you need.

Note The application you interact with must be running while your map is running. In addition, if you use a DDEQUERY to a particular file in that application, *that file must be open*. If a DDEQUERY function is encountered when running a map, and these conditions are not met, then another instance of that program may be started.

For example, if you have Excel open, but with a worksheet other than MyData.xls, the DDEQUERY may start Excel again, resulting in two Excel sessions going at the same time. You could quickly run out of memory if you continually attempt to run the map.

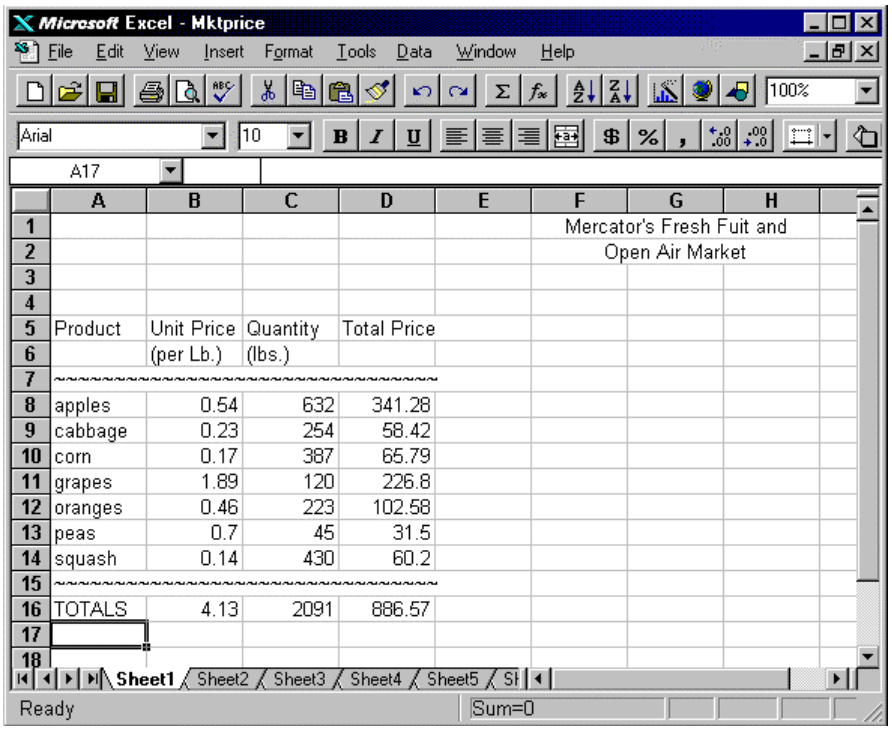
Files Used in this Example

The files for this example include the Excel spreadsheet mktprice.xls. These files are in your mercator\examples\general\ddequery directory (folder in Windows 95).

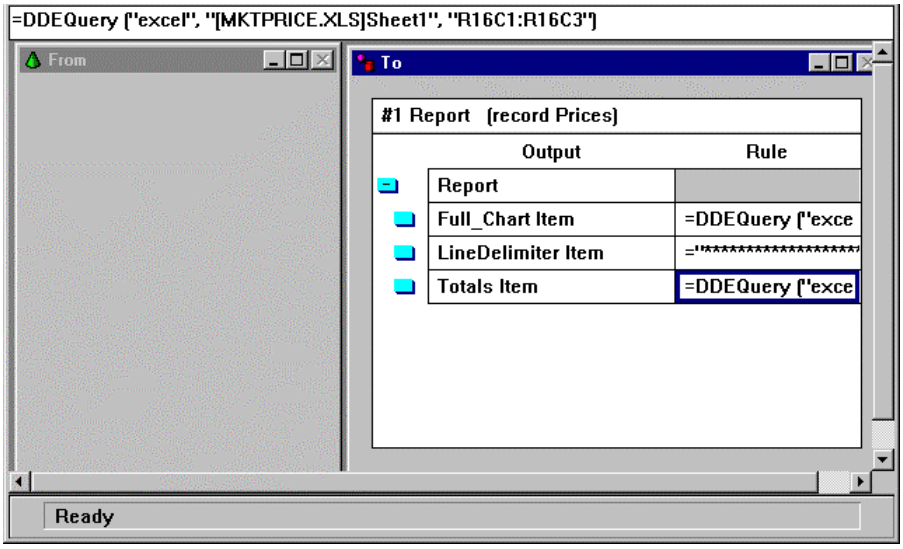
File	Use
mktprice.xls	Open this spreadsheet in Excel when you run the map. This spreadsheet file contains data that the DDEQUERY function requests.
invtory.mtt	This type tree file defines the output data of the map.
inv2txt.mms	This map source file contains a map that uses DDEQUERY in a map rule. When the map is run, it uses input retrieved from the Excel spreadsheet to generate output.
pricelst.tmp	This output file is created by running the map.

Understanding the Map

Suppose that your company, Mercator's Fresh Fruit and Open Air Market, keeps the inventory in an Excel spreadsheet. In the spreadsheet are each product name, its price per pound, the quantity of each, the total pounds in stock, and the total market value. You need to send some of this data with other monthly reports to the main office. The problem is to transform the spreadsheet data into a text based format so that you can transmit it with the rest of your information in a predefined format. Here is the Excel spreadsheet:



You want to map the first three columns of the spreadsheet—the Product, Unit Price and the Quantity. You want to map the main data to Full_Chart Item, and the TOTALS row to Totals Item. You use the DDEQUERY function to retrieve the required data from the rows and columns from the spreadsheet.



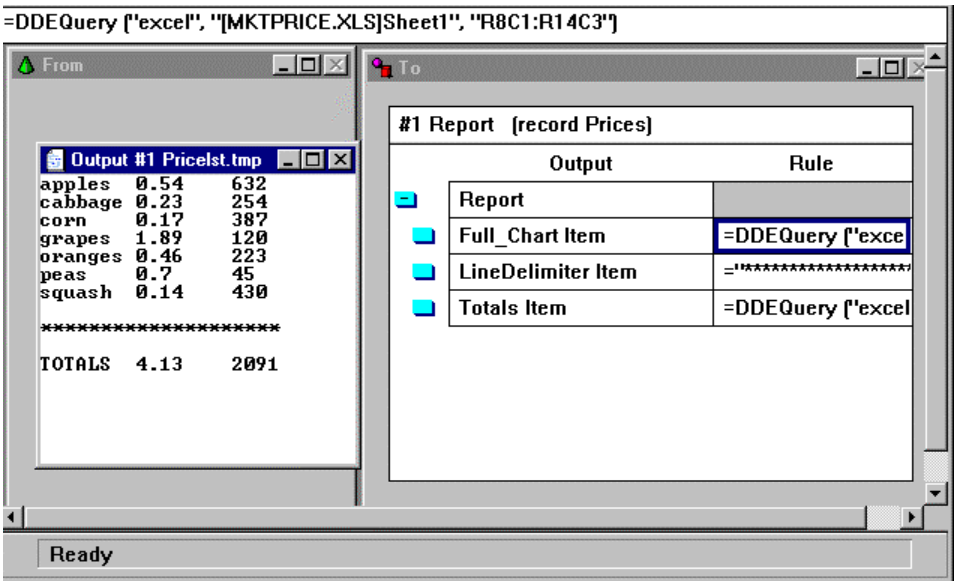
The DDEQUERY for Full_Chart Item retrieves the block of data from row 8, column 1 through row 14, column 3, (in Excel's default nomenclature, block A8:C14). The map rule is:

```
=DDEQuery ("excel", "[MKTPRICE.XLS]Sheet1", "R8C1:R14C3")
```

The DDEQUERY for Totals Item retrieves the block of data from row 16, column 1 through row 16, column 3, (block A16:C16). The map rule is:

```
=DDEQuery ("excel", "[MKTPRICE.XLS]Sheet1", "R16C1:R16C3")
```

When you run the map, you get the expected data:



Note When you run the map, Excel and the spreadsheet file you are retrieving from must be open.

Chapter 18 – Functions that Operate on Text Data

This example uses the text functions FIND, LEFT, MID, and RIGHT.

What You Want to Do

You work for a health care company that has a file of employee names—doctors and nurses. You want to retrieve only the doctors in this employee file, and generate a new file that lists the doctors and the hospital where each works.

A doctor is indicated by the "M.D." after the name. The hospital is indicated by the value in the last byte of one of the fields in the employee record.

How to Do It

Define the input and output file in a type tree.

Create a map that maps the input file to the output file. To map the data, use the functions FIND, LEFT, MID, and RIGHT.

Files used in this Example

The following table lists the files used in this example.

File	Use
hospital.txt	You create this input data file to use as input.
hospital.mtt	You create this type tree to define the input and output.
hospital.mms	You create this map source file.
output.txt	This output file is created by running the map.

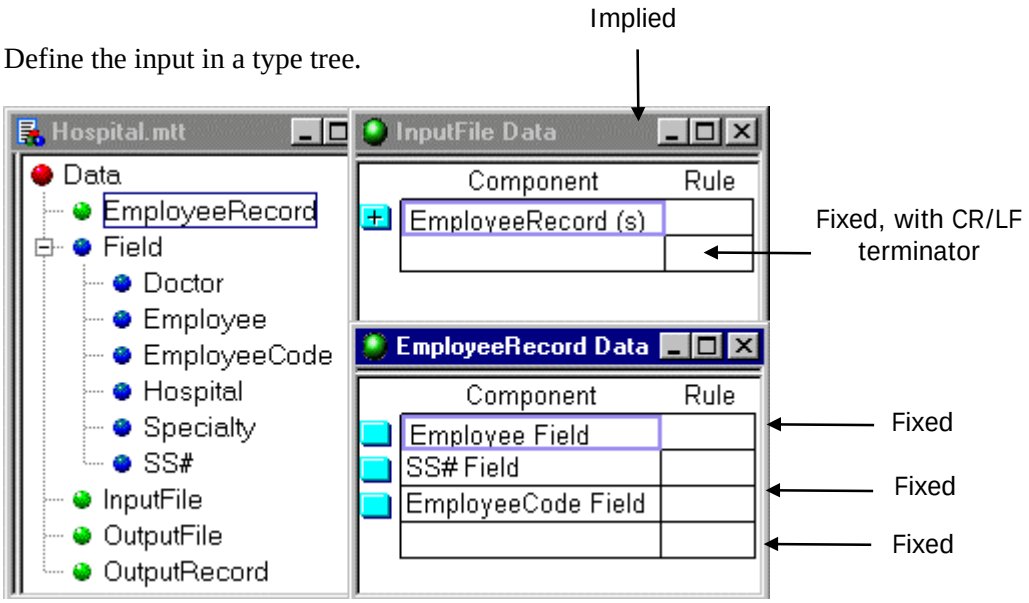
Using the Type Editor

The input data looks like this:

```
Samantha Allibaster, M.D.   111-24-4291   131jja
Kirk Benett, M.D.           314-42-9595   842keu
Valerie Johnson, R.N.       423-11-9188   359iru
Lyle Ropes, R.N.            239-54-5700   879nla
Foster McMann, M.D.         403-67-3920   514ipu
```

Each employee record contains the employee name, social security number, and an employee code. The last digit of the employee code indicates the hospital where the person is employed.

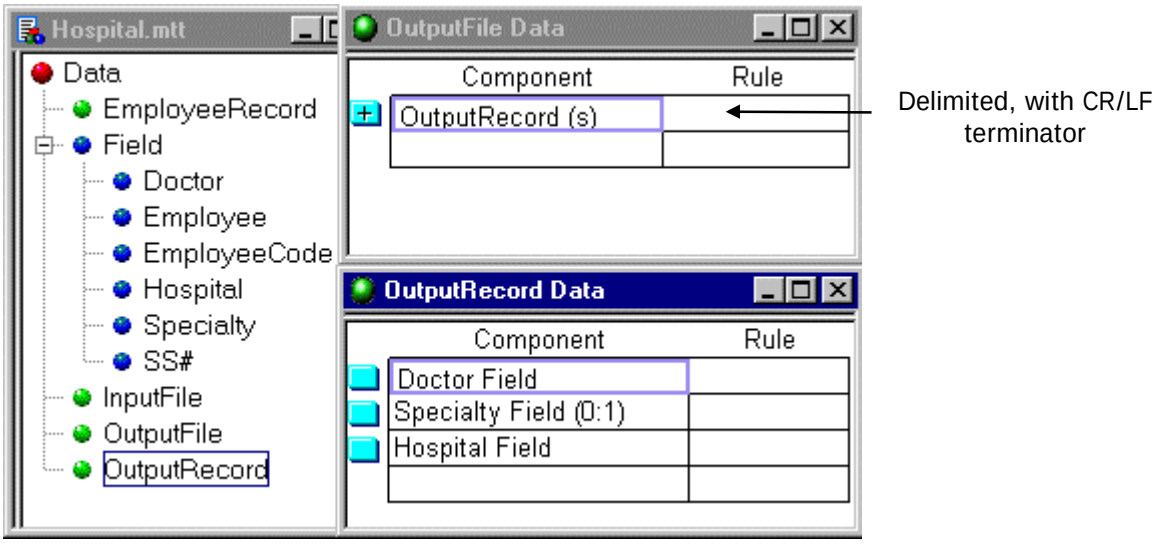
Define the input in a type tree.



You want the output data to look like this:

```
Dr. Samantha Allibaster/Internist/Andrews Hospital
Dr. Kirk Benett/Surgeon/Unitarian Hospital
Dr. Foster McMann//Unitarian Hospital
```

Define the output data in a type tree.



Using the Map Editor

In the executable map, the rule on OutputRecord references the functional map MAKE_RECORD. The only employee records you want to map are the ones that have “M.D.” in the name. You can use the FIND function.

The FIND function has at least two input arguments. It has an optional third argument. The first argument is the text you want to find. The second argument is the text object you’re looking at. The optional third argument specifies the position at which to begin the search. The leftmost byte has position 1.

The FIND function returns the first position at which the given text occurs. If the text is not found, then FIND returns 0. The meaning of the FIND function is:

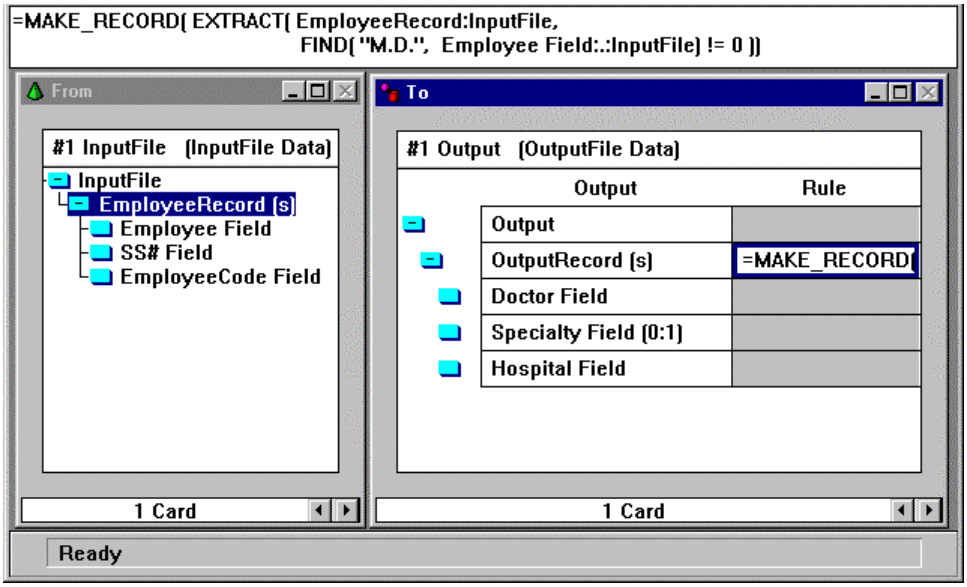
FIND (Text you want to find, text item to look at, [Where to begin the search])

You want to map the employee records for which the FIND of “M.D.” does not return 0.

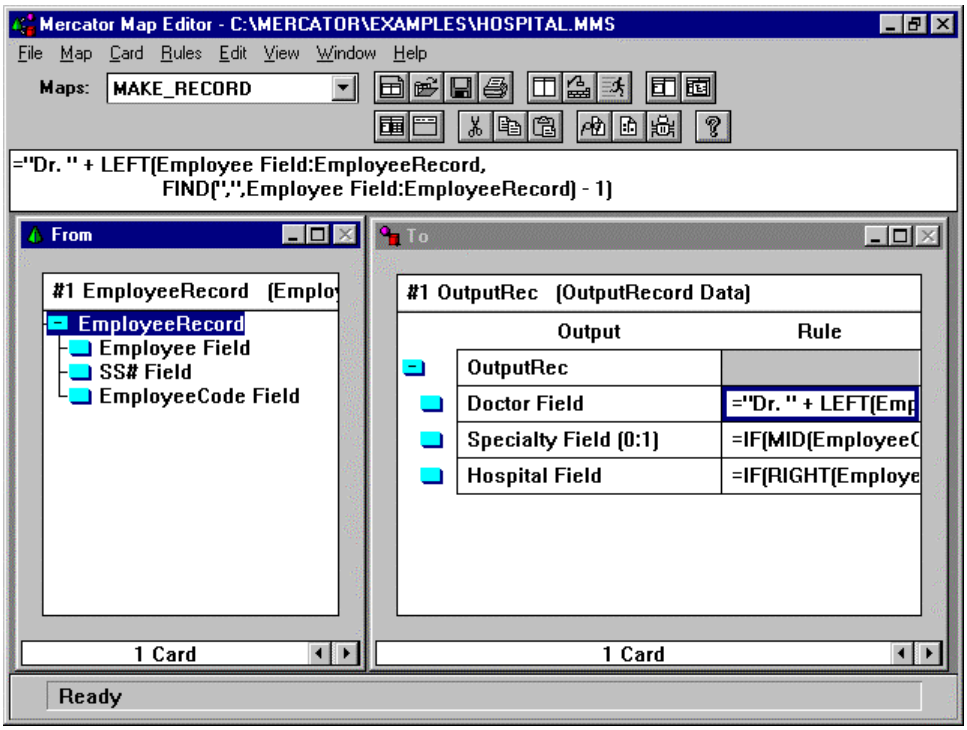
The map rule for OutputRecord is:

```
= MAKE_RECORD (EXTRACT (EmployeeRecord:InputFile,
FIND ("M.D.", Employee Field:.:InputFile) != 0 ))
```

The executable map looks like this:



The functional map MAKE_RECORD looks like this:



To map the Doctor field, use the LEFT, and FIND functions to retrieve everything up to the comma in the Employee field.

The LEFT function retrieves a certain number of bytes from a text Item, beginning at the leftmost byte of that Item. The second argument specifies how many bytes to retrieve. The meaning of the LEFT function is:

LEFT (Text item, Number of bytes to retrieve)

The map rule for Doctor field is:

```
= "Dr. " + LEFT (Employee Field:EmployeeRecord,
                 FIND ("," , Employee Field:EmployeeRecord) - 1)
```

To map the Specialty field, use the MID function. The MID function retrieves a certain number of bytes from a text Item, beginning at the position specified in argument #2. The leftmost byte has position 1. The third argument specifies how many bytes to retrieve. The meaning of the MID function is:

MID (Text item, Starting position, Number of bytes to retrieve)

Use the MID function to look at the fourth byte of the EmployeeCode. The value in this field indicates the specialty of the employee. If the value is “j,” the doctor is an internist. If the value is “k,” the doctor is a surgeon. Any other value is ignored.

The map rule for Specialty field is:

```
= IF (MID (EmployeeCode Field:EmployeeRecord, 4, 1) = "j", "Internist",
 IF (MID (EmployeeCode Field:EmployeeRecord, 4, 1) = "k", "Surgeon",
 NONE))
```

To map the Hospital field, use the RIGHT function to look at the rightmost byte of the EmployeeCode. If the value is “a”, it’s Andrews Hospital, otherwise it’s Unitarian Hospital.

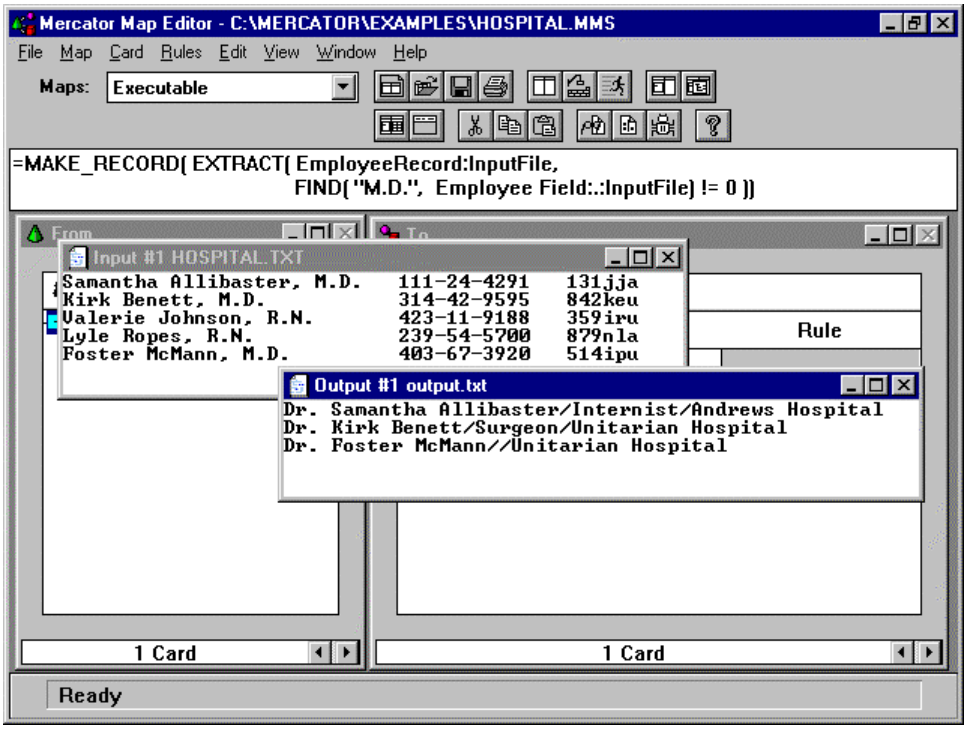
The RIGHT function retrieves a certain number of bytes from a text Item, beginning at the rightmost byte of that Item. The second argument specifies how many bytes to retrieve. The rightmost byte is position 1. The meaning of the RIGHT function is:

RIGHT (Text item, Number of bytes to evaluate)

The map rule for Hospital field is:

```
= IF (RIGHT (EmployeeCode Field:EmployeeRecord, 1) = "a",
            "Andrews Hospital",
            "Unitarian Hospital")
```

Only the doctors were mapped to the output. If the doctor has a specialty, the specialty was mapped.



Index