RTML 101:

The Unofficial Guide to Yahoo! Store® Templates

2nd Edition – Updated for the New Editor

By István Siposs

Copyright and Important Legal Information

Your use of this book means that you have read, understand, and agree to the following terms. The terms are legally binding upon you.

Every reasonable effort has been made to ensure the accuracy of the information presented in this book at the time of its publication. Note, however, that neither the author nor publisher is in charge of your web services provider, nor are they in charge of Yahoo!® stores generally, nor can they control your Yahoo!® store in particular. Consequently, this book may contain passages that are, or that later become, inaccurate. The book probably contains some passages that are not the best solution for your particular needs, too. Therefore, all information presented in this book is provided on an "as is" basis.

Neither the publisher nor the author of this book makes any representations or warranties with respect to the accuracy or completeness of the contents of this book. On the contrary, the author and publisher specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided in this book is not guaranteed or warranted to produce any particular result. Neither the author nor the publisher shall be liable for any loss of profit, money damages, or any other form of relief for problems arising from your use of this book. In other words, even if you believe the information in this book caused something bad to happen, you are not entitled to any remedy from the author or publisher.

All trademarks and product names used in this book are the property of their respective owners. In particular, "Yahoo!" is a registered trademark of Yahoo! Inc., a Delaware corporation headquartered in Sunnyvale, California, USA. Yahoo! Inc. is not the author of this book and this book in no way represents the views or opinions of Yahoo! Inc. or any Yahoo! Inc. personnel or affiliates. Yahoo! Inc.'s mark is used in the title and content of this book only because the subject of this book is otherwise not readily identifiable, especially to the average consumer not expert in web design.

Published by:
Y – Times Publications, L.L.C.
1055 West College Avenue, #227
Santa Rosa, CA 95401
www.ytimes.info

Copyright © 2002-2004 by Y – Times Publications, L.L.C. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the publisher, except for the inclusion of brief quotations in a review.

ISBN 0-9719663-2-X

COPYRIGHT AND IMPORTANT LEGAL INFORMATION I

ABOUT THE AUTHOR X

ACKNOWLEDGEMENTS X

INTRODUCTION XI

Who should read this book .... *xii*

What is RTML? .... *xii*

Things you can do with RTML .... *xiv*

Things you cannot do with RTML .... *xv*

Using customized templates .... *xvii*

PART I. LEARNING RTML 1

THE ADVANCED INTERFACE 2

The Contents List *3*

ID *4*

TYPES 6

Built-in types *7*

Customizing Types *11*

Types of properties *12*

Default property values *16*

Using custom types *17*

Custom types and accessories *20*

Custom types and Database (CSV) Upload *21*

Deleting custom types *22*

TEMPLATES 22

The Template Editor *22*

The Template Editor Toolbar *29*

Built-in templates *32*

Modifying templates *34*

Writing your own templates *38*

Calling templates *45*
PROPERTIES AND VARIABLES 47
Properties *47*
Global variables *48*
Local variables *49*
Naming variables *51*
Nested variables *52*
Parameters revisited *52*
OBJECTS 53
ID revisited *53*
Referring to other objects *54*
Changing context *54*
Up, Next, and Prev *56*
CONSTANTS 59
EXPRESSIONS 63
LOGICAL EXPRESSIONS 63
And *63*
Or *64*
Not *65*
CONTROL STRUCTURES 67
MAKING DECISIONS: CONDITIONALS 67
The IF operator *67*
The WHEN operator *69*
The SWITCH operator *70*
REPEATING ACTIONS: ITERATION 72
For loops *72*
For-Each *75*
For-Each-But *76*
For-Each-Object *77*

Find-One *78*

PART II. WORKING WITH RTML 81

SEQUENCES 82

Elements *82*

Element *83*

Length *83*

Nonempty *84*

Position *84*

Segments *85*

Tokens *86*

Yank *88*

Reverse *88*

Whole-Contents *88*

Make-List and Append *89*

WORKING WITH TEXT 90

Printing text *90*

Concatenating text strings *92*

Working with paragraphs *93*

Working with lines *94*

WORKING WITH NUMBERS 95

Performing calculations *95*

Comparing numbers *96*

Working with prices *97*

WORKING WITH IMAGES 100

Displaying uploaded image files *100*

Creating images from image variables *101*

Creating images from text *102*

Fusing images *103*

Creating hyperlinked images *105*
Determining the size of images: HEIGHT and WIDTH *107*
WORKING WITH COLORS 109
Color *112*
GRAYSCALE, RED, GREEN, and BLUE *112*
WORKING WITH FONTS 114
Font *115*
Text-style *116*
Graphical Fonts *116*
VW-IMG *116*

PART III: RTML REFERENCE 119

<=> 120
< 121
<= 122
> 122
>= 122
ACTION 123
AND 124
APPEND 125
AS-LIST 125
AUCTIONURL 126
BASKET 126
BASKET-MODULE 126
BLUE 126
BODY 127
CALL 129
CAPS 129
CENTER 130

CMP 130
COLOGO 130
COLOR 132
COMMENT 132
ELEMENT 133
ELEMENTS 133
EQUALS 134
EVEN 135
FIND-ONE 135
FONT 136
FONT-WIDTH 137
FOR 138
FOR-EACH 139
FOR-EACH-BUT 139
FOR-EACH-OBJECT 140
FORM 140
FUSE 141
GET-ALL-PATHS-TO 143
GET-PATH-TO 144
GRAB 145
GRAYSCALE 146
GREEN 146
HEAD 146
HEIGHT 147
HEX-ENCODE 147
HRULE 147
IF 148
IMAGE 149
IMAGE-REF 151

INPUT 152
INVENTORY-INFO 154
ITEM 154
ITEM-INVENTORY 154
LENGTH 156
LINEBREAK 156
LINES 156
LIVE 157
LOCAL-LOGO 157
LOG 157
LOOKUP 157
LOWERCASE 158
MAKE-LIST 158
MAXIMUM 158
META 159
MINIMUM 159
MODULE 160
MULTI 161
NOBREAK 162
NONEMPTY 162
NOT 163
NUMBER 163
OBJID-FROM-STRING 164
REVERSE 165
OR 165
ORDER 166
ORDER-FORM 167
PARAGRAPH 167
PARAGRAPHS 168

POSITION 169
PRICE 169
RED 170
RENDER 170
RETURN-WITH 174
SEARCH-FORM 174
SEGMENTS 175
SELECT 175
SET 175
SHIM 176
SHOPPING-BANNER 177
SORT 177
STRCASECMP 178
STRCMP 179
STRING-TRIM 179
SWITCH 179
TABLE 180
TABLE-CELL 182
TABLE-ROW 183
TAG-WHEN 184
TAXSHIP-MODULE 185
TAXSHIP-ORDER-MODULE 187
TEXT 189
TEXT-STYLE 189
TEXTAREA 190
TITLE 193
TO 193
TOKENS 193
VALUE 195

WHEN 196
WHOLE-CONTENTS 197
WIDTH 197
WITH-LINK 198
WITH-OBJECT 199
WITH= 199
WORDBREAK 200
YANK 201
YFUNCTION 201

APPENDICES 203

APPENDIX A: RTML RESOURCES 204
APPENDIX B: YAHOO! STORE® STANDARD GRAPHICAL FONTS 205

INDEX 210

About the author

István Siposs is an independent computer consultant. He is a Yahoo! Store® designer registered with Yahoo! ®'s Designer Referral Program. He has been a software developer since the early nineties, has been developing web sites since 1993 and Yahoo! Store® sites since 1999. He is a recognized Yahoo! Store® expert and a frequent contributor to the Yahoo! Store Forums at http://www.ystoreforums.com. He has a degree in Computer Science from the University of California at Berkeley. His home page is available at www.siposs.com and his e-mail address is istvan@siposs.com.

Acknowledgements

I would like to thank everyone who assisted me in any way in writing this book. Specifically, but in no particular order: Richard Bidleman and Jon Richards for editing; Keith Enloe of Ydesigns.com for making sure the book made sense technically and Stephen Gilmartin of Handrake Development (handrake.com) for indispensable legal advice.

Also: András Vida for introducing me to computers; Richard Bidleman for showing me that computers can be more than a hobby; and my wife Anikó, my daughter Réka, and my son, István, for constantly reminding me that there is more to life than just computers…

István Siposs

Introduction

Yahoo! Store® is one of the largest and fastest growing storefront systems on the Internet. Originally, the Yahoo! Store® platform was developed by a company called Viaweb®. In 1998, Yahoo!® acquired Viaweb® with its membership of about 1,000 merchants. Today, the community of Yahoo! Store® merchants includes over 16,000 storefronts.

The strengths of the Yahoo! Store® platform are manifold:

1. It is affordable.
2. It provides store owners a complete management interface to control all aspects of their store including inventory management, statistics and marketing.
3. It means inclusion in Yahoo! Shopping®, one of the largest online malls on the Internet.
4. It is easy to use yet versatile enough to enable a merchant to create even the most highly customized storefronts.

This last feature, the ability to create stores with a unique look through custom template design, can be implemented using Yahoo! Store®'s proprietary design language, RTML. While the availability of RTML is advertised right on the welcome page of Yahoo! Store®, RTML is probably the least documented feature of the service and to make things more complicated, Yahoo!® provides no technical assistance for custom templates and RTML—which, by the way, is completely understandable considering that, as you will soon learn, RTML is a programming language, and providing technical assistance to customers to debug their code would be an immense task.

The Yahoo! Store® online user's guide includes a very general overview of RTML. There is also an RTML command reference—not much more than an

alphabetical listing of the various RTML commands—hidden deep within the Advanced Editor Interface (under the “Controls” section, for those interested). These two resources, however, represent all the documentation provided by Yahoo!®. Outside of Yahoo!®, there is a Yahoo! Store® RTML Club in Yahoo! Clubs and there are perhaps a few other web sites devoted to the subject (see Appendix A: RTML Resources), but there are no books or any other printed material available. Those who are interested in creating truly custom-designed storefronts are then left to their own devices to learn the ins and outs of RTML. We hope that this book will fill this void.

Who should read this book

This book was written for those interested in learning or using Yahoo! Store®’s proprietary web design language, RTML. Since RTML is ultimately a web design language, you should have some understanding of HTML. You don’t; however, need to know JavaScript, CGI, or any other so-called server-side scripting languages to use the information in this book.

Because RTML is a programming language, it is highly recommended, although not required, that you have some programming experience. Any basic programming language course from college will do.

Anyone who has some moderate experience with Yahoo! Store® and who has felt his or her hands were tied by the Yahoo! Store® design interface, will find this book useful. After reading this book, you'll have the necessary skills to create a store that has your own personal look and feel.

What is RTML?

There are a number of anecdotal explanations of what RTML stands for. According to one such explanation, RTML is *Rob’s Template Mark-up Lan-*

guage, where Rob is one of the original developers of Viaweb. Another, more technical-sounding explanation says that RTML stands for *Recursive Textual Markup Language*. It is a **programming language used to generate or describe static web pages**. RTML has all the elements of a programming language including:

1. Control flow
2. Variables
3. Iteration or loops
4. Conditional execution
5. Subroutine and function calls

For those interested, RTML is loosely based on LISP. For those of you who know LISP, you will appreciate the fact that unlike LISP, RTML does not use the forest of parentheses in its syntax.

Each line of RTML is an **expression**. An expression consists of either some value (variable, or constant) or an **operator** and zero or more **arguments**. A collection of RTML expressions is called a **template**. You could think of templates as programs, sub-routines, or functions. They all return some value and can optionally have the "side effect" of generating some output, i.e. creating HTML text.

While its name resembles HTML (Hypertext Markup Language), RTML is very different from HTML. Although there are a few RTML operators that map directly to some equivalent HTML tags, the similarity pretty much ends there. HTML is a collection of tags to describe a static web page. RTML is a language that can be used to create complex programs that generate web pages or entire web sites. RTML creates HTML as its output.

The real purpose of RTML is to provide a bridge between content and presentation: between the inventory of your store and the HTML pages that display your inventory. This is a great concept for the following reasons:
You don't have to know…

- anything about databases,
- how your inventory is stored within Yahoo! Store®
- how to write pure HTML.

Using RTML, you'll be able to take what's in your store and present it to the public as one or more web pages!

Things you can do with RTML

In short, with RTML you can do anything you could not do in the Store Editor. With RTML, you gain full control over the design of your store. You can modify how the standard page elements are laid out or function. You can, for instance, leave the content buttons (those that correspond to the sections of your site) in the left navigation bar, and move the other buttons (info, privacy policy, shopping cart, etc.) to the top; or include a search box with a button below the left navigation bar; or put banners there.

You can also expand on the existing functionality of your store. You could create, for example, a "Bestseller" list similarly to the built-in "Specials" feature where bestsellers could be items you would show along the right margin of your home page (see www.notetools.com).

Or, at the other extremes, you could create completely custom designed stores that are not even based on any of the built-in templates (see http://store.yahoo.com/belmont0724 or http://www.ytimes.info).

Figure 1 - Custom Templates

Things you cannot do with RTML

While RTML gives you a lot of freedom when it comes to the design of your store, there are things RTML cannot do. To understand what you can and cannot do with RTML, you should know that **RTML is neither what is called a client-side, nor a server-side scripting language**.

Client-side scripting languages such as JavaScript and VBScript are commonly used to validate user input on forms or to generate dynamic content on web pages. Client-side scripting languages require a "client", or web browser, that understands those languages otherwise they cannot function.

Sever-side scripts execute on the web server whenever a web page that uses server-side scripting is requested. They are commonly used to validate user input, to generate dynamic web pages, and to query or work with databases. Some of the most common server-side scripting languages are PERL, ASP, PHP, and CGI.

RTML is a strange breed. It is definitely not a client-side scripting language (have you seen a web browser with RTML support?) It does run on the web

server (on Yahoo!®'s store servers) but you wouldn't call it a server-side scripting language. The main reason for this lies with *when* an RTML template (or program) is executed. While server-side scripts execute each time, they are requested from the web server (each time you load a web page whose url ends with .asp, .php, .pl, .exe, but not in .html or .htm), **RTML is executed only once**, when you generate or publish your store. Once your store is published, no matter how many times your store's pages are brought up in web browsers all over the world, the RTML templates that generated your store do absolutely nothing. Why? Remember, RTML is for generating static HTML pages. The emphasis is on the word *static*.

In addition, there are certain things that are shielded from RTML within Yahoo! Store®. The operation or presentation of the shopping cart and the checkout pages, for instance, are beyond the reach of RTML. Furthermore, you cannot change how the built-in features—such as shipping and tax calculation—of the store function.

So, what are the things you cannot use RTML for? Anything you would use either a server-side or a client-side scripting language. The following is a partial list of the most frequently requested tasks that **cannot** be accomplished using RTML:

RTML cannot be used for:

- User input validation
- Database manipulation
- Building dynamic content (pages that change based on user input)
- Manipulating shipping and tax calculation rules
- Changing the way the shopping cart or the checkout pages function or the way they look
- Manipulating "cookies" (if you don't know what "cookies" are, that's fine as you won't be using them in RTML)

Using customized templates

RTML templates do absolutely nothing by themselves. In fact, you cannot even execute a template by typing its name on a command line or clicking it in some list. The only way to "execute" a template is to call it as part of the "description" of a web page within your Yahoo! Store®.

Each page in a Yahoo! Store® is based on some top-level template that "tells" how that particular page should be rendered in a web browser. This is the "description" of that web page. The easiest way, therefore, to "execute" or "run" a template is to create an item in your store and use your template as the top-level template of that page. To do so, change the *Template* property of your page

from a built-in template to a custom template. Once you do this, whenever you bring up the page in your web browser you will see the output of the custom template.

The following is a very broad outline of the way to create custom templates:

1. Create a custom template or modify an existing one.
2. Edit the page you want to customize, and change its *Template* property to reflect the name of your customized template.
3. Update the page.
4. Publish your store.

From this list, note that if you create custom templates for a store that already contains many pages, changing the template of each page might require a lot of work. The answer to this problem is the *CSV Database Upload* feature. When uploading the contents of your store from a CSV database, you can also specify what templates you want for the section and item pages. The good news is that you can use this method to change the template of existing pages. The bad news is, if your store already has manually created sections and items you really should not use the database upload method. The details of the CSV Database Upload feature are outside of the scope of this book. To read more about this feature, please see the relevant section of the online Yahoo! Store® documentation (http://store.yahoo.com/vw/upload.html).

Part I. Learning RTML

The Advanced Interface

In order to begin the journey into the world of RTML, you must become familiar with the Advanced Interface. Typically, most Yahoo! Store® owners use either the Simple (in the old editor) or the Regular Interface (in the new editor). The Simple Interface is really too simple by any standard. It is a good starting point for someone who's looking at Yahoo! Store® for the first time, but it is very limited. Once you've learned how to add and remove items, you should forget about the Simple Interface. In fact, in the new Yahoo! Store® Editor, Simple mode is no longer available.

The Regular Interface is far more useful than the Simple Interface. Through the Regular Interface, the Yahoo! Store® system exposes many more variables and properties for manipulating your store. It is possible to create a unique and functional design simply by working in the Regular Interface.

However, to gain absolute full control over your store, you need to use the Advanced Interface. This interface gives you access to a number of key areas including the following:

1. Editor configuration settings
2. File upload area
3. Database upload area
4. Controls area
5. The type and template editors.

The discussion of most of these areas is outside the scope of this book, but for the sake of completeness, each is described briefly below:

Editor configuration settings: this screen is accessed by clicking the Config button. Here you will find some settings global to the site and the editor.

File upload area: this area is used for uploading custom files: images, documents, downloadable packages such as "zip" files, etc. Access it by clicking the Files button.

Database upload area: this area is used for populating your store by uploading a specially formatted, comma-delimited spreadsheet file instead of hand-typing the items and sections. Reach this area by clicking the Database Upload button.

Controls area: this area, reachable through the Controls button, has a slightly misleading name. Here you will find a variety of functions very few of which could be considered a control of any sort. Some of the functions accessible here are: site generation and publishing, multiple image upload, editing multiple items at once, and an RTML reference (!)

The last item on the list, the Type and Template Editors, is of most interest. These are the areas where RTML templates are constructed and are discussed in detail later in this book.

The Contents List

Enter the Advanced Interface by clicking "Advanced" under the "Edit" section in your Store Manager. When you arrive at the Advanced Interface, you will see a hierarchical view of your store. The home page of your store is at the top. The sections and any top-level items are indented below the home page. At the bottom of the hierarchy, there are other pages, which do not technically belong under the home page: the shopping cart page, the index (or site map) page, the search page, the info page, and the privacy policy page. For each page of your store, the contents page lists the page's ID, its type, and the template used

to generate the page. Notice that the ID of every page is a hyperlink. Clicking on any of these takes you to that particular page of your store.

ID

The ID of a page is a unique alphanumeric string of characters. In most cases, the ID is generated by the Yahoo! Store® system based on the name you gave to the page at the time you created the page. You can provide your own IDs as well in one of the following two ways:

1. You can create a page by clicking the New button at the top of the Contents list. When you create a page this way, you can enter the ID of your choosing.
2. If you use the Database Upload feature to populate your store, you can include the ID field in your CSV as well.

There are a number of things you should know about IDs:

- Once assigned, the ID cannot be changed. If you didn't use one of the above two methods to assign the ID on your own, then the Yahoo! Store® system generates the ID for you based on the name of the page at the time the page was created. If you subsequently change the name of the page (which you are free to do at any time), the ID remains the same. The only way to change the ID of an existing page is to delete and re-create the page and use one of the two methods above to assign an ID.

- Every page in a Yahoo! Store® has an ID. If you do not explicitly specify the ID and leave the name field of a new page blank, an ID will still be assigned to that page.

- IDs are always unique. If you don't assign your own ID and create two pages with the same name, the Yahoo! Store® system will assign two unique IDs to these two pages, usually by adding some numbers to the end of the ID until the two become different. If you try to manually create a new page with an ID that already exists, the Yahoo! Store® system will create a new page but a sequential number will automatically be appended to the end of the ID you have entered.

- The ID of a page is its name in the URL of the page. The Yahoo! Store® system simply takes the ID of the page and adds the .html extension to it to come up with the name of the actual filename of the page's html page. The ID of your home page, for instance, is index, and its URL is index.html. If you create a page whose ID is, say, foo, then the filename (and URL) of this page will be foo.html.

Types

Besides an ID, every page has a *type*. The type of the page determines what properties that page has, or in other words, what fields are present when you are editing the page. An item, for example, has a *name*, an *image*, a *code*, a *price*, etc. Items are of type *item.* (with a period at the end). The Info page, on the other hand, has a completely different set of properties that you see while editing the Info page. It has a *greeting*, an *image*, and an *info* property. The difference lies with the fact that the Info page has type *info.* The type of any page, therefore, defines *what kind of* a page we are talking about.

As in the case of IDs, the type of a page is set at the time the page was created. In the new Yahoo! Store® Editor, the type cannot be changed. However, in the old editor, there is a way to change the type of a page. If you are using the database upload method to populate your store, you can specify the type for your item and section pages *before* you upload your CSV file. The new type will be assigned even to existing pages of items present in your CSV file. Be careful with changing types. If you assign a new type to your item pages, and the new type lacks fields your old type does not lack, you will lose the values of those fields. It's not fatal as you can only change types by using the database upload feature. The premise is that you have all the information of your items in your database. Still, you should be aware of this (and once again, this does not apply to the New Editor, where the type of existing pages cannot be changed at all.)

Access the list of types (both built-in and custom) by clicking the Types button.

Built-in types

There are ten built-in types in Yahoo! Store®. They are: *empty.*, *raw-html.*, *norder.*, *search.*, *privacypolicy.*, *info.*, *main.*, *section.*, *link.*, and *item.*

Note: the names of these types end with a period. The period is part of the name and indicates that the type is built-in. Since you are probably familiar with some of the objects the built-in types represent, looking at these types is a good way to become acquainted with the concept of types. We will look at each of the built-in types in detail starting with *empty.*

empty.

This type, as its name implies, is completely empty. The *empty.* type can be used for pages that require no customization. The only built-in object that uses this type is the index (or site map) page, and it is easy to see why: the index page needs no parameters. It simply generates its contents based on the list of pages that make up your store.

***raw-html*.**

The *raw-html.* type has a single field: *Html*. When used in conjunction with the *raw-html.* template, a page based on the *raw-html.* type will contain only the HTML you enter. You can think of the *raw-html.* type and template combination as an empty HTML page on which you can write your own tags without restriction. Those of you who are proficient in HTML would probably say: "Great! So I can just make all of my pages be of type *raw-html.*, use the *raw-html.* template and create truly custom store pages! So, why bother with the rest of RTML?" Here is why: from HTML, there is no way to refer to any property of any page in your store. This is such an important statement it should be repeated: **from HTML, there is no way to refer to any property of any page in your store**.

In fact, from HTML, you cannot refer to anything in your store! If you know any server-side scripting language such as ASP or PHP, you know that these languages allow one to insert script blocks inside an HTML page. For example, in ASP, one can write out the page title providing the title is stored in a variable like this:

```
<title><%=Title%></title>
```

The above code segment would yield a <title> header tag with the value of the title set to whatever the variable *Title* holds. RTML, however, is not a server-side scripting language, consequently, there is no way you can insert RTML expressions inside an HTML source. For that reason, the *raw-html.* type and template pair is not nearly as useful as you might think. They may be best to be used for splash screens or perhaps a home page created in, say, *Dream-Weaver*. For anything else, read the rest of this book, learn RTML, and use other types and templates. Once you understand RTML, you will see you *can* do just about anything you can in HTML.

norder.

The *norder.* type is used by the shopping cart. It has three properties: Page-title (with “Shopping Cart” as the default), Page-width (with 640 as the default) and Name (with “Show Order” as the default). When editing the shopping cart page, you will see exactly these fields. There are two interesting facts to note about this type:

1. Types can have default values in their properties.
2. The width of the shopping cart page is 640 pixels unless you specifically change it. Since the *norder.* type contains a property called *Page-width*, that property automatically overrides the global *Page-width* variable. You should remember this if your store’s default page width

is something other than 640 pixels. In that scenario, if your order page seems different from the rest of the pages, edit the order page and make sure you enter the correct width explicitly.

search.

This type, as you might have guessed, is the type for the search page. It has five fields: *Name* (default is "Search"), *Caption*, *Search-label* (default is "Search-for"), *Search-button* (default is "Search"), and *Headline*.

privacypolicy.

This is the base type for the privacy policy page. It has a single field called *Info*.

info.

This is the base type of the Info page. It has the following properties: *Greeting*, *Image*, *Address-phone*, and *Info*.

main.

This is the base type of the home page. It has the following properties:

- *Page-title*,
- *Page-elements* (default page elements are *Name*, *Buttons*, *Image*, *Message*, *Specials*, *Address*, *Final-Text*),
- *Image*,
- *Image-format* (default is *Left*),
- *Buttons* (default buttons are *Home*, *Contents*, *Empty*, *Show-Order*, *Info*, *Privacypolicy*, *Search*, *Index*, *Mall*),
- *Message*,
- *Specials*,

- *Specials-format* (set to *As-contents* by default,) *Contents*, *Contents-elements* (default elements are *Image*, *Screen-title*, and *Price*),
- *Contents-format* (default is *Ell*),
- *Columns* (default is *3*),
- and *Intro-text*.

Pages based on *main.* have a special characteristic. During a *Database Upload/Rebuild* operation, these pages are never replaced. This is a very handy piece of information. If you routinely reload the content of your store using *Database Upload* but would like to leave some pages intact, make those pages type *main.* if you can, and the *Rebuild* process will leave them alone.

section.

The *section.* type *can* be used for section pages. Notice that we said "can be used" and not "is used." Yahoo! Store® does not use this type for sections; it uses the *item.* type instead. You can, however, start using this type for your section pages if you plan to write your templates and make them recognize whether they are dealing with a section or an item page. The *section.* type has the following parameters: *Name*, *Image*, *Headline*, *Contents*, *Caption*, *Abstract*, *Icon*, and *Label*.

link.

This type is used, is its name suggests, for links (remember, links are created by clicking the New Link button). It has the following properties: *Name*, *URL*, *Image*, *Abstract*, and *Label*.

item.

The last built-in type is used for section and item pages. It has the following properties: *Name*, *Image*, *Code*, *Price*, *Sale-price*, *Orderable*, *Options*, *Head-*

line, *Caption*, *Contents*, *Abstract*, *Icon*, *Inset*, *Label*, *Ship-weight*, *Taxable*, *Download*, *Family*, *Softgood*, *Leaf*, and *Gift-certificate*.

Customizing Types

In many cases, you will find that the built-in templates are perfect for your new items either because you will be creating new sections or items or because one of the built-in types would be good enough to serve your particular purpose. On the other hand, there might be instances where it would be far more convenient to have access to some custom properties, properties that are not present in any of the built-in types. If you sell movies, for instance, you might want to use a type that has properties such as *Running Time*, *Director*, *Genre*, etc. In such a case, you can create your own type.

There are two ways you can create a custom type: you can make a copy of an existing type and customize the copy, or create a custom type from scratch. It doesn't matter which method you use; the results are the same, but copying and customizing an existing type might save you some time.

Copying an existing type

If the new type were an expansion on one of the built-in types, you would choose this method for creating a custom type. Taking the movie example above a step further, a custom type for movies is really just an expanded *item.* type, as you will still want to have access to the standard properties such as *Name*, *Image*, *Price*, or *Caption.* Besides these, you will need some extra properties such as *Running-Time*, or *Director*, as well. To create this type, click Types, click *item.* under Built-in Types, click "Copy", enter the name of your new type, say Movie-type, and click "Copy." You now have a copy of the *item.* type. You can now start adding your new properties by clicking the "Define New Property" button.

Creating a type from scratch

Creating a type from scratch is not much different from making a copy of an existing type and customizing it. To start from scratch, click Types, then click New Type. Name the type and click the "Create" button. The type you created this way will have no properties at all. You will have to add all the needed properties by clicking the "Define New Properties" button.

Types of properties

Now you know how to create types. We mentioned earlier you could add custom properties to these types but what types of properties can you add? Yahoo! Store® currently includes 14 property (or variable) types: *Text*, *Big-text*, *Number*, *Numbers*, *Positive-integer*, *Symbol*, *Color*, *Ids*, *Yes-No*, *Font*, *Objects*, *References*, *Image*, and *Orderable*. Depending on what the property will be used for, you will select one of the 14 property types above. The following is an explanation of the various property types available in Yahoo! Store®.

Text Used for small text fields such as *Name*. Whenever you need a custom property that will contain at most one line of text, use this one.

Big-text Used for larger text fields such as *Caption*, or *Abstract*. Use this property type for properties containing several lines of text or HTML.

Med-text Same as Big-text, except it takes up less space on edit forms.

Number Used for numerical information. The value of properties of type *Number* can contain positive and negative numbers (both whole numbers and fractions) and may be used in numerical calculations in RTML.

Numbers A list of numbers. Each element of the list is a number. An example of such a property is the *Price* of an item. The price of an item can contain either a single number, or, in case of volume pricing, a list of numbers the first being the regular price, and the rest, prices for various quantities. Example: 10 2 8.

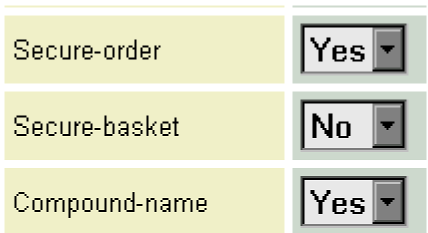
Positive-integer Properties of this type can contain only positive whole numbers. An example might be *Page-width*.

Symbol Values for this property type can contain only numbers, letters, periods, and minus signs (-). Letters are automatically capitalized.

Color This property type can hold a color value. A color value has three numerical components each representing the intensity of red, green, and blue on a scale from 0 to 255. Properties of type *Color* automatically get a “Select” button and a small patch of color for display as in the example below.
Examples of this property type are *Background-color*, *Link-color*, *Text-color*, etc.

Ids A list of IDs separated by spaces.

Yes-No This type can contain only two values: Yes or No. In the user interface, such variables appear as in the example below:



Font This property type is used for storing information on fonts available in Yahoo! Store® only. These are graphical fonts and not standard HTML fonts. Examples are *Button-font*, *Home-button-font*, etc. Properties of this type, in the user interface, contain a button for selecting a font from a pre-defined list:

Objects Remember the hierarchical list of the pages of your web site when you click the Contents button? That hierarchy is created by using the *Contents* property of the pages of your store. Typically, the home page contains some sections, so those sections form the *Contents* of the home page. Those sections, in turn, have pages as their *Contents*. When you edit the home page, for example, you will see under *Contents* the IDs of all the sections (and perhaps items) that are below the home page. The type of the *Contents* property is *Objects*. This type is used to contain other pages.

The page that has a property of type *Objects* is commonly referred to as a "parent" page. It is said to *own* the pages stored in the *Objects* type property. The pages that are stored in the *Objects* type property are referred to as the "children" of the "parent" page.

In the contents list, child pages are listed indented below their parent pages. Each page can be the child of exactly one parent in the contents list; nonetheless, you are free to enter the same page into the *Objects* type properties of two or more pages. This is how multiple containers work, that is, when some of your items belong to more than one category. If a page has multiple parents, then in the contents list, this page will be listed below one of the parents but not both or all.

References This property type is very similar to the IDs type. The difference is that if an item whose ID is entered into a *References* field is deleted, its ID will also be removed from the *References* field automatically. An example is the *Specials* property of the home page.

Image This property type is used to store images such as the *Image* property of most regular pages. In the user interface, properties of this type have an "Upload File" button to allow you to upload an image for the property, a "None" button to allow you to remove an existing image, and a small thumbnail representation of the image loaded into the property:

Orderable This property type works the same way as the *Yes-no* type with the only exception, that these types, in the user interface, also get a "Sell on Yahoo! Auctions" button:

Default property values

When you click the name of a type, you see a form with all the properties available to that type. For example, looking at the built-in *item.* type you will notice that some of the properties have values. The *Orderable* property of the *item.* type, for instance, is set to "Yes." This means that each new item you create in the store will have its *Orderable* property set to "Yes" by default.

These defaults can easily be set for custom types. All you have to do is edit your custom type in the type editor, enter or select the values you want to be the defaults for each property (where you do want a default), and click "Update." Default values don't make sense under all circumstances. For instance, you probably don't want to set a default value for the *Name* property.

Finally, here is an important little detail pertaining to the **Old Editor Only**: if you set a default in the type editor for a property, when you click "Update," the new default value will automatically be set for that field in all objects in your store that are based on the modified type. This will override any existing values. As an example, let's say you have a custom type called *my-type* which has a property called *Color*. You also have some pages in your store based on *my-type*. If you set the default for *Color* "Red" then "Red" will be entered for *Color* for all pages of type my-type in your store regardless of what their *Color* had been set to before. **The New Editor does not have this problem.**

Because you cannot change the defaults for properties of a built-in type, you are never running the risk of overwriting existing property values of pages based on any built-in type.

Using custom types

There are two ways you can specify the type used for a page: when creating a page from scratch by clicking the New button in the contents list, or when using the database upload feature.

When you create a new item in the Advanced Interface by clicking the New button, you can both assign an ID to your new item and select its type from a drop-down list. This way, you can tell the Yahoo! Store® system which type you want to use for the new page.

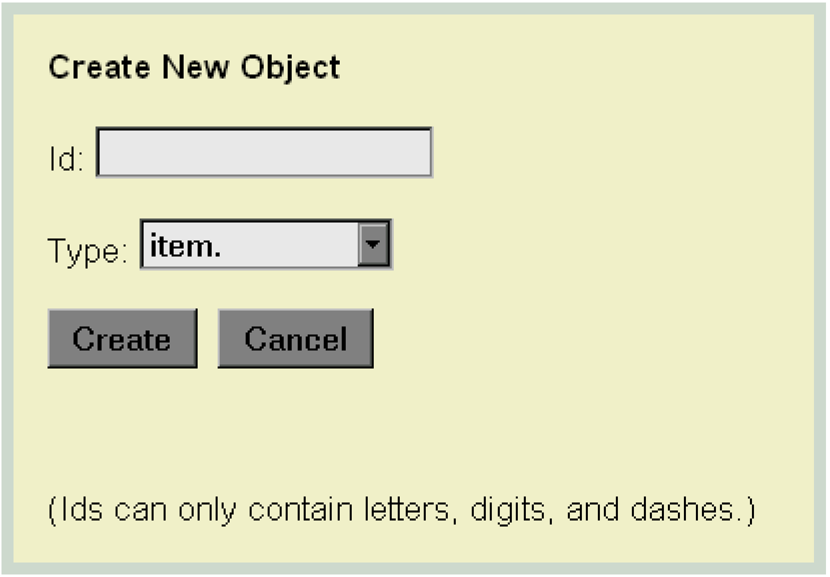


Figure 2 - Creating a new item in Advanced Mode

When using the database upload feature to add items to your store, the upload screen contains drop-down menus (see Figure 3) for you to select the type to be used for the uploaded items and generated sections. In the old editor, you can even change the type of existing pages by including the existing pages in your CSV file, and choosing a different type from the drop-down.

The default type for both items and sections is *item*. If you want to change this default to some of your custom types, click Config, and change the default types on that screen. Those defaults, by the way, only apply to database upload; they do not apply to items created manually.

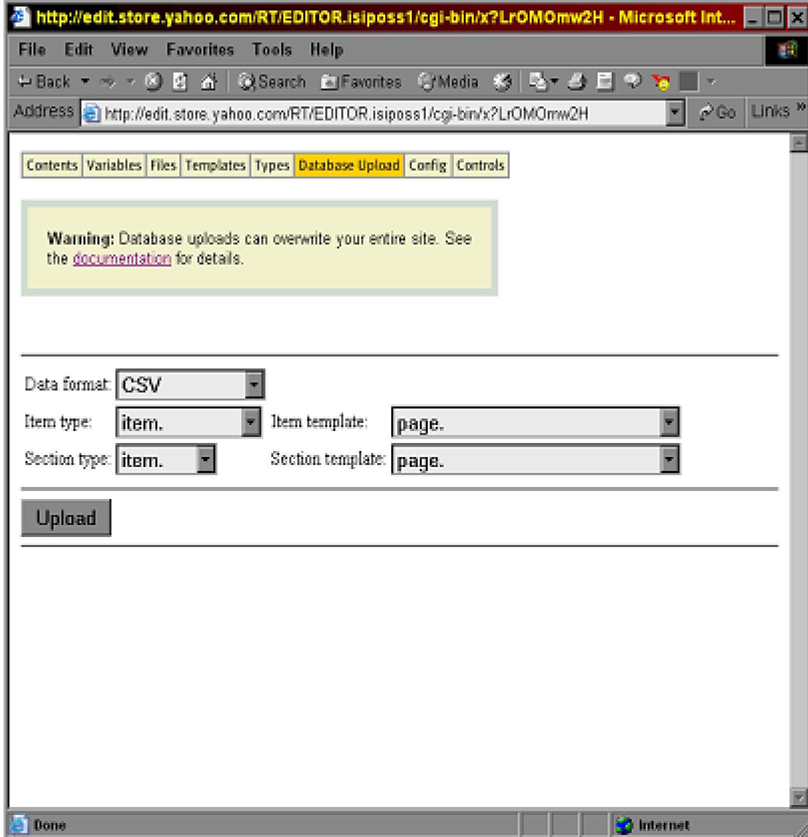


Figure 3 - Specifying the type during a database upload

The only way to change the type of an existing page is through the database upload feature (and even that only works in the Old Editor.) Therefore, it really pays to plan your store correctly early on. Don't just start "hacking at it." We highly recommend you build your store from the very beginning using the database upload feature even if you only plan to sell a handful of products. If you populate your store this way from the beginning, you can always use the upload feature to change the type or template of your existing pages in one sweep.

As a final point, to take advantage of custom types, you must either write your own templates from scratch, or make some modifications to the existing templates to use any custom properties your types might have. Remember, **built-in templates have no knowledge of any custom properties.** Attempting to use custom types without custom templates is useless.

Custom types and accessories

There is something you should be aware of when using custom types: Yahoo! Store® will **not** display the Accessory or New Accessory buttons for items based on custom types. Yet, there is still a way to create an accessory; you just have to realize that accessories are nothing more than items that don't have their own pages. Instead, they share a web page with their parent item. So, to create an accessory for an item based on a custom type, follow these four steps:

1. Create a new item by clicking New. This will be the accessory. Leave the template of the accessory as nil*, but fill in the rest of the page.
2. Edit the item for which you want to set up an accessory.
3. Make sure that the *Leaf* property of this item is set to "Yes."
4. Enter the accessory's ID under *Contents*.

Or, you can click the Item button and create a new item page that way.

* nil in RTML indicates the absence of a value. It is a special name, not the British phrase for null.

Custom types and Database (CSV) Upload

The Database Upload function of Yahoo! Store® has its shortcomings (you cannot use it to override variables, upload text for sections [at least not easily], generate accessories [again, without some "trickery"], or upload any kind of images). Nonetheless, using Database Upload from the beginning can save you a tremendous amount of time—especially if your store contains many items. We mentioned earlier that you could use the database upload function to change or specify the type and template of your store pages. In addition, the Database Upload feature allows you to specify values for custom properties. This feature can have several applications.

When creating your own types, you can use the database upload function to populate your site—as before—and provide values for any custom properties your types might have. For example, if you are selling books using a custom type you created that has an ISBN property, you can specify the ISBN number for the books by simply adding an ISBN field to your CSV file before uploading.

You have probably noticed that with the database upload function it is not possible to override variables. This is quite an annoying problem and is perhaps the most visible when attempting to use the database upload feature to provide keywords for pages. Of course, you cannot do it. That is, not unless you define a custom type! With a custom type, you can add a new property of type text called *Keywords*. Now, you can upload your keywords for all your pages. **If you are using the Old Editor**, the Yahoo! Store® system will treat the custom *Keywords* property as if you have overridden the *Keywords* variable for all of your pages. You can use this trick to override any variable from your database (again, but only in the Old Editor.) To revert to the globally defined variable, simply edit the page, click "Undo Override", select the variable (*Keywords* in this example), and click "Update." If you use the name of an existing variable for a custom

property, the database upload function will treat this property as a simple variable override.

Deleting custom types

In the Old Editor, there is no way to delete a custom (or built-in) type. In the New Editor, custom types can be deleted if there are no objects based on that type. To delete a type, go to the Types page, and click the "Delete" link next to the type.

Templates

Templates are basically programs or sub-routines written in RTML. They consist of RTML operators, all of which return some value. Each template returns some value as well: either the value of the last RTML operator within the template, or a value explicitly specified by you. They can also take one or more parameters or arguments. Templates are constructed in the RTML Editor, which is accessible by clicking the Templates button.

The Template Editor

RTML is different from most conventional programming languages in that it does not allow you to write your RTML code in a text editor and upload your code into your Yahoo! Store®. Instead, you need to use the Template Editor and, in effect, "point and click" your RTML code into the editor.

The Template Editor is a complex web-based programming tool. When you first click the Templates button, you only see the list of built-in templates and a New Template button. At first, using the Template Editor will feel rather strange, but as you learn it, you will soon discover that it is actually quite possi-

ble to "click" RTML code fairly quickly. We believe the best way to start learning this tool is to start using it, so let us go through a very brief tutorial.

In this tutorial, we are going to write a very simple template. All this template is going to do is to write two sentences on a web page: "This is my first template" and "This is great!"

To begin, click the New Template button. You will be asked to enter an Id for your template. The Id of a template is its name. You can use any letter, number, and the hyphen to name your template. For this template, type "my-1st-template" and click "Create." As soon as you hit "Create," your new template will be created on the screen. It will look like this:

```
My-1st-template ()
HEAD
  TITLE "untitled"
BODY
```

Notice that the template editor automatically entered a few operators for your blank template. If you know some HTML you will probably understand right away what they are:

1. The first operator, HEAD, creates the <HEAD> and </HEAD> tags for your page.
2. Within the <HEAD> and </HEAD> tags, the editor placed a <TITLE> tag with the actual title set to "untitled".
3. Finally, the editor also created a <BODY> section for the page.

Without making any changes to this template, it would generate the following HTML page:

```
<HTML>
<HEAD>
    <TITLE>untitled</TITLE>
</HEAD>
<BODY>
</BODY>
</HTML>
```

This isn't much; so let's add a line of text to this template. Click the New button. You are presented with the *New Operator* form (Figure 4.) This form has two sections: a text box labeled "Simple," and a list labeled "Complex." The textbox is used to enter constants, variables, names of objects, etc. The "Complex" list is used to select RTML operators. Scroll down that list, highlight TEXT and click "Create." Note, that you cannot type TEXT into the textbox labeled "Simple." You can never use the "Simple" textbox to create an operator.

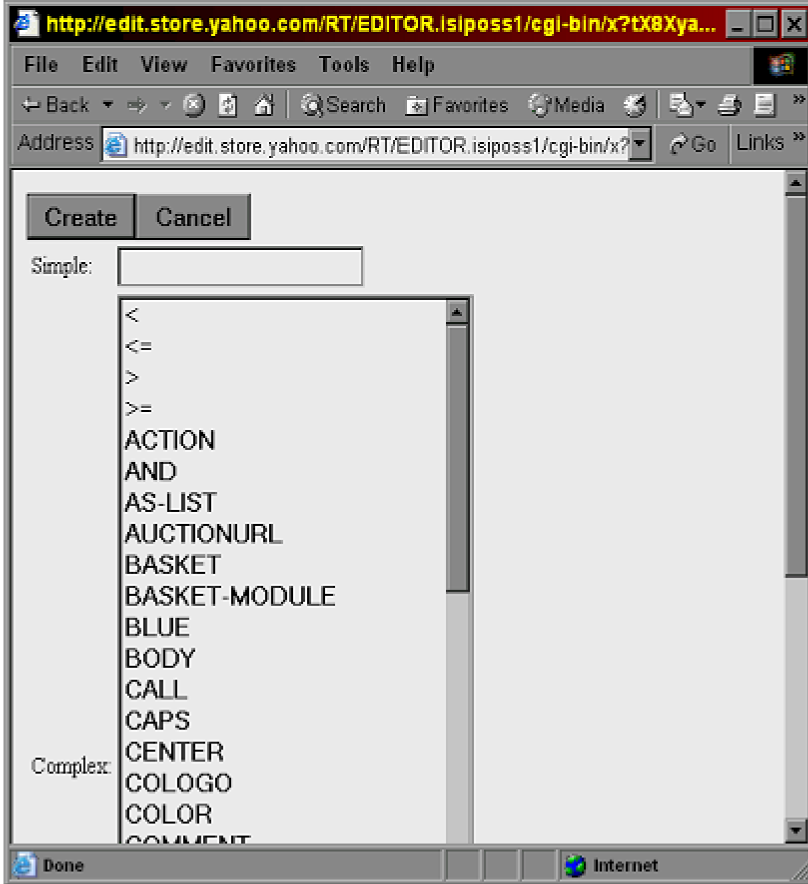


Figure 4 - New operator

When you click "Create", you are sent back to the page with your new template on it. On this page, though, you will notice a new toolbar below the code of your template. Below that new toolbar is the new TEXT operator you just created. This new toolbar is the *Expression Stack Toolbar* and the area below it—with your newly created TEXT operator—is the *Expression Stack*. What is the *Expression Stack*? The *Expression Stack* is exactly like a stack of cards. Each new operator you create goes to the ***top*** of the *Expression Stack* before you can do anything with the operator. In most cases, you can only work with the

operator at the top of the *Expression Stack.* You can take the operator at the top (the first listed) and insert it into your RTML code (we will get to that in a minute), or remove it from the stack. If you have several operators on the *Expression Stack*, you can move any one of those operators to the top by first clicking the operator you want to move, and then clicking the To Top button. You can also remove everything from the *Expression Stack* by clicking the Clear button.

Now, getting back to our brief tutorial, we want to insert the newly created TEXT operator into our template. It should go into the <BODY> section of the page, so click `BODY`. Notice that as soon as you click *BODY*, a number of buttons change from gray to pale yellow indicating that those buttons are now available or enabled. We want to insert the TEXT operator into the BODY section, so click Paste Within. As soon as you click that button, your TEXT operator disappears from the *Expression Stack* and goes below `BODY` indented slightly. Notice, that TEXT is now in black and is not hyperlinked (you cannot click on it). The fact that TEXT is black indicates that it is selected. Commands you click now will act on it. Click Edit. You will see a new page with TEXT and a textbox next to it. This is where you can fill out the blanks (or parameters) for the TEXT operator. Enter "This is my fist template" (including the double-quotes) and then click "Update." You are back to your template, and "This is my first template" appears next to TEXT.

We are going to insert a line break after this text line, so click New again. Select LINEBREAK from the operator list, and click "Create." You now see the following on the *Expression Stack:*

```
LINEBREAK clear nil
          number nil
```

We want to put the line break after our first text line. Since TEXT is already selected, (if not, click it now), we can simply click [Paste After]. LINEBREAK will go from the *Expression Stack* into the template right below the TEXT line. Because it was the last operator we worked with, it will be selected now instead of TEXT. Notice that LINEBREAK has two optional arguments: *clear* and *number*. These don't concern us for now, as we only want a simple line break after our first line. To make those arguments disappear, click the [Contract] button.

Finally, we are going to insert our second line. Because we already have a TEXT operator on the screen, instead of clicking [New], we are going to learn a different method. Click TEXT inside the template to select it. Now, click [Copy]. This will put a copy of the entire TEXT line from the template onto the *Expression Stack*. Because we want this new text line to go after our line break, click LINEBREAK to select it and then click [Paste After]. At this point, we have our second text line after the line break, so we are almost there! All we have to do now is to change the text of this second line. Click [Edit] and change "This is my first template" to "This is great!" Now click "Update."

There you have it: your first template! It wasn't too difficult but it isn't too useful either, at least not without figuring out how to use it. A template by itself is useless until there is a page that makes use of it. So, click on [Contents] to go to the Contents List, and then click [New]. On the Create New Object page, enter *my-page* for Id, leave Type as *item.*, and click "Create." You will now see the edit page of your new item. Change the default template from *page.* to *my-1st-template* and click "Update." You will be back at the Contents List with your new page, *my-page*, at the bottom. Click its ID to look at it. If all went well, you will see a screen similar to the one shown in Figure 5.

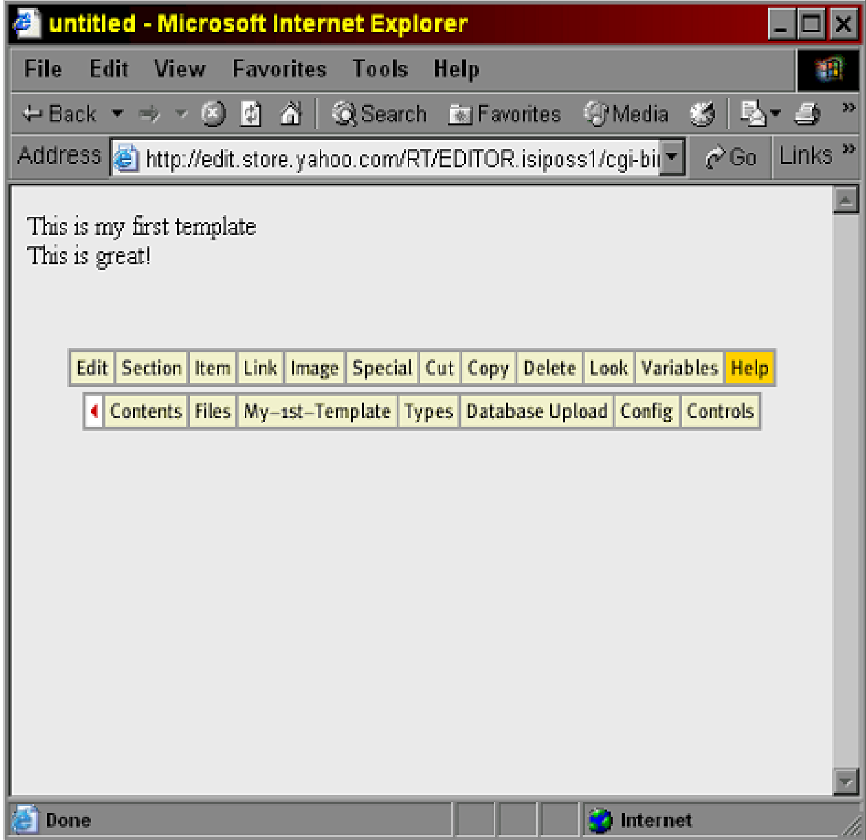


Figure 5 - Your first template

Let's get back to our template. Notice that in the toolbar, there is a button labeled My–1st–Template. Whenever you are editing a page that uses a custom template, Yahoo! Store® will include a button for that template in the toolbar. To go to that template, simply click its button. In this case, click My–1st–Template.

This was definitely a brief exercise, however, it did include a number of important elements of the RTML Editor:

1. It showed you how to create a new template.
2. It illustrated how to create a new operator either from scratch or by copying an existing operator from the template itself.
3. It taught you how to work with the *Expression Stack*.
4. Finally, it demonstrated how you can paste operators within other operators, and how to paste operators after other operators.

Before going any further, "play around" with the template editor. You can use the template you created during the previous tutorial. Just add more TEXT lines or line breaks to it, or rearrange the lines. It doesn't matter what you do, just "click around" so that you get a feel for how the *Expression Stack* works and how you can work the RTML Editor. Don't worry; you cannot "break your store beyond repair" (contrary to what you might have heard or read about this) just by playing with a custom RTML template.

The Template Editor Toolbar

During the brief tutorial in the previous section, you used a few buttons from the toolbar called the Template Editor Toolbar. Besides Paste After and Paste Within, this toolbar has a number of other buttons allowing you to manipulate various template expressions and operators. With the exception of the New button, all the buttons of the Template Editor Toolbar apply to the currently selected template element (a template element is anything clickable in a template). If there is no selection (none of the elements of the template are in **bold** letters, and everything is clickable in the template) then all of the buttons

will be grayed out except New. Whether or not some of the other buttons are enabled, depends on the currently selected template element as explained below.

Edit

This button is enabled if the currently selected template element is *editable*. Editable elements are typically operators that take one or more parameters. If the current template element is editable, clicking the Edit button will bring up a form showing the template element (operator) and all of its parameters. The parameters can be changed and saved.

Cut

This button cuts the current template element from the template and puts it on the expression stack. If a complex element is cut (one that has other expressions pasted within its body) then the entire branch—the element and everything within it—is cut and placed on the expression stack. The element cut this way may be pasted into a different location within the current or another template.

Note: if you cut all the operators from a template, you will not be able to insert any new elements into the template! If this happens, delete and re-create the entire template.

Copy

This button is similar in function to the Cut button. The difference is that Copy places a copy of the current template element on the expression stack and leaves the original element in place.

New

The New button allows you to create and place a new expression on the expression stack. The expression can be either simple such as a text string or an object name, or complex such as an operator.

Paste After

This button pastes the expression on the top of the expression stack after the current template element. If the expression stack is empty, this button will be disabled (grayed out).

Paste Within

This button pastes the expression on the top of the expression stack within the body of the current template element. This will only happen if the current element may contain other expressions within itself. For instance, the `TEXT` operator cannot include any expressions within its body. Therefore, if the current element is a `TEXT` operator, then the Paste Within button will be disabled. It will also be disabled if the expression stack is empty. If the current element already has one or more expressions pasted within its body, then the newly pasted expression will "push" all the existing expressions down and it will become the first expression within the current element.

Replace

This button swaps the current element with the one on top of the expression stack.

Contract and Expand

The last button in the Template Editor Toolbar is a "toggle" button, which can be either Contract or Expand. If the current element is an operator that can take one or more parameters, Expand will reveal all the parameters—even those that have no values. When all the parameters of such an operator are revealed, Expand becomes Contract. When clicked, Contract hides all the unused parameters of the current operator.

Built-in templates

Click the Templates button to go to the template list. You will now see two columns, one titled "Built-in Templates" and the other one, *Your Store ID Templates* where *Your Store ID* is the Yahoo! ID of your store. Your templates right now include only *my-1st-template*, the template you created in the tutorial of the previous section. The other column, the one with the built-in templates; however, includes a long list of template names. These are all the templates that are responsible for generating your Yahoo! Store®. You can probably guess what some of these templates do just by reading their names. *Button.*, *bullet.*, *index.*, and *page.* Built-in templates are great for learning some of the tricks and twists of RTML. Let's look at one. Click on *page.*, for example. By glancing over this template, you will recognize some elements right away. You will see HEAD and BODY (familiar from our tutorial in the last section); you will see references to variables such as background-color and background-image, and you will notice the two separate sections responsible for generating the HTML page (if your store has top or side buttons).

Notice, too, that next to certain lines, there are small red rectangles. These rectangles are always next to lines that start with the CALL operator. These lines are calling other templates to perform certain tasks. You can click the red rec-

tangle to go to the particular template. This is a great feature because it allows you to "drill down" or follow the logic of how certain things are done within RTML. To try this feature, click the red rectangle next to `CALL :nav-buttons.` You'll immediately go to the *nav-buttons.* template. Notice at the top of this page a line that reads, "Called by:" followed by a number of template names. This line tells you which templates (if any) call the template you are looking at. These template names are all hyperlinked to the respective templates. The *nav-buttons.* template, for instance, is called by these other built-in templates: *base-item., home., index., info., item-list., order-page., page., privacy-policy., section.* To go back to the page you came from (the *page.* template), simply click its name in the "Called by:" line.

A word of caution here if you are using the Old Editor: don't ever use your browser's back, forward, or refresh button while in the RTML Editor! In fact, try not to use these buttons at all while editing Yahoo! Stores®. Using these buttons can confuse the store designer system and you might end up with some undesirable results or even lose pages, sections, or templates. The New Editor does not have this limitation.

A good way to learn RTML is to look at the built-in templates and find out how certain things you recognize are done. Here is a partial list:

- *index-body.* : this template generates the alphabetical list of pages for the index page.
- *info-body.* : this template creates the content of the info page.
- *order.* : this template puts the price, the options (if there are any) and the order button on the item pages.
- *side-nav.* : this template creates the side navigation bar.

Two final notes about built-in templates:

The name of a built-in template always ends with a period. This convention is used so that they are easier to tell from custom templates. Custom templates (just like custom types) cannot contain periods in their names.

The operators within a built-in template are not hyperlinks because built-in templates cannot be modified. Remember, earlier we said you couldn't break your store beyond repair just by changing custom templates? That is because—again—built-in templates cannot be modified. Therefore, even if you really mess up one of the templates you created or modified, you can always go back to the built-in templates; they will always work.

Modifying templates

Tweaking the existing templates is an excellent way to get up to speed on RTML, but you now know that the built-in templates cannot be modified directly. What you can do is make a copy of a built-in template and then modify the copy. When you make a copy of an existing template, Yahoo! Store® automatically creates copies of all other templates referenced by your copy. For this reason, a good rule of thumb is, before you do any custom RTML work, make a copy of the following top-level templates: *home.*, *page.*, *info.*, *index.*, *order-page.*, and *privacypolicy*.

Making minor changes to the existing templates is what most Yahoo! Store® owners want at first. Here is a brief tutorial to demonstrate how this is done. We are going to make an often-requested enhancement to a store: we are going to add a "search box" right above the navigation buttons in the left navigation bar.

To begin, click Templates. Scroll down to *page.* and click it. Now click Copy Template. When prompted for the new ID of the template, enter *my-page*. The template responsible for the left navigation bar is called *side-nav*. In the

template list, scroll down until you see *side-nav.,* and then click the name of your copy of *side-nav*. Your copy should be listed right next to *side-nav*. It is named *Your Store ID-side-nav*, where *Your Store ID* is the Yahoo! Store® ID of your store. My store's ID is isiposs1, so below is what my copy of the *side-nav* template looks like:

```
Isiposs1-side-nav (vnav)

TABLE-CELL
  IMAGE source vnav
        antialias-color @button-edge-color
TABLE-CELL
  SHIM height 1
       width 26
```

This template creates two cells in a table. The first one contains the navigation buttons. The second one is just a spacer cell, so that the content of your store won't run together with the navigation bar on the left.

What we want to do is add a "search box" above the image of the navigation buttons in the first cell. To start, click New, select SEARCH-FORM from the operator list, and click "Create." Now, click the first TABLE-CELL operator to select it, and click Paste Within.

At this point, the body of your template should look like this:

```
TABLE-CELL
  SEARCH-FORM label nil
              size nil
              button nil
  IMAGE source vnav
        antialias-color @button-edge-color
TABLE-CELL
  SHIM height 1
       width 26
```

We now need to fill out the search form parameters, so click Edit, and type the following: for label, type "Find:
" (include the double-quotes!); for size, enter 10; and for button, enter "Go".

We will cover the SEARCH-FORM operator later; so don't worry about these parameters for now. The final version of the template body should look like this:

```
TABLE-CELL
  SEARCH-FORM label "Find:
"
              size 10
              button "Go"
  IMAGE source vnav
        antialias-color @button-edge-color
TABLE-CELL
  SHIM height 1
       width 26
```

Notice how the double-quote character at the end of "Find:" was moved down to the next line. This happened because we included a <br> line break tag inside the label parameter of the SEARCH-FORM operator. Since the RTML editor is really a web page, it rendered the <br> tag as it would any other HTML tags.

Finally, we need to modify each page that will have the search box above the navigation buttons to use our custom template. Click on Contents, and then edit any page in your store. (In the Contents list, you may not see any of your pages, only the word Contents:… below index. If this is the case, click the word "Contents:" to expand that branch.) Once in the page editor, change the default *page.* template to *my-page* and you are set. The final result should look something like what you see in Figure 6.

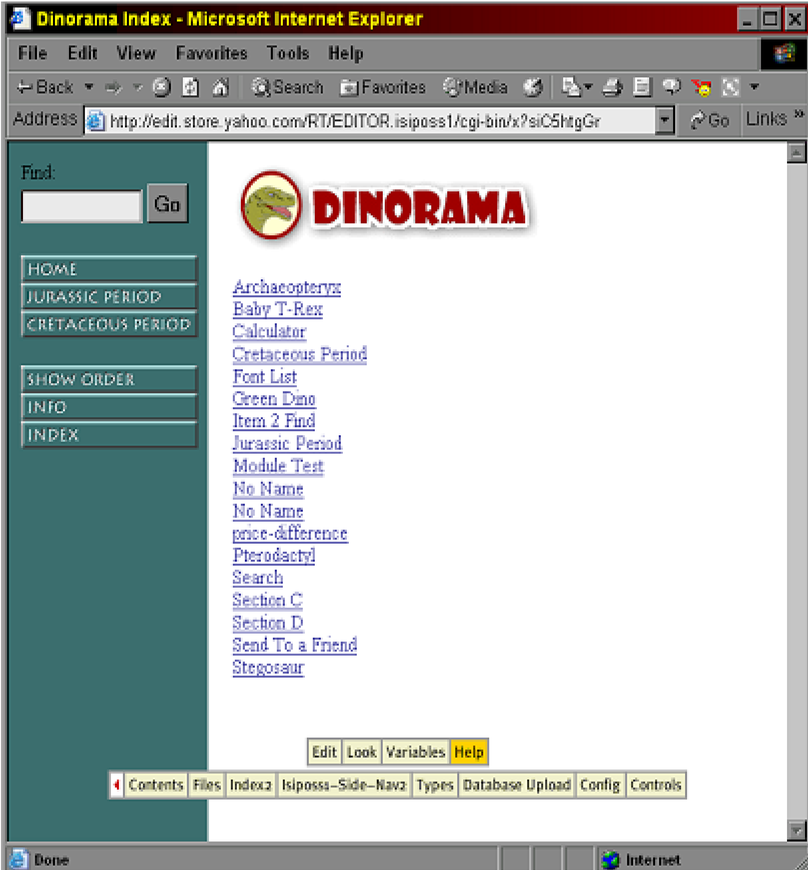


Figure 6 - Search box in the navigation bar

For a more complete solution, you will need to do the following:

First, Make copies of the following templates: *home.*, *info.*, *index.*, *order-page.*, and *privacypolicy.* Second, edit the home, info index order, and privacy policy pages, and replace their default templates with your copies.

And a final note on this example: you should make sure the actual search page does not put the search box on the navigation bar, otherwise on the pub-

lished site you'll end up with two sets of search results! Here is a fairly easy change that will make sure the search page has only one search form:

```
WHEN NOT EQUALS value1 @type
                value2 :search.
    SEARCH-FORM label "Search"
                size 10
                button "Go"
```

Writing your own templates

Modifying the built-in templates can do wonders for your store. You can greatly enhance existing features or add completely new ones without losing any of the existing functionality. Built-in templates are great because they were created by professional RMTL programmers and they work with all the parameters and settings available in your store. There are times, on the other hand, when the built in templates just won't do. Either you need to add some extra functionality or you simply want to get away completely from the built-in look. In such a case, you will have to construct your own templates. Earlier, while discussing the template editor, we created a simple template. We are going to revisit it through another template to show you a few more aspects of templates.

In this example, we are going to write a template as a replacement for an RTML operator, TABLE. The TABLE operator is used to create HTML tables. It takes a number of parameters that affect what the table will look like. However, it is missing some other parameters the equivalent HTML <table> tag has. Among others, the TABLE operator lacks the ability to specify a background image for the table. We are going to write a template that will overcome this limitation.

To begin, click on Templates, and then click New Template. When asked for the id of the new template, enter *my-table* (the name is not important as long

as you conform to the naming rules of including only letters, numbers, or minus signs in the name).

When you click "Create", you will see the following –familiar – template skeleton:

```
My-table ()
HEAD
  TITLE "untitled"
BODY
```

What we want from our template is to create an HTML <table> tag with most of its parameters including the parameter used to specify background images called *background*. In order to do this, our template must take some parameters. To set these parameters, click the () next to *My-table*. You will see a new form with a text box labeled "Parameters of My-table:". Enter the following into the box: *border align cellspacing cellpadding width background*. (no commas, only spaces separating them). When you are done, click "Update." Now, the template has six parameters.

Next, begin building the actual body of the template. Click New, select TEXT from the list, and then click "Create."

If HEAD is not selected in the template skeleton, click it, and then click Replace. In this template, we don't need the HEAD operator (nor the BODY operator but we'll deal with that a bit later). With the newly inserted TEXT operator selected, click Edit. For the parameter of TEXT, enter "<" and click "Update."

Old Editor Note: As a side note, we are creating a modified <TABLE> tag, but because anything we enter for the parameter of the TEXT operator will also appear in the listing of our template, we are going to construct the <TABLE> tag by printing each of its components (<, TABLE, attributes, and >) separately.

At this point, our template should look like this:

```
My-table (border align cellspacing cellpadding width
background)
TEXT "<"
BODY
```

For this next step, we don't need to click New again. We need another TEXT operator and the first one has already been selected. Simply click Copy. This takes the `TEXT "<"` line and places a copy of it on the expression stack. Next, click the word `BODY` in our template. Once it is selected, click Replace. Since we don't need the `BODY` operator in this template, we managed to get rid of it by replacing it with an operator that we do need.

Click Edit, and change "<" to read "table border=" (including the double quotes). When you are done, click "Update." To provide our own border parameter, click Copy and then click Paste After. The last two steps essentially duplicate the `TEXT "table border="` expression. What we now have is the following:

```
My-table (border align cellspacing cellpadding width
background)
TEXT "<"
TEXT "table border="
TEXT "table border="
```

Click Edit again, and replace "table border=" with *border* and click "Update." We have changed the template so that it prints the value of the border parameter of My-table right after "<table border=".

Click Copy again and then click Paste After. Next, click Edit, replace *border* with " align=" and click "Update." (including the double quotes this time, as well as the space before the word *align)*.

Repeat these steps until your template looks like this:

```
My-table (border align cellspacing cellpadding width
background)

TEXT "<"
TEXT "table border="
TEXT border
TEXT " align="
TEXT align
TEXT " cellspacing="
TEXT cellspacing
TEXT " cellpadding="
TEXT cellpadding
TEXT " width="
TEXT width
TEXT " background="
TEXT background
TEXT ">"
```

That's it! We now have a template that can create a table with a background image. Let's put it to use. To start, find an image you might want to use for a background, upload it into the Files area (click Files to access that area), and make note of the image file's name. This template is what you might call a "utility template:" it is used by other templates, not by itself. In order to use our template, we need to create another template that will call My-table. Click on Templates, then New Template. Enter *table-test* for ID and click "Create." In this template, we do want the head and body tags (because this template will be used to render a web page). Click New, select `CALL` from the list and click "Create." Paste this new `CALL` operator within the `BODY` element (in case you forgot how to do this, first click on the word `BODY`, and then click

Paste Within). While `CALL` is selected, click Edit, and type *:my-table* for its parameter.

The next step is to provide the parameters to our template. Parameters need to be pasted within the `CALL` operator (in the order the parameters are defined for the template to be called). For our purposes, we want a table with a background. Let's use no border but make the table fill the width of the page. To do this, first click New, and type (not select!) 0 in the box, then click "Create." Paste this 0 within the `CALL` operator we have just inserted into our template. By doing this, we have just passed 0 for the border parameter of My-table. We don't care about alignment, so click New again. Type the word nil into the box and click "Create." Paste this nil after the 0 you have just pasted. For cell spacing and cell padding, we again don't care (for now). Paste two zeroes after our nil. We want to make our table fill the width of the page. Set its width to 100% by clicking New, enter "100%", click "Create" and paste it after the last 0. Finally, we want to specify our background image. Again, click New, enter the name of the image you uploaded (in my case, it was "/lib/isiposs1/background.jpg") and click "Create." Paste this image name after the "100%" parameter. At this point, your template should look like this one:

Table-test ()

```
HEAD
  TITLE "untitled"
BODY
  CALL :my-table
    0
    nil
    0
    0
    "100%"
    "/lib/isiposs1/backgrnd.jpg"
```

We are not done yet. For the table to work, we need to create a table row and at least one cell. Start by clicking New, select TABLE-ROW from the list, click "Create" and paste this table row after the CALL to :my-table. Next, create a TABLE-CELL operator, and paste it within the TABLE-ROW you have just created.

Finally—so that we actually put something in this table—paste two lines of text separated by a line break ***within*** the table cell. You should have a template like the one below:

Table-test ()

```
HEAD
  TITLE "untitled"
BODY
  CALL :my-table
    0
    nil
    0
    0
    "100%"
    "/lib/isiposs1/backgrnd.jpg"
  TABLE-ROW
    TABLE-CELL
      TEXT "This is my first template"
      LINEBREAK
      TEXT "This is great!"
```

We are almost there. The *my-table* template created an opening <TABLE> tag. A corresponding closing one is required as well. Create three `TEXT` operators and paste them after the `TABLE-ROW` operator. The three `TEXT` operators should write the closing TABLE tag. The finished template should look like this:

Table-test ()

```
HEAD
  TITLE "untitled"
BODY
  CALL :my-table
    0
    nil
    0
    0
    "100%"
    "/lib/isiposs1/backgrnd.jpg"
  TABLE-ROW
    TABLE-CELL
      TEXT "This is my first template"
      LINEBREAK
      TEXT "This is great!"
  TEXT "<"
  TEXT "/table"
  TEXT ">"
```

To test the template, a page needs to be created. Click Contents then click New. Type an Id for this new page (say *table-test*) and click "Create." For the template of this new page, type, *table-test*. When you're done, click "Create". Looking at your new page, you should see a table with your background image similar to the one shown in Figure 7.

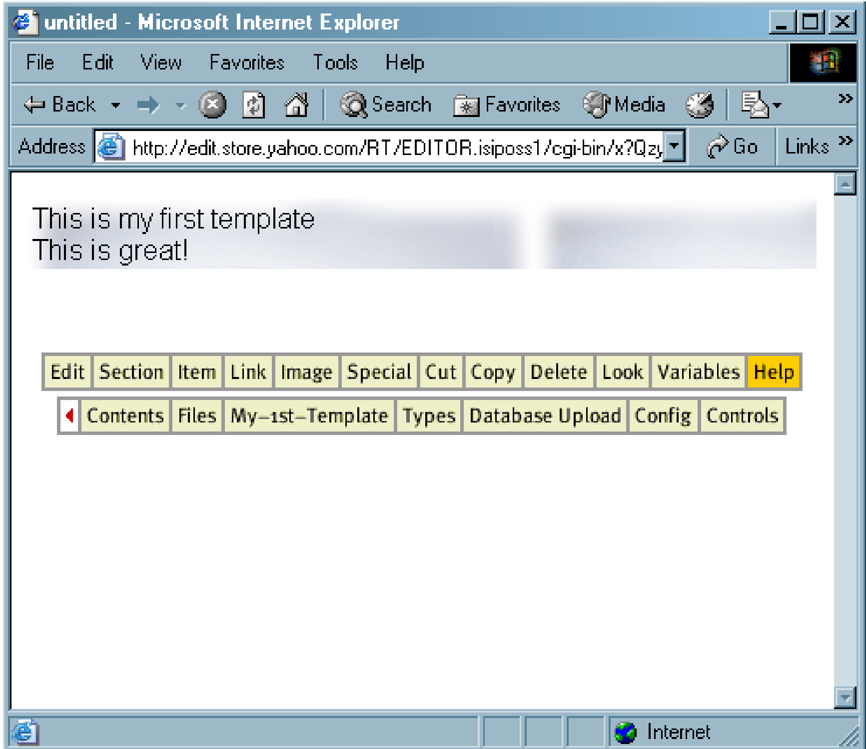


Figure 7 - table with a background image

Calling templates

Once you have your own or modified templates, you need to use them somehow. There are two ways to execute or evaluate a template:

The first method is to edit a page in your store and type your template's name in the *Template* parameter. This method makes your template the base template of the edited item. When Yahoo! Store® needs to generate the page for the edited item, your template will be executed. By default, every item and section in your store is based on the *page.* template. When Yahoo! Store® creates the page for an item, it executes the template called *page*. If you edit a page and change the

default template from *page.* to one of your own templates, you will see the result of your own template when you view the page. You are free to enter the name of any valid template into the *Template* property of a page—as long as the template you enter exists and takes no parameters. If you enter a non-existent template, you will receive an error message similar to the one in Figure 2. If this happens, you can edit your item again and fix the name of the template.

The second method is to use the CALL operator. If you want to execute a template from within another template, use this method. The CALL operator takes one parameter, the name of the template to call. Refer to the "table background" example above to see how these methods are used to call a template in practice.

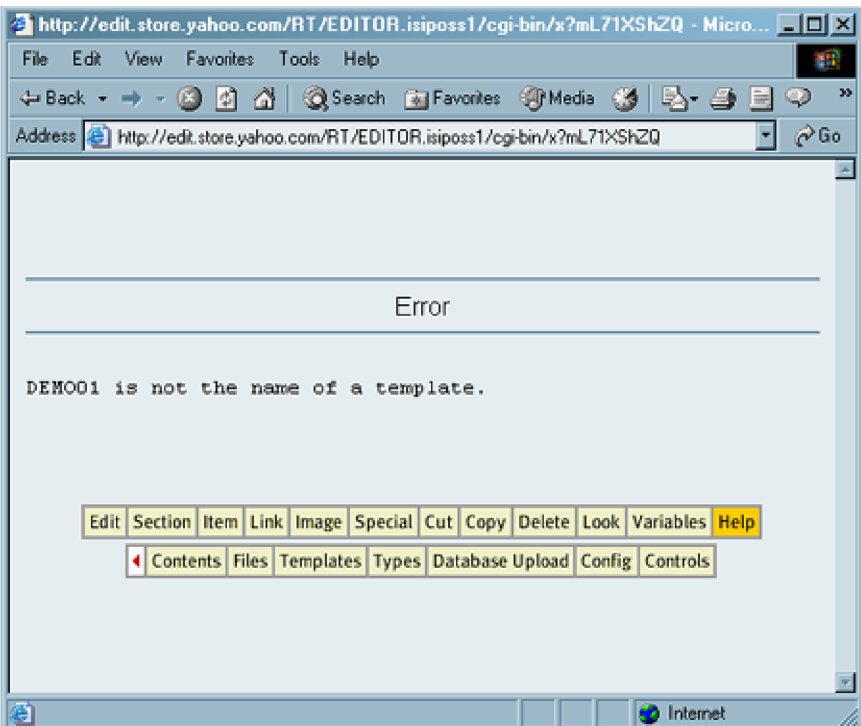


Figure 8 - Non-existent template

Properties and Variables

Like other programming language, RTML allows you to store, retrieve, and work with some values such as numbers or text strings. You can store values in one of three places: in properties of objects (pages), in global, and local variables.

Properties

Whenever you edit a page in your store, you can fill in fields (such as the name and price of the item) or change values from drop-down lists (such as the *Orderable* field). These fields, as mentioned earlier, are called page properties. When a template is executing, it is executing in the context of a page. For each page of your store, when the page is being generated, the Yahoo! Store® system executes the top-level template (the one whose name is entered under the *Template* property) for the page. The top-level template might (and most likely will) call other templates until the page is completed. The page that is being generated is called the current page. In a template, you can reference the properties of the current page by prefixing the name of a property with a @ character as in @name. Consider the following custom template:

Name ()

```
HEAD
  TITLE @name
BODY
  TEXT "The name of this page is "
  TEXT @name
```

If you assign this template to a page and look at the page, it will contain a single line of text announcing the name of the page. If you edit the page and change the *Name* property, this change will immediately be reflected when you

click "Update." One can reference any other property of a page in a similar fashion.

The properties of a page can only be changed manually by editing the page and changing the properties on the edit form. You cannot alter the value of any property within a template using RTML (or any other method for that matter).

Global variables

Global variables are those storewide properties you can set by accessing the Variables page. These variables can be referenced in RTML the same way you can reference the properties of a page by prefixing the name of the referenced variable with a @ character. For an example, look at the built-in *page.* template. The first few lines of this template appear as follows:

```
HEAD
  WHEN VALUE id id
             query :local
             property :keywords
    META name "Keywords"
         content @keywords
  TITLE IF test NONEMPTY @page-title
           then @page-title
           else @name
  TEXT @head-tags
```

Note how the *head-tags* global variable is written to the page by passing it to a `TEXT` operator.

What happens, though, if you override a variable? Well, the trick is that overridden variables "shadow" the global variable of the same name. Suppose you override, say the *head-tags* variable for page *A*. When page *A* is generated, any templates within that context will see the overridden value of the *head-tags* variable. In essence, overridden variables behave like page properties.

As in the case of properties, global variables cannot be altered using RTML. There are only two ways to change the value of a global variable: modify it in the Variables page, or override it within a page.

Local variables

Besides global variables and properties, Yahoo! Store® provides another method to store values: local variables. There are two major differences between global variables and page properties, and local variables:

1. You *can* assign a value to a local variable in RTML, but
2. you *cannot* access a local variable outside of the template where it is used.

When you need to remember a value for some reason within a template, local variables are useful. They are defined by the `WITH=` operator. Consider the following template:

```
Price-difference ()

HEAD
  TITLE "untitled"
BODY
  WITH= variable price
        value ELEMENT position 0
                      sequence @price
    WITH= variable sale-price
          value ELEMENT position 0
                        sequence @sale-price
      TEXT price
      LINEBREAK
      TEXT sale-price
      LINEBREAK
      TEXT price - sale-price
```

While not too useful by itself, this template generates a page with three numbers one below the other: the first number is the price of the item (whose

page is being generated), the second number is the sale price of the item, and the third number is the difference between the price and the sale price (your customer's savings).

Notice at the top how we:

1. Stored the value of the regular price (@price) inside a local variable called *price*.
2. Stored the sale price (@sale-price) inside another local variable called *sale-price*.

In addition, any statement that makes use of the local variables is indented below the `WITH=` operator. The reason for this is the following: the *scope* of a local variable is defined only within the `WITH=` operator that defines it. When you want to refer to a local variable, any statement referring to the local variable must be *pasted within* the `WITH=` operator. To further clarify this rationale, the following template segment would cause an error (in the Old Editor, that is. The New Editor gives no warning but it would not write anything on the screen either):

```
WITH= variable foo
      value 1
TEXT foo
```

whereas

```
WITH= variable foo
      value 1
  TEXT foo
```

would not.

It is easy to see why. In the first example, `TEXT foo` is outside of the `WITH=` block and to the TEXT operator, variable *foo* is undefined. On the other

hand, in the second example, `TEXT foo` is within the `WITH=` block, and there variable *foo* is defined and has a value of 1.

Naming variables

Variable names can contain any letter or number plus any printable character. The following are not acceptable:

- Space
- Double or single quotes
- Left and right parentheses

There might be others that are not allowed either. If you want to use some special character in the name of a variable, try it (in the `WITH=` operator). If your choice is not valid, after clicking "Update", the RTML editor will replace the offending variable name with nil.

You are strongly advised not to use special characters in names. It can lead to confusing code. For example, x+1 is a valid variable name, but imagine using this variable in an expression such as x+1 + 3!

In addition, RTML allows you to start variable names with numbers. In fact, you can even name a variable a number! When you do that, you might have to enclose the number in | characters as in |3| to make it obvious to the RTML interpreter what you mean. Be warned, though, that this is a confusing practice just like using special characters in variable names. Consider the following code:

```
WITH= variable |3|
      value 2
  TEXT |3| + 3
```

Three plus three is six, yet this template would print 5 because the variable |3| contains the value 2!

Nested variables

Look at the savings example on page 49 again. In that example, we used two local variables, price and sale-price. Because a local variable is only "visible" within the `WITH=` operator that defined it, if you need to use more than one local variables (as in the savings example), you have to paste one `WITH=` operator within another. This is an example of nested variables. If you have many local variables, nesting can lead to code that is indented several levels deep. Most of the time, you will rarely use numerous local variables. In most cases, your templates will refer to properties of the current page or to global variables, so don't worry too much about deep indents.

Parameters revisited

Before wrapping up our discussion about local variables, we need to take a brief look at template parameters or arguments one more time. Remember, that each custom template can take zero or more parameters. These are essentially placeholders for values to be passed to the template. Inside the template, each parameter acts just like a local variable. The only difference between a local variable and a parameter is that the **scope of a parameter is the entire template**. For instance, the *border* parameter of the *my-table* template created earlier can be referenced or used anywhere within that template.

Objects

Every page (items, sections, and all other pages) in a Yahoo! Store® is an object. Each object (or page) has:

- A type that determines the properties of the object
- A template that determines how the object (or page) is rendered in the web browser
- An ID, which uniquely identifies the object (or page) within your Yahoo! Store®.

ID revisited

When we introduced the Advanced Interface, we talked about Page ID, but there is more to the story. In many cases in RTML, you need to refer to or find out the ID of an object. RTML provides this information using a special variable called *id*. This variable may be called "special" because it behaves like a regular property in that one can reference it exactly like a regular property; however, its notation (no @ prefix) gives it the appearance of a local variable. To find out the ID of the current object, reference the variable called *id* as illustrated in the following example:

```
TEXT "The id of this page is "
TEXT id
```

When executed, this template segment will always print the name of the ID of the current object.

A word of caution about local variables and special variables such as ID: avoid using the name of a special variable (such as ID) for a local variable. If you were to do this, your local variable will "shadow" the special variable of the

same name. Consequentially, the RTML code might not behave the way you expect it. Consider the following code. It will write two different values for ID!

```
TEXT "The value of ID is "
TEXT id
LINEBREAK
WITH= variable id
      value 12
  TEXT "The value of ID is "
  TEXT id
```

Referring to other objects

When a template is being evaluated, the evaluation always occurs within the context of the current page or current object. The template can always refer to the properties of the current object by simply prefixing the property name with a @ character as in @name or @template. There might be times, however, when you need to refer to an object (or page) other than the current page. To refer to an object, all you have to know is its ID. Once the ID of an object is known, you can refer to it in RTML by prefixing the name with a colon. Here's an example. If you need to make sure the current object is the home page, you would have the following construct (don't worry about the syntax yet):

```
IF test EQUALS value1 id
               value2 :index
  TEXT "This is the home page."
```

Changing context

Although every template is executed within the context of the current object or current page, there are instances where you need to get at the properties of a different object or page. Most computer languages achieve this by allowing the programmer to "qualify" properties. In RTML, on the other hand, the only way to refer to a property is by using it unqualified as in @*name*. For this reason, to

refer to the properties of an object other than the current object, you need to change context to the other object. This can be done using the WITH-OBJECT or the FOR-EACH-OBJECT operator. The WITH-OBJECT operator takes a single parameter, the object whose property or properties you want to access. Any expressions or operators needed to access the other object's properties are pasted *within* the WITH-OBJECT operator. For example, finding out the value of the *Page-title* property of the home page requires the following:

```
WITH-OBJECT :index
  TEXT @page-title
```

First, notice the notation used for the home page: *:index*. A colon prefix is used with the page ID (which in this case is index). Second, all the statements referencing the other object's properties—the TEXT operator in this case—are pasted within and indented below the WITH-OBJECT operator.

One realization is that when you change context within the WITH-OBJECT block all your template "sees" is the other object. Consider the following template segment:

```
TEXT id
LINEBREAK
WITH-OBJECT :index
  TEXT id
```

The first TEXT id line writes the ID of the current page, while the second always writes the ID of the home page, index! Here's why. Within the WITH-OBJECT block, the template is executing within the context of the home page. Consequently, any properties and special properties like ID are understood to be those of the home page.

FOR-EACH-OBJECT also takes a single argument. This argument, however, must be a list of objects or IDs (such as the *@contents* property of a page). FOR-EACH-OBJECT walks through—and changes context to—each list ele-

ment, and every expression pasted ***within*** the `FOR-EACH-OBJECT` block will be evaluated in the context of that element. This probably sounds more complicated than it really is. The following example should clarify it.

Consider the following template segment:

```
WITH-OBJECT :index
  FOR-EACH-OBJECT @contents
    TEXT @name
    LINEBREAK
```

This template segment will first change context to the home page. Next, it will walk through the *contents* property of the home page (because the `FOR-EACH-OBJECT` is indented below the `WITH-OBJECT :index` block and the *FOR-EACH-OBJECT @contents* statement is inside the context of the home page) and output the name of each of the pages below the home page.

Up, Next, and Prev

There are three additional special variables like *id* that need to be mentioned here: *up, next,* and *prev.*. They may be considered special for the same reason *id* was: these three variables are used without the leading @ character just like local variables, but they change value based on the context of the object in which they are referenced.

You might have used the Up and Next buttons in your store. They take you one level higher in the hierarchy of your pages, or, to the next page on the same level. Consider the following sample store structure:

```
index
  Section A
    Item A
    Item B
```

Suppose you had an Up button in your navigation bar and you were looking at Item A in your browser. Clicking the Up button would take you first to Section A. Clicking the Up button again, takes you to the home page. Similarly, suppose you had a Next button in your navigation bar and you were looking at Item A in your browser. Clicking Next takes you to Item B. Clicking Next again, takes you back to Item A again.

The way to find out which page is “Up” or “Next” within a template is by referencing the *up* and *next* variables.

The *up* variable contains the ID of the page *above* the current page. If there is no page above the current page, the value will be *nil* (as in the home page). There are two factors that determine which page is above the current page; or, in other words, which page is the parent page of the current (or any other page).

The first factor is which page contains the current page in its *contents* property. For example, if you edit your home page and look at the *contents* property, its contents will be the IDs of the top-level sections (and perhaps some items that you might have created directly below the home page). For these top-level sections, the parent page is your home page. Consequently, the *up* variable – for these top-level sections – will contain *Index*. Similarly, if you edit any section page and examine its *contents* property, you will see a list of IDs of the pages and/or sections contained within that section. For those contained pages, the *up* variable would have the ID of that parent section.

The second factor comes into play when an item is contained within multiple sections. The *up* variable can have only one ID, but which one? Suppose you have two sections, Section A and Section B. You copy an item, say, Item D, from Section A to Section B. Both of these sections will show Item D; both of them will contain the ID of Item D in their *contents* property, but only one of them will be the actual parent of Item D. If you programmed the scenario described here and were looking at your store's contents list, you would find something like the following:

```
index main. home.
  Contents:
    sectiona item. page.
      Contents:
        itemd item. page.
    sectionb item. page.
      Contents:
```

In this list, *itemd* (the ID of my Item D) only appears once, below *sectiona*. From this list, it is apparent that it is Section A that actually contains Item D; section B simply holds a *reference* to Item D in its *contents* property. Consequently, when looking at the *up* variable in the context of Item D, it would return the ID of Section A. It is also apparent from this picture that **the ownership of a page in case of multiple container sections is determined by which section is first in the contents list**. In the example above, *sectiona* comes before *sectionb*. Accordingly, *itemd* is contained in *sectiona*. Suppose we edited the home page and reversed the order of *sectiona* and *sectionb* within the *contents* property of the home page. The ownership of *itemd* would also change from *sectiana* to *sectionb*.

While the *up* variable contains the ID of the current page's parent page, the *next* variable contains the ID of the current page's next sibling. All pages sharing the same parent are commonly referred to as siblings. By this definition, all the top-level sections of your store are siblings of one another. As a result, for

those sections, the *next* variable would return the ID of the next top-level section in the row.

Finally, *prev* works similarly to next, except it returns the ID of the previous sibling of the current page. For some reason, there is no built-in "Prev" button; still, one could be created by using the *prev* special variable.

Constants

In the previous pages, we saw how one can use properties as well as global and local variables to store and retrieve various values. Some of the values we get back from these properties and variables are simple numbers or text strings. However, there are two cases when we need to work with some special values that are neither numbers nor text strings:

- When dealing with property values that are selectable from a drop-down list (for instance the *Orderable* property of any item), and
- When passing some predefined values to certain operators, such as the alignment option (left, right, center) for the `TABLE` operator.

In both of these cases, the values passed back and forth are constants that are set within RTML.

To refer to a constant, (using the same notation used when referring to objects) prefix the symbol with a colon as in *:left*.

Unfortunately, the online RTML documentation available at Yahoo! ®'s web site says absolutely nothing about constants or when to use them. Here's a reasonable rule of thumb. Whenever referring to values selected from a drop-down list and whenever needing some value that denotes any kind of an attribute (left, right, vertical, horizontal, etc.), use a constant.

Consider the following template segment:

```
IF test EQUALS value1 @page-format
               value2 :top-buttons
   then TEXT "You are using top buttons."
   else TEXT "You are using side buttons."
```

If your *Page-format* variable is set to *Top-buttons*, the template segment above will result in the text "You are using top buttons." If your *Page-format* variable is set to *Side-buttons*, the resulting text will be "You are using side buttons." Notice how we referred to the value of the @page-format variable. We used *:top-buttons* because @page-format is a drop-down list in the variables page.

As mentioned earlier, there is no online documentation on what they are or when to use constants. The following is a partial list of constants based on what can be found in the built-in templates.

Symbol	Explanation
:big	Option of the Price-style variable
:bottom	Bottom (alignment constant)
:center	Center (alignment constant)
:contents	Option of the Nav-buttons variable
:download	Option of the Nav-buttons variable
:email	Option of the Nav-buttons variable
:empty	Option of the Nav-buttons variable
:empty.	Built-in type "empty."
:fixed	Option of the Column-width variable
:help	Option of the Nav-buttons variable
:home	Option of the Nav-buttons variable
:horizontal	Built-in constant used with the FUSE operator

:icon	Option of the Button-style variable
:image	Option of the Nav-buttons variable
:incised	“Incised” selection for the Button-style variable
:index	Option of the Nav-buttons variable
:info	Option of the Nav-buttons variable
:info.	Built-in type “info.”
:item.	Built-in type “item.”
:left	Left (alignment constant)
:link.	Built-in type “link.”
:main.	Built-in type “main.”
:mall	Option of the Nav-buttons variable
:multi-line	Option of the Order-style variable
:next	Option of the Nav-buttons variable
:no	No
:norder.	Built-in type “norder.”
:normal	Option of the Price-style variable
:privacypolicy	Option of the Nav-buttons variable
:privacypolicy.	Built-in type “privacypolicy.”
:quiet	Option of the Price-style variable
:raw-html.	Built-in type “raw-html.”
:register	Option of the Nav-buttons variable
:request	Option of the Nav-buttons variable
:right	Right (alignment constant)
:section.	Built-in type “section.”
:show-order	Option of the Nav-buttons variable
:solid	“Solid” selection for the Button-Style variable
:search	Option of the Nav-buttons variable
:search.	Built-in type “search.”

:side-buttons	Option of the Page-format variable
:solid	Option of the Button-style variable
:t	Logical value True. See the section on Logical Expressions for more information.
:text	Option of the Button-style variable
:top	Top (alignment constant)
:top-buttons	Option of the Page-format variable
:two-line	Option of the Order-style variable
:up	Option of the Nav-buttons variable
:yes	Yes
:variable	Option of the Column-width variable
:vertical	Built-in constant used with the FUSE operator

Expressions

Each line of an RTML template is an expression. An expression may be simple such as 82, or complex. Complex expressions consist of an operator and zero or more arguments. The expression `TEXT "this is one line"` is a complex expression, which consists of the `TEXT` operator and a single argument; `"this is one line"`. Additionally, each expression returns a single value of some type. The expression `1 + 2`, for example, returns a number (3), the expression `"ABC"` returns the text string "ABC", and the expression `RENDER @name-image` returns an image (the value of the name-image variable of your store).

Logical expressions

Logical expressions always return one of two values: true or false. In RTML, the logical value *true* is represented by the constant *:t*, while the logical value *false*, by *nil*. In addition, any global variable or property that can be set to either "Yes" or "No" (such as the *orderable* property) can be treated as a logical value with "Yes" meaning true, and "No" meaning false.

Finally, anything whose value is other than *nil* can also be treated as a logical *true* value. This is a familiar concept to those who know the C language. We will cover the use of logical values in detail in the section about Conditionals.

There are three RTML operators that can be used to manipulate logical expressions: AND, OR, and NOT. These are discussed next.

And

`AND` takes one or more arguments. Each argument can be any valid RTML expression pasted ***within*** the body of `AND`. The operator then evaluates each of

those expressions, and if all are true (none are nil), it returns the value of the last expression.

Consider the following template segment:

```
IF test AND
          @taxable
          @orderable
   then TEXT "This item is orderable and taxable."
   else TEXT "This item is either not orderable or not taxable."
```

This example prints, “This item is orderable and taxable.” if both the *orderable* and *taxable* properties of the current page are set to “Yes.”

If any of the expressions within the AND block are nil (false), then the rest of the expressions are ignored.

Or

OR takes one or more arguments (pasted ***within*** it)—each being a valid RTML expression—and returns the value of the first one that is other than nil. Once it finds an expression whose value is other than nil, the rest of the expressions are ignored.

If either the *orderable* or the *taxable* (or both) properties of the current page is set to “Yes,” the following example will print, “This item is orderable or taxable.”

```
IF test OR
          @taxable
          @orderable
   then TEXT "This item is orderable or taxable."
   else TEXT "This item is neither orderable nor taxable."
```

The fact that the first non-nil value of the OR operator is returned and the rest ignored is important. Based on this fact, we can write expressions such as this:

```
WITH= variable price
      value OR
              @sale-price
              @price
```

In this example, the local variable *price* will be set to the value of the *Sale-price* property, if *Sale-price* is not empty, otherwise, to the value of the *Price* property. Notice, that the above example is NOT equivalent to this:

```
WITH= variable price
      value OR
              @price
              @sale-price
```

Here, `OR` returns the value of its first non-nil expression. This example will almost always set the local variable *price* to the value of the regular price of the current item (unless you forgot to enter the regular price but not the sale price).

Not

This operator returns the logical opposite of its argument. It takes a single RTML expression as its argument. If the expression returns nil, `NOT` returns true (the *logical* opposite of its argument). If the expression returns a value other than *nil,* `NOT` returns false.

Control structures

In a programming language such as RTML, control structures are constructs that allow us to control the flow of execution within a program. They typically fall into two groups: conditionals and iteration or loop structures. RTML offers both and in the following pages, we are going to explore each.

Making decisions: Conditionals

One of the strengths of RTML is that it can be used to create templates that produce different HTML pages based on conditions such as the state of a variable or property. To execute a different set of commands based on some condition, we need some mechanism to make decisions. This mechanism is called a conditional statement, or briefly, a conditional. RTML offers three kinds of conditionals, the “If”, “When” and “Switch” structures.

The IF operator

The `IF` operator lets us execute one set of RTML expressions if some condition is true (or not nil) and another set of expressions if the same condition is false.

Note: It is best for you to get into the habit of thinking of a logical *true* value as one that is other than *nil*.

The `IF` construct is probably the most powerful construct of any programming language. Once you have the means to execute statements conditionally, you can create a program to do just about anything.

The IF operator has three arguments: a *test* expression, a *then* expression, and an *else* expression.

The *test* expression is an expression that is evaluated and depending on its result, either the *then* or the *else* expression is evaluated. If the result of the *test* expression is not nil, the evaluation continues to the *then* expression, otherwise, to the *else* expression. The result of the entire IF expression is the result of either the *then* or the *else* expression.

To clarify this operator, turn the page back to the discussion of the OR operator. To demonstrate the use of the IF operator, look at the following example again:

```
IF test OR
         @taxable
         @orderable
   then TEXT "This item is orderable or taxable."
   else TEXT "This item is neither orderable nor taxable."
```

Here, the *test* expression is the OR block. If the result of the OR block is not nil, the *then* expression is evaluated (the line "This item is orderable or taxable." is printed), otherwise, the *else* expression is executed (and the line "This item is neither orderable nor taxable." is printed).

Remember that every RTML expression returns a value. The IF operator is not an exception. As mentioned previously, the IF operator returns the value of whichever expression (the *then* or the *else* expression) it evaluates based on the result of the *test* expression. Knowing that, you could rewrite the above example like this:

```
TEXT IF test OR
              @taxable
              @orderable
        then "This item is orderable or taxable."
        else "This item is neither orderable nor taxable."
```

What happens here is the `TEXT` operator will print the result of the `IF` operator, which will either be, "This item is orderable or taxable.", if either @taxable or @orderable is **true;** or, "This item is neither orderable nor taxable.", if both @taxable and @orderable are false.

Finally, while all three of the `IF` arguments, *test*, *then*, and *else*, can be very complex, valid RTML expressions, each of those arguments can contain only one operator. This might appear to be a contradictory statement, but it is not. There are two ways to enter complex expressions for *test*, *then*, or *else*. They are:

- Use the `CALL` operator to evaluate another template, which could possibly do something complex.
- Use the `MULTI` operator. The `MULTI` operator groups together multiple expressions pasted ***within*** its body and returns the result of the last such expression. For an example of a complex use of the `IF` structure, look at the **Element-image-cell.** built-in template.

The WHEN operator

The `WHEN` operator similar to the `IF` operator: it is basically "one half" of the `IF` operator. It says, "evaluate the following if this expression is true (not nil)" and is equivalent to the following `IF` block:

```
IF test <some expression>
   then <some other expression>
   else nil
```

The `WHEN` operator has a single argument, a condition. If the result of the condition is true (not nil) then the expression or expressions pasted ***within*** the `WHEN` block is/are evaluated. Similar to the `IF` operator arguments, the `WHEN` operator's condition argument can only be a single operator. However, the

WHEN operator may contain any simple or complex expression. (Use the CALL or MULTI operators to enter a more complex expression).

For an example on how to use the WHEN operator, consider the following template snippet:

```
WHEN @sale-price
  TEXT "This item is on sale."
```

If the current page has a sale price entered, the code above will print: “This item is on sale.” Otherwise, it will do nothing.

The SWITCH operator

The SWITCH operator is very much like a multi-state switch (hence the name). It takes one argument, the *switch expression*, and zero or more *key-expression pairs*. It then compares the result of the *switch expression* to each key. If there is a match, the corresponding expression is evaluated and its value returned. If there is no match, SWITCH returns nil. It is much easier to understand how SWITCH works by looking at an example:

```
SWITCH @page-format
  :top-buttons
  TEXT "You have top buttons."
  :side-buttons
  TEXT "You have side buttons."
```

In this example, the *switch expression* is the global variable @page-format. There are two *key-expression pairs*, one starting with :top-buttons, the other, with :side-buttons. The operator will compare the value of @page-format first to the constant *:top-buttons* (because @page-format is a variable that has values from a drop-down list, we must use the constant notation for those values). If our Page-format variable is set to “Top-buttons” then this template segment will

print "You have top buttons." If, on the other hand, our Page-format variable is set to "Side-buttons", then the template will print "You have side buttons."

Entering a `SWITCH` block into a template can be a bit strange at first, so let's step through the example above. Begin by clicking New. Select `SWITCH` from the operator list and click "Create." This will put a `SWITCH` operator into the expression stack. Paste it inside your template. Next, click Edit, and enter *@page-format* as the switch operator. When done, Click "Update". Enter the first key. Click New again, type *:top-buttons* into the box labeled "Simple," and click "Create." Because `SWITCH` is still selected in the template, you can click Paste Within. Click New, select `TEXT` from the operator list, and click "Create." Now *:top-buttons* is selected, and because you want the `TEXT` operator to go after *:top-buttons*, click Paste After. With `TEXT` selected, click Edit, enter "You have top buttons." and click "Update." Great. We have the first key-expression pair in place.

To enter the second, click New, enter *:side-buttons* into the box labeled "Simple," and click "Create." `TEXT` is still selected after our last operation, so click Paste After. Click New, select `TEXT` from the operator list, and click "Create." With *:side-buttons* still selected, click Paste After. Now, since the newly inserted `TEXT` operator becomes the current (selected) element, you can click Edit, enter "You have side buttons." and click "Update." That's it.

Repeating actions: Iteration

Any time you need to repeat an action many times, you can do it by using some sort of a loop or iteration structure. RTML is very rich in iteration structures. It offers the following five:

- For loops
- For-Each
- For-Each-Object
- For-Each-But
- Find-One

In the following pages, we are going to explore each.

For loops

Every programming language has looping structures. The `FOR` command is one used most frequently, and RTML is no different.

The `FOR` operator in RTML has the following four arguments:

1. A variable called the *control variable*
2. An *initial value* (can be any valid RTML expression)
3. A *test condition* (can be any valid RTML expression)
4. An *update operation* (can be any valid RTML expression)

In theory, here's how the `FOR` operator works in RTML. It takes the *control variable* and assigns it *initial value.* It then evaluates *test condition*. If *test condition* is false (nil), then the `FOR` operator returns the current value of the *control variable*. If *test condition* is true, then the expression pasted ***within*** the `FOR`

block is evaluated (executed). Finally, the update operation is evaluated and the whole cycle starts again. It repeats until *test condition* is false (nil).

In practice, the `FOR` operator is used to execute some expressions a given number of times. Below is the RTML equivalent of a classic BASIC program, one that prints the numbers from 1 to 10:

```
FOR variable I
    initial 1
    test <= value1 I
            value2 10
    update i + 1
  TEXT I
  LINEBREAK
```

Here, *control variable* is i, initial value is 1, *test condition* is i <= 10, *update operation* is i + 1, and the body of the `FOR` loop is the `TEXT` and `LINEBREAK` operators. When executed, this template snippet will print the numbers from 1 to 10 one below the other. Here is how it works: it assigns the value 1 to *i*. Until *i* is greater than 10, it prints the current value of *i* followed by a line break and increments *i* by 1.

By now, you might expect that as any other operator in RTML, `FOR` returns a value as well. The value returned by `FOR` is the last value of the control variable. So, in the above example it would be 10, right? **Wrong,** it would be 11. Here's why.

When *i* is 10, test condition is still true (since *i* is less than or equal to 10), so it prints *i* followed by a line break. Then it executes the *update operation* again: it adds one to *i*. Now *i* becomes 11, *test condition* becomes false (since *i* is now greater than 10), the `FOR` operator terminates, and returns the last value of *i*, 11. `FOR`, therefore, returns the last value of its control variable, or in other words, the first value for which the test condition is false. This is easy to test. If you modify the above example by cutting the entire `FOR` block and pasting it in as the argument of a `TEXT` operator (as in the example below), the output will

be the numbers from 1 to 11 (one below the other). The first then are printed by the FOR loop itself as before. The last one is the result of the FOR operator (the last value of the control variable).

```
TEXT FOR variable I
         initial 1
         test <= value1 I
                 value2 10
         update i + 1
       TEXT I
       LINEBREAK
```

Before finishing with the FOR operator, let's look at two small but important details.

- You might be tempted to type something like "i <= 10" for *test condition* especially since when you edit a FOR operator, there is a text box next to *test* (see Figure 9). You can't do that. Instead, you should paste a more complex expression in place of the test condition. We used the complex <= operator in the above example.
- While some programming languages evaluate the test condition of their FOR commands *after* each cycle, others (and RTML is among these) evaluate the test condition *before* each cycle. Those that test *after* each cycle execute at least one cycle guaranteed (since they don't check the test condition until the completion of the first cycle). Those that test *before* each cycle might not execute a single loop at all (e.g., if the test condition is false to begin with). To better understand what is meant here, change the test condition of our example (the one that prints the numbers from 1 to 10) to this:

```
test <= value1 i
        value2 0
```

When the template executes, it will print nothing at all. The test condition (i <= 0) will be false from the beginning.

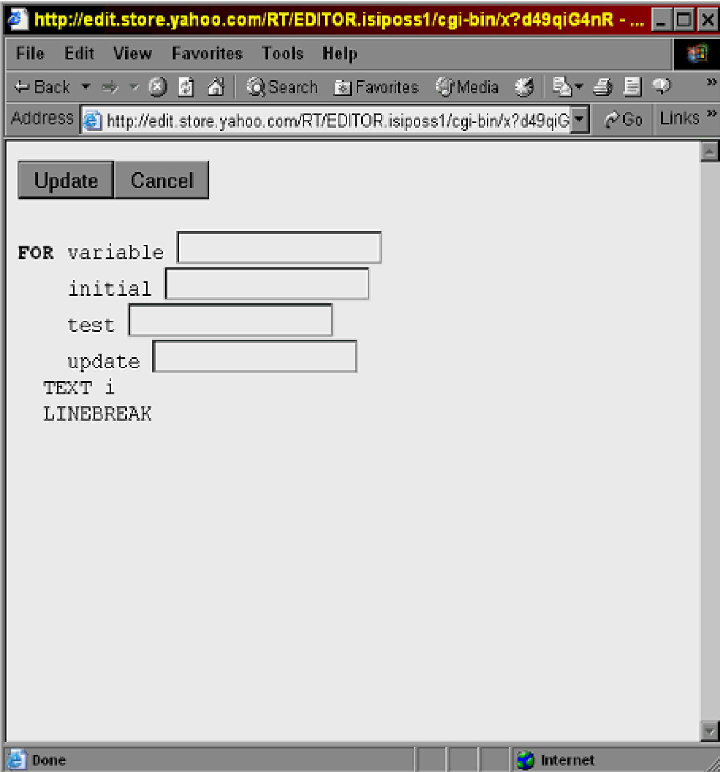


Figure 9 - FOR operator parameters

For-Each

`FOR-EACH` takes a variable and a sequence. It assigns each element of the sequence to the variable, one after the other. For each element, it evaluates the expression pasted ***within*** the `FOR-EACH` block. We will talk about sequences in detail later. For now, sequences can be text strings such as "abcde", a sequence of letters, or variables such as the *contents* property of a page (which is a

sequence of IDs). The following example will print the letters from a to z, one on each line:

```
FOR-EACH variable x
         sequence "abcdefghijklmnopqrstuvwxyz"
  TEXT x
  LINEBREAK
```

Here's how it works. It takes the first element of the sequence "abcdefghijklmnopqrstuvwxyz" (which is, of course, a), assigns it to the variable x, and evaluates the expressions pasted within the `FOR-EACH` bock. In this case, it does the following:

1. Prints the current value of x and outputs a line break.
2. Goes back and takes the next element of the sequence (b).
3. Evaluates the expression block.
4. Takes the next element.
5. Does this until the end of the sequence is reached.

For-Each-But

`FOR-EACH-BUT` is very similar to `FOR-EACH`. It takes a variable, a sequence, and a *last expression*. It evaluates the expression or expressions pasted *within* for every element of the sequence, but—and here is where it differs from `FOR-EACH`. It also evaluates *last expression* for each element ***except*** for the last one.

The following example will print the letters a, b, c, and d separated by commas, and there will be no comma after d:

```
FOR-EACH-BUT variable x
             sequence "abcd"
             last TEXT ","
  TEXT x
```

This example works as follows:

1. x is assigned consecutive elements of the sequence “abdc.”
2. It begins with assigning the element “a” to the variable x.
3. It evaluates the expression pasted within the FOR-EACH-BUT operator (in this case, printing out the current value of x, “a”).
4. “a” is not the last element of the sequence “abcd,” so it evaluates *last* expression (which, in this case prints out a comma).
5. Takes the next element of the sequence “abcd” and repeats the whole process.

It will keep doing this until the last element, “d”, is reached. At that point, the expression `TEXT x` will still be evaluated (so “d” will be printed). However, the *last expression*, which would print a comma, will be skipped, because “d” is the last element of the sequence “abcd.”

It is worth mentioning that both the *sequence* and *last* arguments of `FOR-EACH-BUT` can be any valid RTML expressions (even very complex ones). If you use a complex expression for *sequence*, make sure that the expression you use returns a sequence. Again, we will deal with sequences in more detail later

For-Each-Object

We discussed `FOR-EACH-OBJECT` in the section about Objects on page 54. It is one of the iteration structures in RTML and needed mentioning again here

Find-One

FIND-ONE takes two arguments: a *control variable* and a *sequence*. It then assigns consecutive elements of *sequence* to *control variable* and evaluates the expression pasted ***within*** its body. If the expression is true (not nil) then the current value of *control variable* is returned. If the expression is false for every element of *sequence*, then FIND-ONE returns nil.

Let's see how FIND-ONE works on a rather mindless example. This example will find the search page in your store and print out its ID.

Begin with pasting a TEXT operator into a template and then replacing the argument of TEXT with a FIND-ONE operator. At this point, your template should look something like this:

```
TEXT FIND-ONE variable nil
              sequence nil
```

Edit FIND-ONE, enter x for variable, and click "Update." Now click New, select WHOLE-CONTENTS from the operator list and click "Create." Click nil next to sequence in your template and click Replace. Your template should now look like this:

```
TEXT FIND-ONE variable x
              sequence WHOLE-CONTENTS
```

Click New again, select `EQUALS` from the operator list, and click "Update." Click `FIND-ONE` in your template, and click Paste Within. With `EQUALS` selected, click Edit, and type *x* for *value1*, and *:nsearch* for *value2*. Click "Update" when done. The final version of the template should look like this:

```
TEXT FIND-ONE variable x
              sequence WHOLE-CONTENTS
       EQUALS value1 x
              value2 :nsearch
```

When this template is evaluated, the screen will show the word NSEARCH (the ID of the search page). Not very useful but it still shows you how `FIND-ONE` works: it takes the first element of *sequence*, which in our case is `WHOLE-CONTENTS`. `WHOLE-CONTENTS` is an operator, which returns the sequence of all the pages in your store. The ID of the first page of your store is assigned to x and then the expression pasted within `FIND-ONE` is evaluated. Here, the expression checks whether the variable x is equal to the object *:nsearch* (the search page). If not, x is assigned the second page of your store and the expression is evaluated again. This goes on until x contains *:nsearch*. At that point, the current value of x is returned as the result of `FIND-ONE`. Since it was a `TEXT` operator that evaluated our `FIND-ONE` operator, this `TEXT` operator receives the result of `FIND-ONE` and prints it.

`FIND-ONE` is a very powerful operator. It can be used for much more than what we've shown here.

Let's finish this discussion with another example. We don't want to explain everything seen here; you will learn all of it in due course. What we want to show you for now is what you can do with `FIND-ONE`.

The following example will find the first item in your store whose price is greater than 100 and print its name.

```
WITH= variable over100
      value FIND-ONE variable x
                     sequence WHOLE-CONTENTS
              WITH-OBJECT x
                WITH= variable price
                      value ELEMENT position 0
                                    sequence OR
                                               @sale-price
                                               @price
                  WHEN price
                    > value1 price
                      value2 100
  WHEN over100
    WITH-OBJECT over100
      TEXT @name
```

Part II. Working with RTML

Sequences

Sequences are a fundamental part of Yahoo! Store®. You will run across them all the time. They include:

- Any text string.
- Lists such as the *contents* property of a page (which is a sequence of objects).
- The *specials* property of the home page (which is a sequence of page IDs).
- Paragraphs and lines of texts.

Because sequences are so common, RTML offers an extensive set of operators for handling them. In this section, we are going to explore these operators.

Elements

The `ELEMENTS` operator returns a **sub**sequence of a sequence. It takes three parameters: a *sequence,* a *start index*, and an *end index.* It returns a **sub**sequence consisting of the elements of *sequence* from *start index* to *end index*. The following example, for instance, will print "bcd":

```
TEXT ELEMENTS sequence "abcde"
              first 1
              last 3
```

From this example, you can see that the numbering of sequence elements is zero-based, so the letter "a" in the sequence "abcde" is at position 0.

The second and third parameters of `ELEMENTS` are optional. If *first* is omitted, it is assumed to be 0 (meaning the start of the sequence). If *last* is omitted, it is assumed to be the last element of the sequence. Consequently, if both *first* and

last are omitted, then the ELEMENTS operator simply returns the *sequence* itself.

Element

ELEMENT takes two parameters, a *position* and a *sequence*. It then returns the element of sequence *sequence* at position *position*. As in the case of the ELEMENTS operator, the numbering of sequence elements starts at zero. If sequence *sequence* has no element at position *position*, ELEMENT returns nil. The following example will print the first letter of every special in your store:

```
WITH-OBJECT :index
  FOR-EACH-OBJECT @specials
    TEXT ELEMENT position 0
                 sequence @name
    LINEBREAK
```

Length

If you use the ELEMENT or ELEMENTS operator, you might need to find out how many elements there are in a sequence. You can do that with the LENGTH operator. The length operator takes a single parameter, a *sequence*, and returns the number of elements within the sequence. If the sequence is empty, LENGTH returns 0.

Remember, that all operators that work on sequences and refer to particular positions within a sequence (such as the ELEMENT or ELEMENTS operator) consider element 0 to be the first element within a sequence. Therefore, when you use these operators in conjunction with the LENGTH operator, you should know that the last element of a sequence is at the position returned by LENGTH minus one.

To demonstrate this, consider the following example:

```
WITH-OBJECT :index
  FOR-EACH-OBJECT @specials
    WITH= variable len
          value LENGTH @name
      TEXT ELEMENT position len - 1
                   sequence @name
    LINEBREAK
```

This example is very similar to last one where we printed the first letter of each special item, except in this case; we are printing the ***last*** letter of every special item. For each special, we first store the length of the special's name in the local variable *len*, and then we print the element of the name at position *len - 1*.

Nonempty

Returns true if a text string contains at least one non-whitespace character. Whitespace characters include the space character, the tab character, and a new line (carriage return) character. `NONEMPTY` takes a single argument, a text string.

Position

`POSITION` takes two arguments: an element and a sequence. It then returns the position at which the sequence contains the specified element or nil, if the element was not found in the sequence. This operator is most commonly used to check if an element exists within a sequence. The following example demonstrates this use:

```
WHEN POSITION element :contents
              sequence @nav-buttons
  TEXT "Contents are part of Nav-buttons."
```

If "Contents" is specified in the *Nav-buttons* variable, then this template prints "Contents are part of Nav-buttons."

This example also shows you that the global variable *Nav-buttons*, the home page properties *Page-elements* and *Buttons*, as well as any variables and properties that allow you to make multiple selections (*contents-elements*, for example), are considered sequences.

Segments

The SEGMENTS operator takes a sequence, and returns successive segments of a specified length of that sequence. Consider the built-in template **Pack-contents**. This template is used to show the *contents* of a page if *contents-format* is set to Pack. For your reference, the template is included below:

```
Pack-contents. (ids)
FOR-EACH variable tuple
         sequence SEGMENTS length @columns
                           sequence ids
  FOR-EACH-OBJECT tuple
    WITH-LINK TO id
      IMAGE source RENDER image CALL :shown-image.
            alt @name
  LINEBREAK
```

This template takes a sequence of IDs, those included in the *contents* property of a page. Visualize how contents are rendered on the page when contents format is set to Pack and you will understand how SEGMENTS is used. Contents are generated on the page arranged into a number of columns as set by the *columns* global variable. If the *columns* global variable is set to 3, for instance, and a page has, say, 6 items in its *contents* property, then the first row will contain the images (or icons) of the first three items, and the second row will contain the second three items. Basically, what you need to do is to break apart the *contents*

property into subsets of three—each subset being a sequence. This is exactly what is happening in the **Pack-contents.** template.

The first `FOR-EACH` operator—we can call it the "outer loop—takes each subset of the *contents*. These subsets are returned from the `SEGMENTS` operator. The SEGMENTS operator takes the IDs (from the *contents* property) and returns subsets of the IDs each having a length specified by the *columns* global variable.

The second `FOR-EACH` operator—the "inner loop"—walks through each of these segments, and displays the image for each object pointed to by the IDs contained within the segments.

If the last segment doesn't contain enough elements (fewer than the number specified by the *length* parameter), it will simply contain however many elements are left from the original sequence. For example, if the sequence has seven elements and we want subsets of three elements each, `SEGMENTS` will return three sub-sequences, the first two containing three elements each, while the last sub-sequence, only one element.

Tokens

The `TOKENS` operator takes a text string and turns it into a sequence in which each element is a "token" from the original string. Tokens are either single words or phrases enclosed in double quotes and separated by spaces from one another. The following example will print each word of the sentence "This is how TOKENS works" on a new line:

```
FOR-EACH variable word
         sequence TOKENS "This is how TOKENS works"
  TEXT word
  LINEBREAK
```

At the time this book was written, the only place where the `TOKENS` operator was used in the built-in templates was in the **Order.** template. The **Order.** template is responsible for displaying the price, the options—if there are any—and the order button for each orderable item. In this template, `TOKENS` is used to convert each line of the *options* property into a sequence, so that the second word of each line of the *options* property can be examined whether it includes the word “Monogram” or “Inscription.”

For your reference, we included the relevant section of the **Order.** template below:

```
FOR-EACH variable para
         sequence PARAGRAPHS @options
  WHEN EQUALS value1 @order-style
              value2 :multi-line
    LINEBREAK
  WITH= variable set
        value TOKENS para
    IF test AND
              EQUALS value1 ELEMENT position 0
                                    sequence set
                     value2 "Monogram"
              EQUALS value1 LENGTH set
                     value2 1
       then CALL :monogram.
       else IF test AND
                      > value1 LENGTH set
                        value2 2
                      EQUALS value1 ELEMENT position 1
                                            sequence set
                             value2 "Inscription"
               then CALL :inscription.
                      set
               else MULTI
                      TEXT ELEMENT position 0
                                   sequence set
                      TEXT ": "
                      SELECT name ELEMENT position 0
                                          sequence set
                             options ELEMENTS sequence set
                                              first 1
    TEXT " "
```

Yank

YANK has two parameters: a sequence and an element. It returns the same sequence but with all occurrences of the given element removed. Below is a modified version of the template we used to demonstrate the use of the TOKENS operator. This modified version prints each word of the sentence “This is how TOKENS work” but with the word TOKENS removed.

```
  FOR-EACH variable word
           sequence YANK element "TOKENS"
                         sequence TOKENS "This is how
TOKENS works"
    TEXT word
    LINEBREAK
```

Reverse

The REVERSE operator takes a sequence as its argument and returns the same sequence in reverse order.

Whole-Contents

This operator returns a sequence consisting of the IDs of all objects (pages) in the store in alphabetical order. Right now, only the built-in **Index-body.** template uses this operator to generate the index page of your store. The WHOLE-CONTENTS operator is very useful and is the only way to obtain a list of all the pages in your store.

Make-List and Append

These two operators are only present in the New Editor. Make list takes any number of values pasted within it and returns a sequence consisting of all those values.

Append takes any number of *sequences* (except text strings) pasted within it and returns a new sequence by joining all the sequences together.

Working with text

A typical web site—and as such, a typical Yahoo! Store®—consists of some graphics and lots of text. The following pages will deal with the various operators RTML offers for manipulating text strings.

Printing text

Printing is probably the most fundamental operation you will do with text. There is not much to it in RTML. Having followed the examples in this book up to this point, you already know how to print text. Enter the text as the parameter of the `TEXT` operator and you are done. Text can either be a text string entered directly within double quotes, or the value of a *text* or *big-text* type property. Examples of these properties are *name* or *head-tags*.

Text strings printed by the `TEXT` operator may contain HTML tags. As the `TEXT` operator sends its parameter verbatim to the browser, any such HTML tags will be rendered by the browser according to the rules of HTML. Therefore, one can use the `TEXT` operator to create any complex formatting including JavaScript (or VBScript for Internet Explorer).

Printing HTML tags using TEXT in the old editor

Since the RTML editor is a web page itself, one must be careful when entering html tags directly. Any HTML tag entered directly will be rendered on the page inside the listing of your template. Take for example the template shown in Figure 10. Here, we entered the HTML tag <HR> as the parameter of the `TEXT` operator. As you can see in the listing of the template, the string <HR> is not visible. Instead, the RTML editor rendered a horizontal line (which is exactly what <HR> is supposed to do). The <HR> tag is quite harmless, but if you make

a mistake in your HTML tags, you might end up with a template that's very hard or (in some cases) impossible to repair. Netscape, for instance, is very unforgiving when it comes to mistakes in HTML. If you enter, for instance, the following template segment, a page that uses this template will be uneditable in some versions of Netscape:

```
TEXT "<table>"
TEXT "<tr>"
TEXT "<td>"
```

This situation can be avoided. Whenever you need to use HTML tags with the TEXT operator, always cut the HTML tags into various parts as shown below:

```
TEXT "<"
TEXT "table>"
```

These two lines will still yield the HTML tag <TABLE>, however, since the tag itself will not be rendered by the RTML editor, the listing will always be correct.

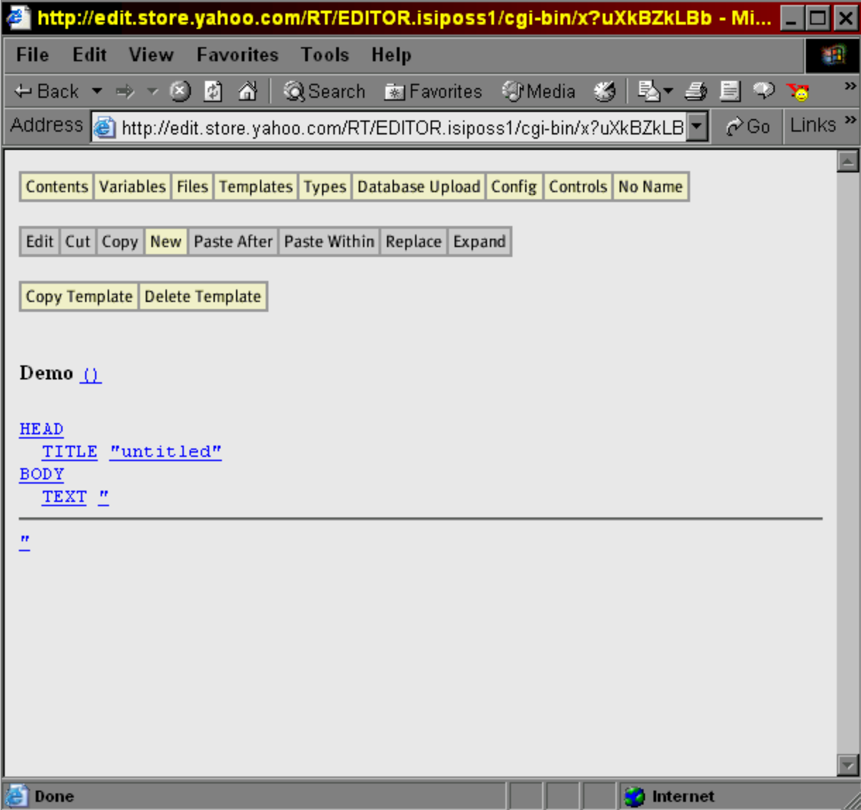


Figure 10 - HTML tags used with the TEXT operator

Concatenating text strings

There will probably be times when you need to concatenate or "glue" text strings together. While most programming languages understand statements like "string1 + string2", RTML does not have a separate operator for this purpose. With a little trick, on the other hand, it is possible to concatenate strings. The key is the `GRAB` operator. The `GRAB` operator takes the output of all expressions within its body and returns them as a string. Using the `GRAB` operator, you can

concatenate strings by printing those strings out inside the body of the GRAB operator like this:

```
WITH= variable concat
      value GRAB
              TEXT "this is one string "
              TEXT "this is another string "
  TEXT concat
```

Here we use the GRAB operator to concatenate the two strings “this is one string “ and “this is another string”, store the concatenated string in the local variable *concat*, and print the content of the *concat* variable using the TEXT operator.

Working with paragraphs

Printing out text strings verbatim is handy, but in many cases, we need to work with paragraphs, blocks of texts separated by blank lines. One of the best-known examples where paragraphs are needed is in interpreting lines of multiple options. Recall, that to enter multiple options for an item, you need to separate those options with blank lines so that the template (**Order.**) in RTML can interpret the options line by line. RTML provides the PARAGRAPHS operator to separate a text string into paragraphs. The PARAGRAPHS operator takes a single parameter, a source text string, and returns a sequence of text strings consisting of the paragraphs of the source text string. The following example takes the *Message* property of the home page and prints each paragraph of it on a separate line.

```
WITH-OBJECT :index
  FOR-EACH variable para
           sequence PARAGRAPHS @message
    TEXT para
    LINEBREAK
```

Working with lines

As each text string may consist of several paragraphs, each paragraph may also consist of several lines. The lines of a paragraph are separated by new line characters (but not with blank lines). To obtain a sequence of the lines of a paragraph, use the `LINES` operator. The `LINES` operator takes a single argument, a text string, and returns a sequence of text strings consisting of the lines of the original. The following example will take the *Message* property of the home page and print each line of text on a separate line.

```
WITH-OBJECT :index
  FOR-EACH variable line
           sequence LINES @message
    TEXT line
    LINEBREAK
```

Working with numbers

Numbers as well as text are used quite extensively in templates. We use numbers to figure out the spacing of items on a page, or display the price of the products. While you cannot perform complicated mathematical calculations in RTML, the language does provide some basic methods allowing you to manipulate numerical data.

Performing calculations

RTML can perform basic arithmetic calculations such as adding, subtracting, multiplying and dividing numbers. Calculations can be performed on numbers entered manually as in

```
TEXT 2 + 3
```

or on variables (local or global) or properties containing numbers as in

```
TEXT @page-width / 2
```

The result of a numerical calculation can be stored in a local variable, and that local variable can then be used in another arithmetic operation:

```
WITH= variable x
      value @page-width / 2
  TEXT x + 3
```

When entering arithmetic operations, always use a space before and after the arithmetic operators. Because arithmetic operators (+, -, /, *) can be used in variable names, omitting a space between a variable name and an arithmetic operator makes it part of the name. Along these lines, x+1 and x + 1 are completely different. While x+1 is a variable name, x + 1 means adding one to the value of the local variable x.

The order in which arithmetic operations are evaluated is determined by the rules of algebra. According to these rules, multiplication (*) and division (/) are "stronger" than addition (+) and subtraction (-), so multiplication and division are always evaluated before addition and subtraction. In addition, the two "stronger" operations have the same "strength" and so do the two "weaker" operations, so more than one multiplication in a row, for example, are evaluated in order. When you mix arithmetic operations of different "strength", the RTML editor automatically inserts parentheses into your expression to disambiguate the order of evaluation. If you try to enter, for instance, the following expression: 2 + 3 * 2, the editor automatically converts this into: 2 + (3 * 2). This is consistent with the order in which these operations should be evaluated (first the multiplication, then the addition.)

Comparing numbers

One number can be less or greater than another, or they may be equal. In RTML, you can make decisions based on how numbers or numerical values (variables or properties) relate to each other by using comparison operators. Why would you want to compare numbers? Well, let's say a section page contains only one item; you might want to format that page differently then if it contains 2 or more items.

RTML has five comparison operators: <, >, <=, >=, and `EQUALS`. All five of these operators take two numeric parameters, the numbers to compare. They return True (:t) if the comparison is valid, or false (nil) otherwise. The < operator returns true, if the first parameter is less than the second parameter. The > operator returns true, if the first parameter is greater than the second parameter. The <= operator returns true, if the first number is less than or equal to the second number. The >= operator returns true, if the first number is greater than or

equal to the second number. Finally, the `EQUALS` operator returns true if the two numbers are equal.

> **Note**: The `EQUALS` operator can also be used with entities other than numbers. It can be used, among other things, to determine if two strings are the same.

While there is no separate operator for inequality (<> in some languages), in RTML you can test if two numbers are not equal by combining the `EQUALS` operator with the `NOT` operator as in:

```
NOT EQUALS value1 1
           value2 2
```

Working with prices

Although prices are just numbers, they require special attention. You might have tried to figure out the customer's savings by subtracting the sale price from the regular price of an item only to find out that RTML won't let you do that. If you try to evaluate the following: @price - @sale-price, it will return an error telling you that the argument of the – operator must be a number. So, what is wrong? Aren't @price and @sale-price numbers? Of course not! Think about how prices are entered: typically, you would enter a single number, the price of the item. However, Yahoo! Store® also allows you to enter what they call "quantity pricing" (where you give a discount for higher quantities). Quantity pricing can be entered by first entering the regular price followed by pairs of quantities and prices such as: 10 2 15 3 18. This pricing means that the item costs $10, but a pair of the same item sells for $15, and 3 for $18. Since the price and sale-price fields can potentially take a list of numbers, they are stored as sequences and not regular numbers.

Since prices are stored as sequences, to obtain the base price of an item, we need to retrieve element 0 of the price property. To do so, use the `ELEMENT` operator:

```
WITH= variable regprice
      value ELEMENT position 0
                    sequence @price
```

The above example will store the regular price (the base price) in the local variable *regprice*. Now, let's store the base price of the sale price the same way:

```
WITH= variable regprice
      value ELEMENT position 0
                    sequence @price
  WITH= variable saleprice
        value ELEMENT position 0
                      sequence @sale-price
```

Remember, that the second `WITH=` operator is pasted ***within*** the first one. We now have the two base prices. To figure out the savings, subtract *saleprice* from *regprice*:

```
WITH= variable regprice
      value ELEMENT position 0
                    sequence @price
  WITH= variable saleprice
        value ELEMENT position 0
                      sequence @sale-price
    WITH= variable saving
          value regprice - saleprice
```

Finally, we should print the savings. We could simply use the `TEXT` operator with *saving* like this: `TEXT saving`. The problem with this approach, though, is that it would print the number unformatted. What we want to do is prefix the number with a dollar sign and to use two numbers after the decimal point. To format the number, use the `PRICE` operator. It takes two parameters, a number and a currency. Although we could add some conditions such as testing

whether there is a sale price or not, the following template snippet will print the customer's saving:

```
WITH= variable regprice
      value ELEMENT position 0
                    sequence @price
  WITH= variable saleprice
        value ELEMENT position 0
                      sequence @sale-price
    WITH= variable saving
          value regprice - saleprice
      TEXT @currency
      TEXT PRICE number saving
                 currency @currency
```

Working with images

A web site is a few graphics and a lot of text. We have dealt with text in RTML. It is now time to turn our attention to how images are manipulated or rendered.

Displaying uploaded image files

Many Yahoo! Store® users new to RTML often have some experience with HTML. They find it easier to use Yahoo! Store®'s file upload feature to upload images and then reference those images directly using HTML tags and the `TEXT` operator. This approach involves two steps:

1. Upload an image by clicking the Files button and
2. Use the TEXT operator to print the <IMG> HTML tag to display the image on a web page.

If, for example, you uploaded an image called visa.gif, you can display this image from RTML by entering the following expression:

```
TEXT "<img src=/lib/yourstoreid/visa.gif>"
```

where *yourstoreid* is the Yahoo! Store® ID of your store.

While this approach is quick and familiar to those who are experienced with HTML, it is not very flexible. If the image changes, you either have to use the same image name and upload the new image, or, if the image name changes as well, you'll have to make the change in your template (since the name is hard-coded in the template).

A much cleaner solution is to use image variables or properties. This will be the subject of the next section.

Creating images from image variables

Image variables or properties are those that let you upload and store an image. An example of such a variable is the *name-image* variable, the one that stores the banner of your store, or the *image* property of a page.

Yahoo! Store® includes image variables not just by accident; image variables have two major advantages:

- They are easy to manage. If an image needs to change, all you have to do is upload a new image without the need to worry about file names.
- They are easy to work with in RTML.

There are many image variables already defined in Yahoo! Store®, but nothing prevents you from defining your own. As with any other custom variables, when creating a new image variable, you have two choices: either define it as a new global variable, or add it to a page as a custom property. You could also add an image property to a custom type you created. Technically, it would be the same as adding a custom property to a page.

Once you have an image variable, displaying it from a template is quite easy. You need to use two operators: `IMAGE` and `RENDER`. The `IMAGE` operator is the RTML equivalent of the <IMG> tag in HTML. It has much of the same parameters as the <IMG> tag (see the RTML reference later in this book for a complete coverage of the `IMAGE` operator). The first argument of this operator is *source*, which is where we need to plug in the result of the other needed operator, `RENDER`. `RENDER` is a very versatile operator. It, too, will be covered later in the RTML reference. For our discussion here, all we are interested in is

its first argument, *image*. This is where we need to put the image variable we want to display. The following example will display the *name-image* variable:

```
IMAGE source RENDER image @name-image
```

Creating images from text

Another frequently used technique in Yahoo! Store® is to turn text into an image. This technique is mostly used for headlines (names of pages, prices, etc). For regular text, you have to rely on the fonts that are installed on the user's machine. Because you cannot know in advance what fonts are installed on your customers' machines, the only sure bet is to use some standard fonts such as Times New Roman, Arial, or Helvetica. On the other hand, when you turn text into an image, you can specify the exact font to use—limited only to those Yahoo! Store® offers.

To render text as an image, we use the same `RENDER` operator we talked about briefly before. In this case, we fill in the *text* parameter instead of passing a variable of type image to `RENDER`:

```
IMAGE source RENDER text "This sentence is rendered as
an image."
```

The above expression will create and display the image representation of the sentence "This sentence is rendered as an image." The font used will be the default black, about size 10pt, Arial or Helvetica-looking typeface. To change that, specify some of the other parameters of `RENDER`: *text-color*, *text-align*, *background-color*, *font*, *font-size*, to name a few. If you want to jump ahead into creating your own image texts without reading the sections about fonts and colors, here are two important facts about those parameters: the *font* parameter of the `RENDER` operator needs a value of type font. You cannot simply type the name of a standard font in quotes there such as "Courier". Either use a variable

of type font (for example, @display-font), or enter the name of one of the standard Yahoo! Store® graphical fonts. When entering the name of one of these fonts, start the name of the font with a colon and end it with a period as in

```
:alexa.
```

Refer to Appendix B for a list of graphical fonts available to you in Yahoo! Store®.

You can enter a color constant—such as *blue*—directly into any of the color parameters of the `RENDER` operator. Alternatively, you may use a variable or property of type color (@emphasis-color, for instance), or use the `COLOR` operator to specify a color using the RGB convention. Both the `COLOR` operator and the RGB convention will be discussed in detail in a later section.

Fusing images

Besides rendering individual images, RTML allows you to glue images together to create one larger image out of two or more smaller ones. Images can be glued together either horizontally or vertically using the `FUSE` operator.

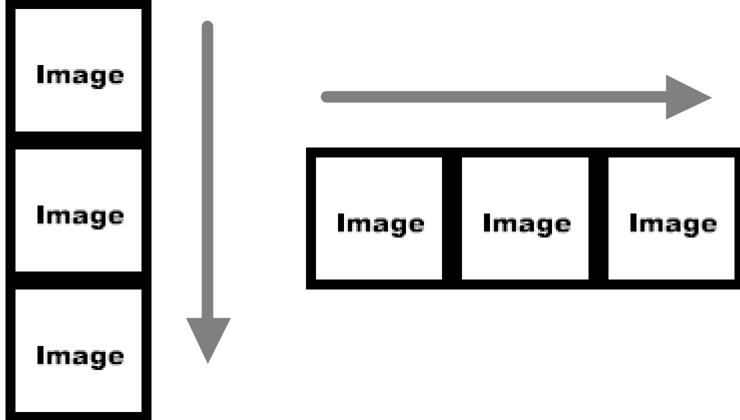


Figure 11 - Fusing Images

The FUSE operator requires at least its *axis* parameter set, and then it takes all the images pasted ***within*** its body and glues them together. The *axis* parameter must be either *:vertical* or *:horizontal*. The images pasted within the FUSE body must all be passed through the RENDER operator. The result of the FUSE operator can then be passed on to an IMAGE operator to actually display the fused image.

The following template fragment takes the *name-image* picture and the graphical representation of the name of your store, fuses the two together vertically, and displays the resultant image:

```
IMAGE source FUSE axis :vertical
               RENDER image @name-image
               RENDER text @title
```

The FUSE operator has a number of other parameters besides the required *axis*. To learn how to use these other parameters, please refer to the RTML reference later in this book.

Creating hyperlinked images

One of the most popular uses of images is to make them "clickable" by assigning hyperlinks to them. When the visitor clicks on one of these images, they are taken to another web page as if they clicked a regular hyperlink. RTML provides two ways to make an image clickable.

To assign a regular hyperlink to an image, you need to paste the RTML instructions that display the image within a `WITH-LINK` block. For those who know HTML, `WITH-LINK` is the RTML equivalent of HTML's <a> tag. It takes one parameter: a URL or Internet address. This URL can be either an explicit address as in:

```
WITH-LINK "http://stores.yahoo.com"
```

or the address of a page within your store. In this case, use the `TO` operator to obtain the address of the page. The `TO` operator takes a single argument: the ID of a page within your store.

The following example will render the *name-image* picture of your store and hyperlink it to the home page (whose ID is *:index*):

```
WITH-LINK TO :index
  IMAGE source RENDER image @name-image
```

The `RENDER` operator includes a *destination* parameter. The second way of making an image clickable is by using this *destination* parameter. The destination parameter of the `RENDER` operator works exactly the same way as the parameter of the `WITH-LINK` operator: it takes a destination URL either in the form of a literal Internet address or the result of the `TO` operator.

Note: there is one more operator, ACTION, which returns a hyperlink. ACTION is used with some "special" links, such as a link to the shopping cart. Refer to the RTML Reference later in this book on how to use the ACTION operator.

The following example is a variation to the clickable *name-image* template. This example uses the *destination* property of the RENDER operator to create a hyperlink to the home page.

```
IMAGE source RENDER image @name-image
                    destination TO :index
```

Because the RENDER operator has a *destination* parameter and FUSE expects RENDER expressions pasted within its body, it is possible to create an image by gluing other images together, each hyperlinked to a different page. To achieve this, provide a value for the *destination* parameter of each RENDER expression within FUSE. When you fuse together images (each of which is hyperlinked to a page), you are creating what is called an *imagemap*. This is, in fact, how the navigation bar is created within a Yahoo! Store®. Each button is linked to a page in your store and individual buttons are fused together to create a single navigation bar image. Look at the **nav-bar.** template to see how it is done.

Although the FUSE operator also has its own *destination* property, practical tests show that FUSE does not use this property. Either this is a bug in the implementation of FUSE or this feature was never implemented. Either way, save yourself some trouble and don't try to use the destination property of the FUSE operator; it won't work.

We discussed two methods for creating clickable images in RTML. One method was used the WITH-LINK operator. The other was used with the *destination* parameter of the RENDER operator. While they both let you achieve the same goal, there is one difference. The IMAGE operator has a parameter called *alt*. If you are familiar with HTML, you will know that *alt* is used to provide a

textual alternative for the image. This textual alternative is used in non-graphical browsers (yes, there are such things!) to show up in place of images and in graphical browsers, in place of images—if image rendering has been turned off. In graphical browsers, if image rendering is on (the default), then the *alt* text shows up as a small label when the mouse is hovered over an image. In addition, since search engines cannot interpret images, some of them use the *alt* text when "reading" a page that contains images. When you use the `WITH-LINK` method, you can use the alt tag of your `IMAGE` operator and it will work as expected. However, if you use the *destination* property of the `RENDER` operator, any text entered for the *alt* parameter of `IMAGE` will behave a bit differently. Both of the following two examples do the same (they display the *name-image* picture hyperlinked to the home page). Unlike the first one, which pops ups a small tag saying, "Index", when you hover the mouse over the image, the second does not.

```
WITH-LINK TO :index
  IMAGE source RENDER image @name-image
        alt "Index"
```

The example above uses the *alt* property as expected. The one below does not.

```
IMAGE source RENDER image @name-image
                    destination TO :index
      alt "Index"
```

Determining the size of images: HEIGHT and WIDTH

One of the many useful features of RTML is its ability to determine the size of any image. This can be a very handy feature as the problem of determining image sizes is a recurring theme in web page design and layout.

RTML provides two operators that return size information from an image: `HEIGHT` and `WIDTH`. Both take the result of either the `FUSE` or the `RENDER`

operator as their parameter and return the height and width of the image, respectively.

The following example displays the *name-image* picture and puts a horizontal line exactly the width of the *name-image* below it:

```
WITH= variable banner
      value RENDER image @name-image
  IMAGE source banner
  LINEBREAK
  HRULE width WIDTH banner
        align :left
```

Note that in this example, we stored the rendered version of @name-image in the local variable *banner*. Needing to determine the size of the image, we pass this local variable to the `WIDTH` operator. We could also have done the following:

```
HRULE width WIDTH RENDER image @name-image
```

Because `WIDTH` needs an image passed through the `RENDER` (or `FUSE`) operator, you could ***not*** have done this:

```
HRULE width WIDTH @name-image
```

Working with colors

The topic of colors deserves a few paragraphs on its own for a couple of reasons. First, RTML includes more than one operator for dealing with color. Secondly, it is easy to get confused and make mistakes with colors—especially if you have prior HTML experience.

RTML generates HTML and uses the same method to represent colors as HTML. Colors in both HTML and RTML are represented by their so-called RGB value. Each color on a computer screen can be broken up into three components: red, green, and blue. The RGB representation of a color specifies the color by providing the intensity for these three components. This intensity value can range from 0 to 255. Because there are 3 components and 256 options for each, in theory 256 * 256 * 256 = 16,777,216 or roughly 16 million colors can be represented on the computer's screen. In practice, there are only 216 so-called "web safe" colors guaranteed to look exactly the same on any computer. This limitation exists because of the differences among various computers (Macintosh, IBM Compatible, etc) and how they generate colors. The 216 "web safe" colors are those you see when you click the "Select" button in the Yahoo! Store® editor to pick a color for a color property.

In the Regular Editor, all you can do is use one of the 216 "web safe" colors. In the Advanced Editor, on the other hand, you are free to use any of the 16 Million colors. Simply enter the numerical RGB value for a color in the editor next to the color swatch as shown in Figure 12.

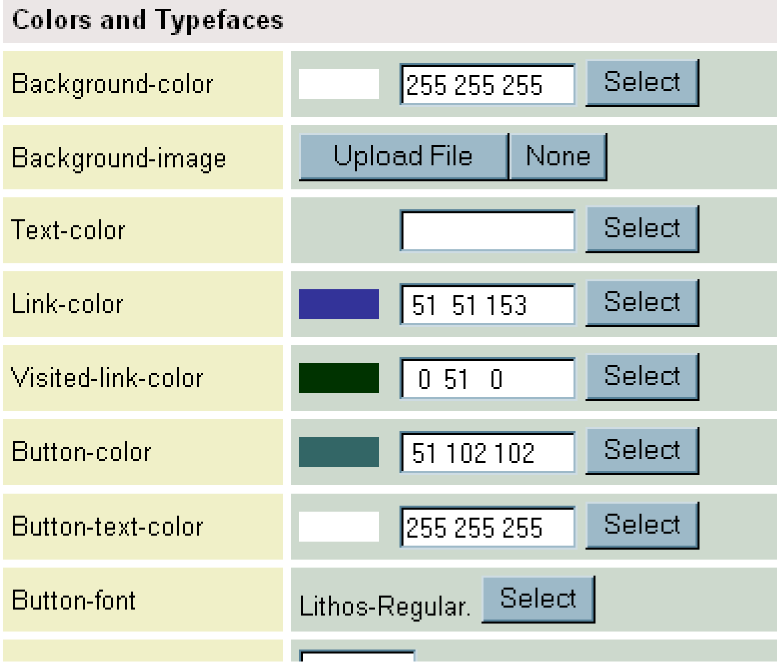


Figure 12 - colors

Properties that have the color swatch and the color selection button next to them in the editor are of type Color. Examples include link-color, background-color, and button-color. Several operators use color arguments, all of which expect values of type color. There are three ways to provide a value for such an argument: by using the name of a property of type color; by using the `COLOR` operator; or by entering an intrinsic color constant such as *blue* or *red*. Note that there is no way to enter a color value by providing the HTML representation of, say, #ff0000. It is a common mistake to try to enter something like this for a color parameter. This might come as a surprise, especially to those with prior HTML experience.

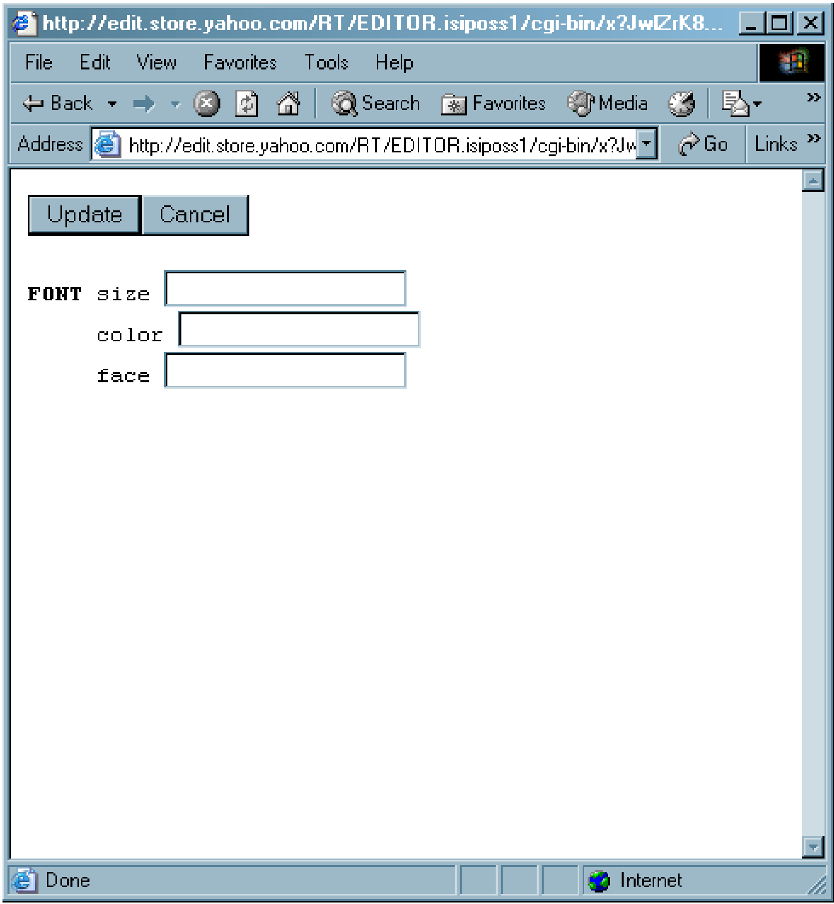


Figure 13 - Specifying colors

Color

The color operator is quite simple. It takes three parameters, one for the red, green and blue component of a color. Each component can range from 0 to 255. The result of `COLOR` is of type color, which can be used for a parameter that expects a color. The `FONT` operator, among others, has such a parameter. The following example demonstrates how color can be applied to a line of text:

```
FONT color COLOR red 255
                 green 0
                 blue 0
  TEXT "This text is red"
```

GRAYSCALE, RED, GREEN, and BLUE

RTML provides four other operators for managing colors: `GRAYSCALE`, `RED`, GREEN, and `BLUE`. All four are used to return some information about a particular color. `GRAYSCALE` returns a number representing the “grayness” of a color on a scale from 0 (black) to 255 (white.) `RED`, `GREEN`, and `BLUE` return the red, green, and blue component of a color, respectively. By using these operators, it is possible to compare colors, or to generate color effects such as a rainbow or different shades of a color, etc.

Working with fonts

In Yahoo! Store®, there are two types of fonts: regular text and graphical fonts. Regular text fonts are probably familiar to anyone with previous HTML experience. These fonts can be used to specify typefaces to be used with regular text. Graphical fonts, on the other hand, are used for headings and taglines in a Yahoo! Store®. These are selected from the font selection screen shown in Figure 14.

A fundamental difference exists between how the two types of fonts can be used. Because one is applied to text and the other to graphics, the two types typically cannot be mixed (there is a "gray area" here represented by the special Yahoo! Store® tag vw-img to be discussed later).

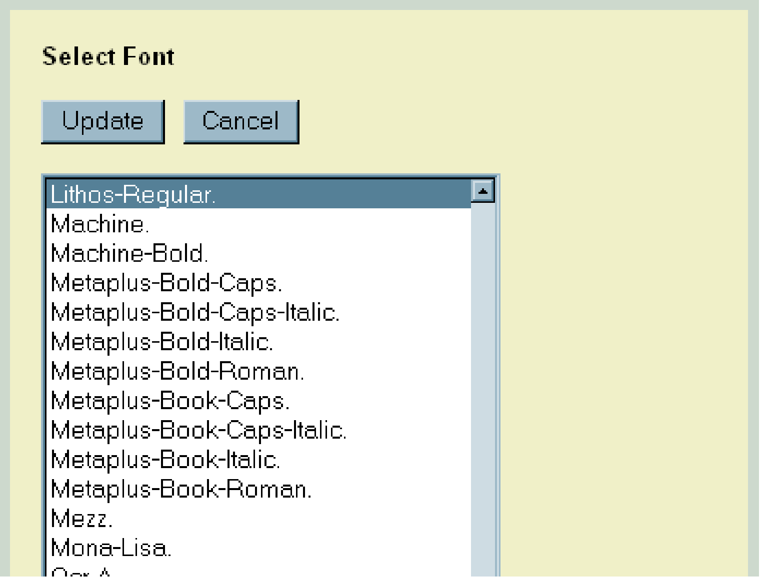


Figure 14 - Font selection screen

Font

To manipulate text fonts, RTML provides two operators: FONT and TEXT-STYLE. FONT can be used to specify the typeface, the color, and the font size to be used for text. The FONT operator has three parameters for this purpose, *size*, *color*, and *face*.

Size was borrowed from HTML where the size parameter can be an integer between 1 and 7. The rendering (actual size) of the font depends on each user's browser. A relative size may also be specified as "-1" or "+2". In these examples, "-1" would be one size smaller than the rest of the text in the same text block, whereas "+2" would mean two sizes larger.

Color must be a variable of type color (e.g. @text-color), the result of the COLOR operator, or an intrinsic color constant such as *red*. All other data types (text, for example, such as "HTML-ish" "#ff0000") are ignored.

Finally, *Face* is the name of the font to be used to render text. This parameter defines a comma-separated list of font names the visitor's browser should search for in order of preference. The setting "Arial,Helvetica", for instance, tells the browser that Arial is to be used, but if that font is not available, then Helvetica. Graphical Yahoo! Store® fonts (*Display-font*, for instance) cannot be used here. Also, note that there are only a few standard text fonts that are guaranteed to be available on most computers (some examples are Arial, Helvetica, Times, Courier). The use of any other non-standard font might result in an undesirable outcome. The text might look different on each visitor's web browser if the visitor's computer does not have the font specified. To be safe, try to stick to standard fonts.

FONT applies to text generated by expressions pasted ***within*** its body.

Text-style

TEXT-STYLE is used to apply an HTML text style to anything printed by expressions within its body. The possible HTML text styles are: *:bold*, *:italic*, *:typewriter*, *:underline*, *:strike*, *:big*, *:small*, *:subscript*, *:superscript*, and *:blinking*. These styles are applied in addition to the settings provided by the `FONT` operator.

Graphical Fonts

Graphical fonts are those used by RTML's `RENDER` operator. These fonts are different from those used by the `FONT` operator. There are two ways to specify a graphical font: by a property of type font (*Display-font*, for example), or by using the "colon-notation" where the font's name would be preceded by a : character as in *:toxica.* (note: the period is part of the font name here).

VW-IMG

The only operator that uses graphical fonts is `RENDER` and the only one that uses text fonts is `FONT`. Text fonts (such as Arial) cannot be used with `RENDER`, and similarly, graphical fonts (such as Palatino-bold) cannot be used with `FONT`. There is one exception, the special Yahoo! Store® tag *vm-img*.

Vm-img is a special tag because it is used in text fields such as the caption property of a page. However, it generates an image representing a piece of text using standard Yahoo! Store® graphical fonts. The syntax of this special tag can be best explained through an example. Entering the following line into a text box (such as the Message property of the home page or the Caption property of a regular page) generates the output shown in Figure 15.

```
<vw-img text="This text is in Kids-plain font"
font=Kids-plain. font-size=22 text-color="#000000"
align=right>
```

This text is in Kids-plain font

Figure 15 - vw-img

Text is any text within quotes. *Font* is one of the standard Yahoo! Store® graphical fonts. For this parameter, you must enter the name of the font exactly as shown in Appendix B including the period at the end. *Font-size* is the size of the font to be used. *Text-color* is the color to be used to generate the text. This parameter must be a quoted string and should specify the color using HTML's hexadecimal notation (refer to an HTML documentation for more details on this). Finally, *align* is one of "left", "right", or "center" to determine text alignment.

Part III: RTML Reference

The following section of this book covers the operators that make up the RTML language at the time this book was written. Because Yahoo! Store® Technical Support does not cover RTML, the operators and their syntax may change without notice. Such changes are probably unlikely; still, you should be aware of that possibility.

This RTML reference will list all the RTML operators at the time this book was published. As RTML is largely undocumented, the descriptions and explanations of the operators in this section are based on the online documentation available at store.yahoo.com, and from personal experience gained through trial and error while developing numerous Yahoo!® stores. Certain operators are not mentioned in Yahoo!®'s online documentation and without proper documentation, their use and/or purpose is nearly impossible to discover. Such operators are labeled as *Undocumented*.

<=>

Compares two numeric values, n1 and n2, and returns a negative number if n1 is less than n2, a positive number if n1 is greater than n2, or zero if n1 is equal to n2. This operator can be used with the SORT operator when sorting numbers.

The following example uses the <=> operator to sort the contents of the current page by price.

Example:

```
WITH= variable sorted
      value SORT var1 a
                 var2 b
                 sequence @contents
              <=> n1 WITH-OBJECT a
                       ELEMENT position 0
                               sequence @price
                  n2 WITH-OBJECT b
                       ELEMENT position 0
                               sequence @price
  FOR-EACH-OBJECT sorted
    TEXT @name
    TEXT ", "
    TEXT @price
    TEXT "<br>"
```

See Also: SORT

<

Takes two numbers, *value1* and *value2*, and returns true (:t) if *value1* is less than *value2*, or nil otherwise.

Example:

```
IF test < value1 10
          value2 20
   then TEXT "10 is less than 20"
   else TEXT "10 is not less than 20. How can that be?"
```

Takes two numbers, *value1* and *value2*, and returns true (:t) if *value1* is less than or equal to *value2*, otherwise, nil.

Example:

```
IF test <= value1 20
          value2 20
   then TEXT "20 is less than or equal 20"
   else TEXT "20 is greater than 20. How can that be?"
```

>

Takes two numbers, *value1* and *value2*, and returns true (:t) if *value1* is greater than *value2*, otherwise, nil.

Example:

```
IF test > value1 20
          value2 10
   then TEXT "20 is greater than 10"
   else TEXT "20 is not greater than 10. How can that be?"
```

Takes two numbers, *value1* and *value2*, and returns true (:t) if *value1* is greater than or equal to *value2*, otherwise, nil.

Example:

```
IF test >= value1 20
          value2 10
   then TEXT "20 is greater than or equal to 10"
   else TEXT "20 less than 10. How can that be?"
```

ACTION

Returns a URL special to Yahoo! Store®. Most often it is used in place of the `TO` operator. The possible arguments are: *:email*, *:help*, *:request*, *:search*, and *:show-order.*

:email causes `ACTION` to return a *mailto:* hyperlink with the e-mail address set in the *email* global variable. *:help* causes `ACTION` to return the URL of the store help pages (http://stores.yahoo.com/help.html). *:request* causes `ACTION` to return the URL of the registration form. *:search* causes `ACTION` to return the URL of the search page.

Finally, *:show-order* causes `ACTION` to return the URL of the shopping cart page.

Example:

```
WITH-LINK ACTION :show-order
  TEXT "Click here to shopping cart"
```

See also: TO, WITH-LINK, FORM

AND

Takes one or more arguments pasted *within* its body. If none of the arguments returns nil (false) then AND returns the value of the last argument, otherwise, it returns nil. Most commonly AND is used in decision making constructs such as WHEN and IF to test if a number of conditions are all met.

Example:

```
IF test AND
          @orderable
          @price
   then TEXT "This item is orderable"
   else TEXT "This item is not orderable"
```

Special Note for the Old Editor:
There is a minor bug in the Old RTML editor when working with the AND operator. After you paste the first argument within the body of AND, the Paste After buttons turns gray as if indicating that you cannot paste anything after the first argument. This is incorrect as you, in fact, can paste any number of arguments within the body of AND. When this happens, all you have to do is click on the AND operator (or any other operator in your template other than the newly pasted expression) and then click again the expression you had just pasted within AND. Now, the Paste After button will light up.

See also: OR, IF, WHEN

APPEND

Takes any number of sequences (except text strings) pasted within it and returns a new sequence by joining all the sequences together. The APPEND operator is only present in the new editor.

Example:

```
TEXT APPEND
       @price
       @sale-price
```

See also: SET, MAKE-LIST

AS-LIST

Creates an ordered or unordered bulleted list. Takes one parameter, which can either be *:ordered* or *:unordered*. The list items are pasted within `AS-LIST` using the `ITEM` operator. An unordered bulleted list uses bullets before each list item; an ordered list uses increasing numbers.

Example:

```
AS-LIST :ordered
  ITEM
  TEXT "First list item."
  ITEM
  TEXT "Second list item."
  ITEM
  TEXT "Third list item."
```

See also: ITEM

AUCTIONURL

Returns a URL to the auction page of an item. Yahoo! Store® items can also be sold on Yahoo! Auctions®. When you mark an item to be sold on Yahoo! Auctions®, `AUCTIONURL` returns the URL that will take you to the auction page for that item. `AUCTIONURL` takes a single parameter, the ID of an item.

Example:

```
WHEN AUCTIONURL id
   WITH-LINK AUCTIONURL id
     TEXT "Bid on this item in Yahoo! Auctions"
```

See also: WITH-LINK

BASKET

Undocumented. Purpose and use unknown.

BASKET-MODULE

This operator has not been released to the public. Currently it only works when the page that contains it is called using the http://store.yahoo.com/storeid URL and it displays the customer's shopping cart contents with a "Calculate Shipping" button – which does not work at all.

BLUE

Returns the blue component (between 0 and 255) of a color. Much like HTML, RTML uses the RGB representation of colors, in which each color is made up of three components: red, green, and blue. A color is specified by setting the three components to a number between 0 and 255, 0 being the least in-

tensity and 255 the most. In this representation, the RGB value 0 0 0 represents the color black, while 255 255 255 represents the color white.

The following example creates two 100 x 30 boxes. One will have the color specified by the *button-color* global variable. The other will be the same color but 50% lighter.

Example:

```
IMAGE source RENDER background-color @button-color
                    min-height 30
                    min-width 100
LINEBREAK
WITH= variable r
      value RED @button-color
  WITH= variable g
        value GREEN @button-color
    WITH= variable b
          value BLUE @button-color
      IMAGE source RENDER background-color COLOR red r * 2
                                                 green g * 2
                                                 blue b * 2
                          min-height 30
                          min-width 100
```

See also: RED, GREEN, COLOR

BODY

Inserts the html <BODY> and </BODY> tags into the page. Every well-formed web page should have exactly one header and one body section. All the contents of the page should be pasted within the `BODY` operator. The `BODY` operator has the following nine parameters:

Background-color: the color used for the background of the page. Usually set to *@background-color.*

Background-image: the image used for the background of the page. When specified, this must be the result of a `RENDER` or `FUSE` operator.

Text-color: the default color of any text on the page. Usually set to @*text-color*.

Link-color: the color for the regular hyperlinks on the page. Usually set to @*link-color*.

Visited-link-color: the color used for already visited hyperlinks. Usually set to @*visited-link-color*.

Topmargin: the distance, in pixels, of the page body from the top side of the browser window. Set it to 0 if you want your page to begin right at the top of the browser window.

Leftmargin: the distance, in pixels, of the page body from the left side of the browser window. Set it to 0 if you want your page to begin at the left side of the browser.

Marginheight: the size of the bottom margin of the page in pixels.

Marginwidth: the size of the right margin of the page in pixels.

Example:

```
HEAD
  TITLE "untitled"
BODY
  TEXT "This is a very simple web page."
```

See also: HEAD, META

CALL

Calls a template and returns its value. If the template has any output, that output becomes part of the current page. `CALL` takes a single argument, the name of the template to evaluate. Template names must begin with a colon. In addition, built-in templates always end with a period while custom templates never.

Optionally, templates can have arguments. Arguments are pasted ***within*** the body of the `CALL` operator, so in the RTML editor, templates should appear indented below the `CALL` operator.

If the template name parameter of the `CALL` operator is entered correctly, then a red rectangle will appear to the right of the template name. Clicking this rectangle will display the body of the called template. The absence of the red rectangle indicates that the template does not exist (or its name was mistyped).

Example:

```
HEAD
  TITLE "untitled"
BODY
  CALL :price-text.
    @price
```

CAPS

Capitalizes all letters in a string and returns the capitalized string.

Example:

```
TEXT CAPS "This is a string in all capital letters."
```

See also: LOWERCASE

CENTER

Centers all text and images pasted within its body. The `CENTER` operator is equivalent to the <CENTER> and </CENTER> tags of HTML.

Example:

```
CENTER
  TEXT "This text is centered on the screen."
```

CMP

This operator compares two symbols x1 and x2 case insensitively, and returns a positive number if x1 is greater than x2, a negative number if x1 is less tan x2, or zero if the two symbols are the same. This operator is most commonly used with SORT.

See Also: SORT, <=>, STRCMP, STRCASECMP

COLOGO

Returns a Yahoo! Shopping® image usually used in buttons. In addition, it automatically links the image either to http://shopping.yahoo.com, or to the URL specified by the *@mall-url* global variable. Takes four parameters as follows:

Height: the height of the shopping image in pixels. The maximum height allowed in this parameter is determined by the combination of the other parameters. If *thickness* is nil and *axis* is *:horizontal* then the maximum height of the logo is 39 pixels (higher values are automatically reset to 39.) If *thickness* is not nil and *axis* is *:horizontal* then the maximum height is 26 pixels. If *thick-*

ness is not nil and *axis* is *:vertical* then the maximum height is 30 pixels. Finally, if *thickness* is nil and axis is *:vertical* then the maximum height is 49 pixels.

Background-color: determines the color of the border (!) of the shopping logo. Only used when *thickness* is not nil, and in that case, the logo will have a raised look much like a button.

Thickness: when this parameter is nil (or not supplied), the shopping logo will have a flat appearance. When *thickness* is not nil, the shopping logo will have a three-dimensional border of color *background-color.*

Axis: must be either *:horizontal* or *:vertical.* When *axis* is *:horizontal*, COLOGO generates an image similar to this:

When *axis* is *:vertical*, the resultant image is like this:

Example:

```
IMAGE source COLOGO height 49
                    background-color COLOR red 255
                                           green 0
                                           blue 0
                    thickness :t
                    axis :vertical
```

COLOR

Returns a color. Like in HTML, RTML colors are represented by their RGB value. RGB stands for Red-Green-Blue and the RGB value of a color consists of the combination of the red, green, and blue component of an actual, visible color. Each component can take on a value in the range of 0 and 255 with 0 having the least intensity and 255 the most. Since the three components can take on a value from 0 to 255, there are 16 million possible color combinations. `COLOR` takes three parameters, *red*, *green*, and *blue*, corresponding to the intensity of the desired color on a scale of 0 to 255. `COLOR` can be used for any color parameter of an operator that expects some colors.

Examples are the `BODY` operator, which has many color parameters such as *background-color*, *link-color*, *visited-link-color*, etc., or the `FONT` operator, which has a *color* parameter.

Example:

```
FONT color COLOR red 255
                 green 0
                 blue 0
  TEXT "This text is red."
```

See also: RED, GREEN, BLUE, GRAYSCALE

COMMENT

Inserts a comment into the current page. Equivalent to HTML's <!-- … --> construct. A comment is just some text that becomes part of the page's HTML source but does not get displayed on the screen. The actual text of the comment must be a string enclosed in double quotes (or a variable of type text).

Example:

```
COMMENT "nosearch"
HEAD
  TITLE "untitled"
BODY
  TEXT "This page will be skipped by the Yahoo! Store Search
engine."
```

ELEMENT

ELEMENT takes two parameters, a *position* and a *sequence*. It then returns the element of sequence *sequence* at position *position*. The numbering of sequence elements starts at zero. If sequence *sequence* has no element at position *position*, `ELEMENT` returns nil. The following example will print the first letter of every special in the store:

Example:

```
WITH-OBJECT :index
  FOR-EACH-OBJECT @specials
    TEXT ELEMENT position 0
                 sequence @name
    LINEBREAK
```

See also: ELEMENTS, POSITION

ELEMENTS

The `ELEMENTS` operator returns a subsequence of a sequence. It takes three parameters: a *sequence*, a *start index*, and an *end index*. It then returns a subsequence consisting of the elements of *sequence* from *start index* to *end index*. The numbering of sequence elements is zero-based. The second and third parameters of `ELEMENTS` are optional. If *first* is omitted, then it is assumed to be 0 (mean-

ing the start of the sequence). If *last* is omitted, it is assumed that it is the last element of the sequence. Consequently, if both *first* and *last* are omitted, then the ELEMENTS operator simply returns the *sequence* itself.

Example:

```
TEXT ELEMENTS sequence "abcde"
              first 1
              last 3
```

See also: ELEMENT

EQUALS

The EQUALS operator takes two parameters, *value1* and *value2*, and returns true if the two values are the same or nil otherwise. *Value1* and *value2* can be any valid data type (string, number, sequence, …). They should both be of the same type, otherwise EQUALS will always return nil. For example, a number and a text string will never be equal.

While comparing values of type image is allowed, the comparison will only return true if both images are empty (nil). Therefore, comparing values of type image is meaningless.

Example:

```
IF test EQUALS value1 @name
               value2 @code
   then TEXT "The nam of this item is the same as its ID."
   else TEXT "The name and the ID of this item are different."
```

See also: NOT

EVEN

Returns true if its argument is an even integer. Works with integers only; therefore, even though 12.0 is an even number, `EVEN 12.0` returns false (nil) because 12.0 is not an integer (rather, a real number).

Example:

```
IF test EVEN LENGTH WHOLE-CONTENTS
   then TEXT "The number of contents pages in this store is even."
   else TEXT "The number of contents pages in this store is odd."
```

FIND-ONE

`FIND-ONE` takes two arguments: a *control variable* and a *sequence*. It then assigns consecutive elements of *sequence* to *control variable* and evaluates the expression pasted ***within*** `FIND-ONE`. If the expression is true (not nil) then the current value of *control variable* is returned. If the expression is false for every element of *sequence*, then `FIND-ONE` returns nil. In plain English, `FIND-ONE` says: "find one element in a sequence for which a given expression is true." The following example will find the first item in the store that has no name and will print its ID.

Example:

```
TEXT FIND-ONE variable item
              sequence WHOLE-CONTENTS
       WITH-OBJECT item
         EQUALS value1 @name
                value2 "No Name"
```

FONT

Allows for specifying the font face, size, and color for any text pasted within its body. Equivalent to the <FONT> tag in HTML. Takes three parameters:

Size: the size of the font. This is not an absolute size. This parameter was borrowed from HTML where the size parameter can be an integer between 1 and 7. The rendering (actual size) of the font depends on each user's browser. A relative size may also be specified as "-1" or "+2". In these examples, "-1" would be one size smaller than the rest of the text in the same text block, whereas "+2" would mean two sizes larger.

Color: the color of the text. Must be a variable of type color (@text-color, for example), the result of the `COLOR` operator, or an intrinsic color constant such as *red* or *blue*. All other data types (text, such as "HTML-ish" "#ff0000") are ignored.

Face: the font name to be used. This parameter defines a comma-separated list of font names the visitor's browser should search for in order of preference. The setting "Arial,Helvetica", for example, tells the browser that Arial is to be used, but if that font is not available, then Helvetica. Graphical Yahoo! Store® fonts (*Display-font*, for instance) cannot be used here. Note there are only a few standard text fonts that are guaranteed to be available on most computers (examples are Arial, Helvetica, Times, Courier). The use of any other non-standard font might not result in the desired outcome. The text might look different on each individual visitor's web browser if the visitor's computer does not have the font specified. To be safe, try to stick to standard fonts.

Example:

```
FONT size 2
     color COLOR red 0
                 green 0
                 blue 0
     face "Arial"
  TEXT "This text is in size 2 black Arial font."
```

See also: TEXT-STYLE

FONT-WIDTH

Returns a number indicating how wide a font is relative to Helvetica Bold. The FONT-WIDTH operator works with Yahoo! Store®'s graphical fonts only. Those fonts can be selected from a list. An example is *Display-font*. According to `FONT-WIDTH`, *Lithos-Bold*, for example, is 1.3068392 times wider than Helvetica Bold.

FOR

The FOR operator repeats the expressions pasted within its body a predefined number of times. It takes four parameters: a variable called the *control variable*, an *initial value*, a *test condition*, and an *update operation*. Of these, the *initial value*, the *test condition*, and the *update operation* can be any valid RTML expression.

In theory, the FOR operator in RTML works as follows: it takes the *control variable* and assigns it *initial value*. It then evaluates *test condition*. If *test condition* is false (nil) then the FOR operator returns the current value of the *control variable*. If *test condition* is true, then the expression pasted ***within*** the body of FOR is evaluated (executed). Finally, the update operation is evaluated and the whole cycle starts again. It repeats until *test condition* becomes false (nil).

As any RTML operator, FOR returns a value as well. The value returned by FOR is the last value of the control variable, that is, the ***first*** value for which the test *condition fails*.

There are a couple of important details you should know about the FOR operator:

- You might be tempted to type something like "i <= 10" for *test condition* especially since when you edit a FOR operator, there is a text box next to *test* (see Figure 9.) You can't do that. Instead, you should paste a more complex expression in place of the test condition such as the <= operator.
- While some programming languages evaluate the test condition of their FOR command ***after*** each cycle, others—and RTML is among these—evaluate the test condition ***before*** each cycle. This is an important fact, because those that test after each cycle execute at least one cycle guaranteed (since they don't check the test condition until the first cycle,)

whereas those that test before each cycle might not execute a single loop at all if the test condition is false to begin with.

Example:

```
FOR variable I
    initial 1
    test <= value1 I
            value2 10
    update i + 1
  TEXT I
  LINEBREAK
```

FOR-EACH

FOR-EACH takes a variable and a sequence. It then assigns each element of the sequence to the variable, one after the other, and for each element, it evaluates the expression pasted ***within*** its body. FOR-EACH returns a sequence consisting of the values returned by the last expression during each iteration.

Example:

```
FOR-EACH variable x
         sequence "abcdefghijklmnopqrstuvwxyz"
  TEXT x
  LINEBREAK
```

FOR-EACH-BUT

FOR-EACH-BUT is very similar to FOR-EACH. It takes a *variable*, a *sequence*, and a *last expression*. It evaluates the expression or expressions pasted *within* for every element of the sequence, but—and here is where it differs from FOR-EACH – it also evaluates *last expression* for each element ***except*** for the last one.

The following example prints the letters a, b, c, and d separated by commas and there will be no comma after d:

Example:

```
FOR-EACH-BUT variable x
             sequence "abcd"
             last TEXT ","
  TEXT x
```

FOR-EACH-OBJECT

`FOR-EACH-OBJECT` takes a single argument, a list of objects or IDs, such as the *Contents* property of a page. It then walks through and changes context to each element of the list, so every expression pasted ***within*** the `FOR-EACH-OBJECT` block will be evaluated in the context of that element.

Equivalent to the following:

```
  FOR-EACH variable item
           sequence nil
    WITH-OBJECT item
      ...
```

Example:

```
WITH-OBJECT :index
  FOR-EACH-OBJECT @contents
    TEXT @name
    LINEBREAK
```

FORM

Creates a web form. Equivalent to the <FORM> tag of HTML. `FORM` takes a single argument, which becomes the *action* of the form. The action or target URL argument can be a text string specifying some location (typically a script, a

CGI program or something similar) or a URL returned by the `TO` or `ACTION` operators. When the form is submitted, its values are *POST*ed to the target URL. The following example will display a single submit button, which when clicked, transfers the visitor to the shopping cart page.

Example:

```
FORM ACTION :show-order
  INPUT type :submit
        value @show-order-text
```

See also: ACTION, TO, INPUT, TEXTAERA, SELECT

FUSE

`FUSE` takes the images pasted within its body and "glues" them together either vertically or horizontally creating a single image. Of the `FUSE` operator's parameters, at least *axis* must be set (but only if more than one images are pasted within `FUSE`).

The images pasted within `FUSE`'s body must all be passed through the `RENDER` or another `FUSE` operator. Because `RENDER` can take a *destination* parameter (which is essentially a hyperlink), `FUSE` can be used to generate what are called *image maps*. Image maps are images that have certain "hot spots" that can be clicked. Much like clicking a regular hyperlink, clicking such a hot spot will transfer the visitor to a different page. The standard navigation bar—either horizontal or vertical—in a Yahoo! Store® is an image map created by the `FUSE` operator to join the individual images of the navigation buttons.

The result of the `FUSE` operator can then be passed on to an `IMAGE` operator to actually display the fused image. Fuse has 10 parameters:

Axis: either *:vertical* or *:horizontal*. This parameter determines the direction in which the individual pieces are joined. The *axis* parameter can be omitted only if there are fewer than two images pasted within the body of `FUSE`. In this case, `FUSE` simply degenerates into a `RENDER` operation.

Background-color: the color used for the background of the resultant image. Must be a variable of type color, the result of the `COLOR` operator, or the special constant *transparent*.

Top-margin, bottom-margin, left-margin, right-margin: determine how far inside the resultant image the individual pieces are.

Spacing: the distance, in pixels, between the individual images.

Destination: currently ignored. If the entire `FUSE`d image needs to be hyperlinked, paste it within a `WITH-LINK` block.

Align: one of *:left*, *:right*, or *:center*. Determines how the individual pieces are aligned. The default is centered. If omitted and *axis* is *:vertical*, all images within FUSE will be stretched to the width of the widest image.

Thickness: if not nil, creates a raised border around the entire image but only if *background-color* is set to a color other than *transparent*.

The following example takes the *name-image* picture and the graphical representation of the name of your store, fuses the two together vertically, and displays the resultant image:

Example:

```
IMAGE source FUSE axis :vertical
               RENDER image @name-image
               RENDER text @title
```

See also: RENDER, IMAGE

GET-ALL-PATHS-TO

Returns all the paths between two objects. More specifically, it can be used to return all the paths leading from the home page to the current page. It can be useful, for example, to display multiple "breadcrumbs" paths when an item has been copied into several sections. It takes four parameters:

Dst: the ID of the destination page

Src: the ID of the source page

Nodst: if set to :t, then the results will not include the ID of the destination page itself

Nosrc: if set to :t then the results will not include the ID of the source page

The result is a sequence whose elements are sequences of Ids leading from the source page to the destination page.

Example:

Consider the following store structure and template:

```
⊟ index main. home.
   ⊟ triassic-period item. page2
        test2 item. test2  [Show Parents]
   ⊟ jurassic-period item. page.
        test2(shown under triassic-period)
```

```
Test2 ()

HEAD
  TITLE "Untitled"
BODY
  TEXT GET-ALL-PATHS-TO dst id
                        src :index
                        nodst nil
                        nosrc nil
```

When this template is applied to object test2 above, it outputs the following: ((:INDEX :TRIASSIC-PERIOD :TEST2) (:INDEX :JURASSIC-PERIOD :TEST2))

See Also: GET-PATH-TO

GET-PATH-TO

This operator returns a sequence of IDs leading from one page to another. The operator takes the following four parameters:

Dst: the ID of the destination page

Src: the ID of the source page

Nodst: if set to :t, then the results will not include the ID of the destination page itself

Nosrc: if set to :t then the results will not include the ID of the source page

Example:

The most popular application of this operator is to display the "bread-crumbs" path for a page:

```
WITH-LINK TO :index
  TEXT "Home"
TEXT " > "
FOR-EACH-OBJECT GET-PATH-TO dst id
                            src :index
                            nodst :t
                            nosrc :t
  WITH-LINK TO id
    TEXT @name
  TEXT " > "
TEXT @name
```

See Also: GET-ALL-PATHS-TO

GRAB

GRAB captures any text to be written to the current page and returns it as a string. This operator can be used to convert variables of various types to strings or to concatenate text strings. The following example assigns to the local variable *line* a text string made of the sentence "This store is called " and the name of the current store. It then prints the entire line on the current page.

Example:

```
WITH= variable line
      value GRAB
              TEXT "This store is called "
              TEXT @title
  TEXT line
```

See also: TEXT

GRAYSCALE

Returns a number representing the "grayness" of a color on a scale of 0 (black) to 255 (white).

See also: RED, GREEN, BLUE

GREEN

Returns the green component (between 0 and 255) of a color. See the `BLUE` operator for more details.

See also: RED, BLUE, GRAYSCALE, COLOR

HEAD

Corresponds to the HTML <head> section. Each web page should contain exactly one HEAD and one BODY section. The HEAD or header area of the page contains some general information about the page such as its title, description, or keywords, and other data that is not considered part of the document content (not displayed in the web browser). Any text pasted within the `HEAD` operator will be inserted between the <HEAD> and </HEAD> tags of the current page.

Example:

```
HEAD
  TITLE "untitled"
```

See also: BODY, META

HEIGHT

HEIGHT returns the height of an image in pixels. The image passed to the HEIGHT operator must be an image that is already rendered, i.e., one that was returned by the RENDER or FUSE operators. Passing a variable of type image (such as the *name-image*), to HEIGHT returns nil.

See also: WIDTH, RENDER, FUSE

HEX-ENCODE

Takes a string and returns another containing the hexadecimal (base 16) representation of each character of the source string. HEX-ENCODE "A", for example, returns 41 (the hexadecimal value 41 is 65 in decimal which is the ASCII code of the letter A). Most commonly used for generating the names of INPUT tags in forms as those names may contain only letters and numbers.

Example:

See the built-in template *Inscription*.

HRULE

Inserts a horizontal line into the current page. Equivalent to the <HR> tag in HTML. Takes three parameters, any of which may be omitted:

Width: the horizontal size of the line in pixels. If omitted, the line will extend to the entire width of the page (or table cell depending on where the line is used).

Align: Specifies the horizontal alignment of the line with respect to the surrounding text. Possible values are: *:left*, *:right*, or *:center*. If not specified, *:center* is assumed.

Noshade: when this parameter is other than nil, the line will be solid rather than the standard two-color groove.

Example:

```
HRULE width 200
      align :center
      noshade :t
```

IF

The `IF` operator takes three parameters: a *test* expression, a *then* expression and an *else* expression. If the *test* expression is other than nil (meaning it is true or not false), then the *then* expression is evaluated. If the *test* expression is nil (or false) then the *else* expression is evaluated. The return value of IF is the result of either the *then* or the *else* expression depending on which one was evaluated.

Although each of the *test*, *then*, and *else* expressions can only be single expressions, they can be complex expressions either by calling another template using the `CALL` operator or by grouping a number of expressions with the `MULTI` operator.

The following example will print the title of the store if the current page has no name, or the title of the store and the name of the current page separated by a horizontal line if the current page does have a name.

Example:

```
IF test NOT EQUALS value1 @name
                   value2 "No Name"
   then MULTI
          TEXT @title
          HRULE
          TEXT @name
   else TEXT @title
```

See also: WHEN

IMAGE

Inserts an image into the current page. Takes the following ten parameters:

Source: An image. Must be the result of a RENDER or FUSE operator (one of the most common mistakes is to simply use a variable or property of type image such as @*name-image*).

Lowsource: a low-resolution image (also the result of a RENDER or FUSE operator). By using a low-resolution image, you can greatly reduce the loading time of pages that include large photos. A low-resolution image is typically the same as the actual image but in grayscale and lower resolution or higher compression to reduce the size of the low-resolution image file. When a low-resolution image file is used, the web browser will render the low-resolution image first. Then, it will make a second pass and will "gradually" wipe the actual, high-resolution image over the low-resolution copy. This "wipe-over" effect, however, will only be noticeable while viewing the page on a slow Internet connection. On faster connections, the low-resolution

net connection. On faster connections, the low-resolution image might not even show up.

Width, height: the size of the image in pixels. Typically, web pages that include width and height parameters for the embedded images load faster because they allow the web browser to "reserve" space for the image on the page. These parameters – if they don't match the actual size of the image – don't scale the image, they simply stretch it to the values of these parameters or change the HTML tags that tell the Web browser how to display the picture. To scale the image and enlarge it or shrink it, use the appropriate parameters of the `RENDER` operator (see later).

Align: can be one of *:top*, *:middle*, *:bottom*, *:left*, or *:right*. The first three (*:top*, *:middle*, and *:bottom*) determines how the image should be aligned vertically relative to the surrounding text. The last two settings, *:left* and *:right*, will cause the image to float either on the left or the right-side of the page (or table cell enclosing the image) and surrounding text will be wrapped around the image.

Border: sets the size of the border around the image in pixels. If the image is hyperlinked, setting this parameter to 0 will cause the image not to have a border around it. When the image is not hyperlinked, omitting the `border` parameter will also cause the image not to have a border.

Hspace, vspace: set the vertical (*vspace*) and horizontal (*hspace*) spacing around the image. Useful to specify some "breathing room" around the images so that the surrounding text doesn't go flush against the image.

Alt: a textual representation of the image. This property has three basic uses:

1. In non-graphical browsers (yes, these really exist!) or in graphical browsers if image support has been turned off, the text entered for the *alt* parameter is displayed in place of the image.
2. When the mouse hovers over an image that has the *alt* parameter specified, the *alt* text pops up as a so-called "tool tip."
3. Search engines cannot understand images, but they can read and index the *alt* text.

Antialias-color: This color is used for *anti-aliasing* the image. *Anti-aliasing* refers to a technique to remove the jagged edges from an image by blurring the edges to trick the eye into seeing a more continuous border. For best results, use an anti-alias color that matches the background behind the picture. Successful anti-aliasing is most noticeable when creating an image using the `RENDER` operator to create text.

Example:

```
IMAGE source RENDER image @image
```

See also: RENDER, FUSE

IMAGE-REF

Returns the URL of properties or global variables of type image. When an image is uploaded into any image type property or global variable (such as the *image* of an item), the image is stored internally by the Yahoo! Store® system, however, the exact location of the image is not revealed by the editor. To find out the location (URL) of an image, pass the image to the `IMAGE-REF` operator.

The following example will show a thumbnail of the current item. When the thumbnail is clicked, the full size image will be shown. This is almost exactly how thumbnails work in a published store.

Example:

```
WITH-LINK IMAGE-REF @image
  IMAGE source RENDER image @image
                      max-height @thumb-height
                      max-width @thumb-width
```

See also: IMAGE

INPUT

The `INPUT` operator can be used to create a form field. While Internet Explorer works fine with input fields not contained within forms, for compatibility with Netscape, you should always paste the `INPUT` operator within a `FORM` block.

The `INPUT` operator has five parameters and by manipulating these parameters, it is possible to create the following types of input fields: text box, password field, submit button, reset button, button, hidden, checkbox, and radio button. Notice, that two other input types, *image* and *textarea*, are missing from the list above. While a *textarea* input field (one that allows the entering of multiple lines of text) can be created using the `TEXTAREA` operator, RTML currently does not provide a direct way to create an *image* type form element (*image* type form elements can be used to create graphical submit buttons).

The five parameters of the `INPUT` operator are: *name*, *type*, *value*, *maxlength*, and *size*. Any and all of these parameters can be left blank in which case `INPUT` will create a simple text field. Some of the parameters are only used for certain *types* as detailed below.

Name: the name of the form element. This can either be a string enclosed in double quotes or a colon followed by the name of the element. The two notations are equivalent.

Type: can be one of *:text*, *:password, :submit, :reset, :button, :hidden, :checkbox*, or *:radio* to create a text box, a password field (one that displays asterisks for every character typed), a submit button, a reset button (which clears the entire form), a regular button, a hidden field, a check box, or a radio button, respectively.

Value: for a text box, a hidden field, and a password field, *value* specifies the default value for the field. For a check box and a radio button, it specifies the value returned when the check box is checked or the radio button is selected. For a submit button, it sets the label shown on the button and the value returned when the submit button is clicked and the form is submitted. Finally, for a reset or regular button, *value* sets the label for the button.

Maxlength, Size: these parameters are used only for text boxes and password boxes. *Maxlength* determines the maximum number of characters allowed in the text box, while *Size* controls the number of characters shown in the text box (the width of the text box).

The following example generates a functioning search form. On a published site, this form can be used to search for a word or phrase among the pages of the store. The `SEARCH-FORM` operator does the exact same thing, however the example below is perfect to demonstrate how `INPUT` is used.

Example:

```
FORM ACTION :search
  INPUT name :catalog
        type :hidden
        value account
  INPUT name :query
        type :text
        size 12
  INPUT type :submit
        value "Find"
```

Note the special variable *account*. This variable contains the Yahoo! Store® ID of the current store.

See also: FORM

INVENTORY-INFO

Undocumented.

ITEM

Creates a line item in a bulleted list. See `AS-LIST`.

ITEM-INVENTORY

This operator generates a submit button that can be used to order an item. In the old editor, below the order button, it will also display the number of the particular item in stock. This feature only works if Database Inventory is enabled

for your store and only if your store is still based on the old editor. In the new editor, this operator simply creates an Order button. If either Database Inventory is not enabled, or your store is using the Real-time Inventory method, `ITEM-INVENTORY` simply creates a submit button.

`ITEM-INVENTORY` takes two parameters: *code* and *order-text*. *Code* must be the code or SKU of an existing item in your store. *Order-text* is a string providing the label for the order button. For more information on the Database Inventory feature, visit http://store.yahoo.com/vw/dbinv.html.

The example below is a variation to the built-in **Order.** template. The **Order.** template is responsible for producing the "order block" for the item pages (the "order block" contains the availability info, the options if there are any, and the order button). This variation replaces the order button with one that, when Database Inventory is enabled, will also show the number of items in stock. The code only lists the relevant part of the template.

Example:

```
WHEN AND
       dest
       OR
         @price
         @sale-price
       @orderable
  INPUT name :vwitem
        type :hidden
        value id
  INPUT name :vwcatalog
        type :hidden
        value account
  ITEM-INVENTORY code @code
                 order-text @order-text
```

Note: this operator does nothing in the new editor.

LENGTH

Returns the length of a sequence (either a list or a string). The length of a list is the number of members in the list. The length of a string is the number of characters in the string.

Example:

```
TEXT "The number of specials in this store is "
WITH-OBJECT :index
  TEXT LENGTH @specials
```

See also: POSITION, ELEMENTS

LINEBREAK

Inserts a line break into the current page. LINEBREAK has two optional arguments: *number* and *clear*. When *number* is specified, it will cause that many number of line breaks inserted into the current document. The *clear* parameter can take one of the following values: *:none* (this is the default,) *:left*, *:right*, or *:all*. When specified, this parameter controls the flow of text around floating objects. Floating objects are typically tables or images whose *align* property is set.

LINES

Each paragraph may consist of several lines. The lines of a paragraph are separated by new line characters (but not with blank lines). To obtain a sequence of the lines of a paragraph, use the LINES operator. The LINES operator takes a single argument, a text string, and returns a sequence of text strings consisting

of the lines of the original. The following example will take the *Message* property of the home page and print each line of text on a separate line.

Example:

```
WITH-OBJECT :index
  FOR-EACH variable line
           sequence LINES @message
    TEXT line
    LINEBREAK
```

LIVE

Undocumented. Removed from the new editor.

LOCAL-LOGO

In the old editor, LOCAL-LOGO returns a rendered Y! Store logo. Can be passed directly to the `IMAGE` operator for display. Takes one parameter, which is undocumented. This operator is no longer present in the new editor.

Example:

```
IMAGE source LOCAL-LOGO
```

LOG

Undocumented.

LOOKUP

Undocumented. Not present in the new editor.

LOWERCASE

Converts a string into all lowercase characters.

Example:

```
TEXT LOWERCASE "This string is now in lowercase."
```

See also: CAPS

MAKE-LIST

Make list takes any number of values pasted within it and returns a sequence consisting of all those values. This operator is only available in the new editor.

Example:

```
TEXT MAKE-LIST
       "a"
       "b"
       "c"
```

See also: SET, APPEND

MAXIMUM

`MAXIMUM` takes any number of numbers pasted ***within*** and returns the highest one.

Example:

```
TEXT "The highest number of 1, 2, and 5 is "
TEXT MAXIMUM
       1
       2
       5
```

See also: MINIMUM

META

Inserts an HTML *meta tag* into the current page. Meta tags are HTML tags that contain information *about* the current web page. Although these tags are not rendered by the web browser, certain meta tags are important because they are used, for example, by the various search engines. Some of the most commonly known meta tags are *keyword* and *description*. Meta tags should only be entered within the header of a web page. In RTML, that is within the `HEAD` operator.

The `META` operator has three parameters, *name*, *http-equiv*, and *content*. They correspond to the attributes of the HTML META tag.

Example:

```
HEAD
  META name "keywords"
       content "demo,page,rtml"
  TITLE "demo"
```

See also: HEAD

MINIMUM

`MINIMUM` takes any number of numbers pasted ***within*** and returns the lowest one.

Example:

```
TEXT "The lowest number of 1, 2, and 5 is "
TEXT MINIMUM
       1
       2
       5
```

See also: MAXIMUM

MODULE

The MODULE operator creates a table with a specific title, title color, and body color. Any output generated by expressions pasted within MODULE will be rendered inside the table. Module has three parameters, *title*, *title-color*, and *body-color*. *Title* is a string and is used to specify the heading for the table. The heading will have a background of color *title-color*. The color of the table's body is determined by *body-color*. The table generated by the MODULE operator always occupies the maximum width it can.

The following example will create a module table titled "Specials" and it will list the names of the specials of the current store within the table.

Example:

```
MODULE title "Specials"
       title-color COLOR red 102
                         green 204
                         blue 204
       body-color COLOR red 294
                        green 255
                        blue 255
  WITH-OBJECT :index
    FOR-EACH-OBJECT @specials
      TEXT @name
      LINEBREAK clear nil
                number nil
```

MULTI

The `MULTI` operator groups together several expressions pasted within it and returns the value of the last one.

Certain operators have parameters that only allow for entering a single expression. The `IF` operator, for example, can contain only one RTML expression in its *test*, *then*, and *else* parameters. By using the `MULTI` operator, it is possible to go around this limitation and enter several expressions for these and similar parameters.

Note that the RTML editor has a minor bug in its treatment of the MULTI operator. After pasting the first expression within MULTI, the Paste After button becomes disabled even if the expression stack is not empty. To re-enable the Paste After button, simply click on any other expression within the template, and then click back on the operator most recently pasted within MULTI.

Example:

```
IF test EQUALS value1 id
               value2 :index
   then MULTI
          TEXT "The title of this page is "
          TEXT @page-title
   else MULTI
          TEXT "The title of this page is "
          TEXT @name
```

NOBREAK

NOBREAK causes any output generated by expressions pasted within it to be rendered all on one without line breaks. This operator is equivalent to the NOBR tag in HTML.

Example:

```
NOBREAK
  TEXT "This sentence will never wrap around."
```

NONEMPTY

NONEMPTY takes a *string* as its parameter and returns true if the string contains at least one non-whitespace character.

The following example is from the built-in **page-body**. template. It checks if Final-text contains anything, and if so, prints it on the current page.

Example:

```
WHEN NONEMPTY @final-text
  CALL :paras-in-box
    @final-text
    wid
```

NOT

`NOT` takes a single argument and it returns true if the argument is false, or false if the argument is true. The following example will print “This is not the home page” if the current page is other than the home page.

Example:

```
WHEN NOT EQUALS value1 id
                value2 :index
  TEXT "This is not the home page."
```

NUMBER

`NUMBER` is used to format numeric values. It takes two parameters, *number* and *digits* and prints *number* with *digits* numbers after the decimal point. `NUMBER` guarantees that there will be exactly *digit* numbers after the decimal point.

The `NUMBER` operator does not simply return the formatted number; it actually prints it on the current page. Therefore, if you need to store the formatted number instead, use the `GRAB` operator to capture the output of `NUMBER`.

Example:

```
NUMBER value ELEMENT position 0
                     sequence @price
       digits 2
```

See also: PRICE

OBJID-FROM-STRING

Takes one string parameter and converts it into an object Id. For example, OBJID-FROM-STRING “foo” will return *:foo* if there is an object (page) in the store whose id is *foo*. If there is no object with the specified name, OBJID-FROM-STRING returns nil.

Example:

The following example converts a space-delimited list of IDs stored in the “Label” property into a sequence of object Ids:

```
TEXT FOR-EACH var elem
              sequence TOKENS @label
       OBJID-FROM-STRING elem
```

REVERSE

Reverses a sequence. This operator works on true sequences and cannot be used to reverse a text string.

OR

OR takes one or more arguments (pasted ***within*** its body), each of which must be a valid RTML expression, and it returns the value of the first one that is other than nil. Once it finds an expression whose value is other than nil, the rest of the expressions are not evaluated.

The following example will print "This item is orderable or taxable" if either the *orderable* or the *taxable* (or both) property of the current page is set to "Yes."

```
IF test OR
         @taxable
         @orderable
   then TEXT "This item is orderable or taxable."
   else TEXT "This item is neither orderable nor taxable."
```

The fact that the first non-nil value of the OR operator is returned and the rest will not be evaluated is important. Because of this, we can write expressions such as this:

```
WITH= variable price
      value OR
              @sale-price
              @price
```

The above example sets the local variable *price* to the value you have under Sale-price if you, in fact, entered anything for Sale-price, or the value of Price (regular price) otherwise. Notice, that the above example is NOT equivalent to this:

```
WITH= variable price
      value OR
              @price
              @sale-price
```

Because `OR` returns the value of its first non-nil expression, this example would almost always set the local variable price to the value of the regular price (unless you forgot to enter the regular price but not the sale price).

See also: AND, NOT

ORDER

`ORDER` takes the ID of an item and returns the URL of the page that will order that item. Because the Yahoo! Store® shopping cart requires a number of hidden parameters passed to it in order to function correctly (refer to the **Order.** template to see the details), the `ORDER` operator should only be used as the *action* parameter of a `FORM` and with all the required parameters included in the form. The following example creates a form with an order button that will put the current item in the shopping basket.

Example:

```
FORM ORDER id
  WHEN AND
         ORDER id
         OR
           @price
           @sale-price
         @orderable
    INPUT name :vwitem
          type :hidden
          value id
    INPUT name :vwcatalog
          type :hidden
          value account
    ITEM-INVENTORY code @code
                   order-text @order-text
```

ORDER-FORM

Inserts the shopping cart into the current page. The following example shows how ORDER-FORM is used in the built-in **Order-body.** template:

Example:

```
Order-body. (wid)
TABLE border 0
      cellspacing 0
      cellpadding 0
      width wid
  TABLE-ROW valign :top
    TABLE-CELL
      ORDER-FORM
```

PARAGRAPH

`PARAGRAPH` begins a new paragraph in the current page. It takes a single, optional alignment argument, which can have any of the following values: *:left,*

:right, :center. When the alignment argument is set, subsequent text printed to the current page will be aligned accordingly. To cancel the effect of the alignment set by the PARAGRAPH operator, paste a new PARAGRAPH operator into the template. The PARAGRAPH operator is equivalent to the <P> tag in HTML. Because PARAGRAPH always inserts a blank line before itself, use LINEBREAK instead if only a new line is needed between two lines.

Example:

```
PARAGRAPH :center
TEXT "This line is centered."
PARAGRAPH
TEXT "This line is left-aligned."
```

PARAGRAPHS

RTML provides the PARAGRAPHS operator to separate a text string into paragraphs. The PARAGRAPHS operator takes a single parameter, a source text string, and it returns a sequence of text strings consisting of the paragraphs of the source text string. The following example takes the *Message* property of the home page and prints each paragraph of it on a separate line.

Example:

```
WITH-OBJECT :index
  FOR-EACH variable para
           sequence PARAGRAPHS @message
    TEXT para
    LINEBREAK
```

POSITION

`POSITION` takes two arguments: an element and a sequence. It returns the position at which the sequence contains the specified element or nil, if the element was not found in the sequence. The numbering of the elements within a sequence starts at position 0.

This operator is most commonly used to check if an element exists within a sequence. The following example demonstrates this use.

Example:

```
WHEN POSITION element :contents
              sequence @nav-buttons
  TEXT "Contents are part of Nav-buttons."
```

See also: ELEMENT, ELEMENTS

PRICE

The `PRICE` operator is used to format currency values for printing. `PRICE` takes two parameters, a number and a currency symbol. The number parameter is the price to be formatted, while the currency symbol is a string to be used for currency. This parameter typically takes on the value of the global @currency variable.

The result of the operator is the number preceded by the currency symbol specified and formatted to include two digits after the decimal.

Example:

```
TEXT PRICE number ELEMENT position 0
                          sequence IF test @sale-price
                                      then @sale-price
                                      else @price
           currency @currency
```

See also: NUMBER

RED

Returns the red component (between 0 and 255) of a color. See the `BLUE` operator for more details.

See also: GREEN, BLUE, GRAYSCALE, COLOR

RENDER

The render operator can create an image from an image property; render some text as an image, or both. It is an extremely versatile operator. With its twenty parameters, it provides quite a few ways to manipulate images.

The `RENDER` operator does not actually display an image; to show the image, the result of `RENDER` must be passed on to the `IMAGE` operator as its *source* value. The parameters of `RENDER` are as follows:

Image: a property or variable of type image. The *name-image* global variable or the *image* property of an object is such type. When the *image* parameter is specified, `RENDER` generates an image from the image variable.

Text: when specified, renders *text* as an image. If both *image* and *text* is specified, then RENDER superimposes *text* over *image*. This technique can be used to generate uniform buttons, for example.

Text-color: the color to be used to render *text.*

Text-align: one of *:center*, *:left*, or *:right*. Sets the alignment of *text.*

Background-color: sets the color for the image. This parameter has no visible effect if *image* is specified. In other words, *background-color* should only be used when rendering text as an image. Must be a variable or property of type color, the result of the COLOR operator, or the special constant *transparent*.

Font: font to be used when rendering text as an image. This parameter uses Yahoo! Store®'s graphical fonts, therefore, *font* must either be a variable or property of type font (button-font for example) or the name of one of the graphical fonts in the format of *:font-name*. (such as *:toxica.* Note the colon before and the period after the name of the font).

Font-size: size of the font to be used to render *text* as an image.

Destination: a URL. When specified, the image will be hyperlinked to this URL. The URL can either be entered directly as a string (for instance *"http://www.yahoo.com"*) or obtained from the TO or ACTION operators.

Top-margin, bottom-margin, left-margin, right-margin: margins in pixels. Default to 0 if not specified.

Max-height, min-height, max-width, min-width: used to resize the image. If neither is specified, the image will be rendered in its original size. When an im-

age is resized using these parameters, it will also be resampled. Resampling changes a picture's pixels to match the desired display size, therefore, a resampled image will appear "smooth".

Thickness: if not nil, the image will have a raised border but only if *image* is not specified and *background-color* is specified, and it is other than *transparent*. In other words, when *thickness* is not nil, it causes RENDER to create a button image out of *text*.

Intaglio: when set to anything other than nil, *intaglio* causes text to have a "chiseled" or "incised" appearance.

Crop: one of *:left*, *:right*, or *:center* (this is the default). If *max-width* is smaller than the text rendered as an image, this parameter determines how the text should be cropped.

Expand: Undocumented

The following example demonstrates how RENDER can be used to create uniform buttons by pasting text on top of some image. The example uses the image of a blank button shown in Figure 16 uploaded into a custom variable called *blank-button*. It then superimposes the name of each top-level section over this button image and links the image to the appropriate section page. The resulting navigation bar will be similar to the one shown in Figure 17.

Figure 16 - Blank button image

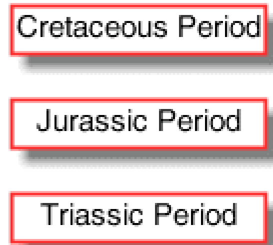


Figure 17 - Custom buttons using RENDER

Example:

```
WITH-OBJECT :index
  FOR-EACH-OBJECT @contents
    WITH-LINK TO id
      IMAGE source RENDER image @blank-button
                          text @name
                          text-align :center
                          max-width 150
            alt @name
      LINEBREAK
```

See also: FUSE, IMAGE

RETURN-WITH

RETURN-WITH stops the evaluation of the current template and returns immediately with the value specified as its argument. The following template—called **Find-100**—will find the first item in the store whose price is at least $100.

Example:

```
Find-100 ()
FOR-EACH-OBJECT WHOLE-CONTENTS
  WITH= variable price
        value ELEMENT position 0
                      sequence IF test @sale-price
                                  then @sale-price
                                  else @price
    WHEN price
      WHEN >= value1 price
              value2 100
        RETURN-WITH id
```

Note that the above task could have been done easier using the FIND-ONE operator.

SEARCH-FORM

Generates a search form with a label, a text box for typing a search phrase, and a search button. This search form can be used to search the store for a word or phrase similarly to the Search page.

Example:

```
SEARCH-FORM label "Find"
            size 12
            button "Go"
```

SEGMENTS

The `SEGMENTS` operator takes a sequence, and returns successive segments of a specified length of that sequence. For a more in-depth explanation, see the discussion of the `SEGMENTS` operator earlier in this book.

See also: FOR-EACH

SELECT

The `SELECT` operator creates a drop-down list. The operator takes three arguments: *name*, *options*, and *selected*. *Name* is the name of the drop-down list, *options* is a sequence whose elements will form the list, and *selected*, if specified, will be the default selection (the pre-selected option from the list.) The `SELECT` operator should only be used inside a `FORM`. For an example, see the built-in **Order.** template. It uses the `SELECT` operator to create a drop-down if an item has *options* specified.

See also: FORM

SET

This operator takes two parameters: a *variable* and a *value*. It then assigns value to *variable*. This operator can be used to reset the value of a local variable (one declared using the WITH= operator.) A special requirement is that the variable used *must* begin with a dollar sign ($). At present, the operator is not available in the editor but it will be released at a later date.

Example:

```
WITH= variable $x
      value "abc"
  TEXT $x
  LINEBREAK
  SET variable $x
      value "5"
  TEXT $x
```

(Please note: the variable used may be of any type not just a text string.)

See also: WITH=

SHIM

Creates a transparent placeholder image. SHIM takes three parameters: *height*, *width*, and *align*. *Height* and *Width* determines the placeholder's size, while *align*—when specified—aligns the placeholder with respect to the surrounding text. For more information, see the *Align* argument of the `IMAGE` operator.

The following example uses the SHIM operator to create a one-pixel wide separator line between two columns of a table.

Example:

```
TABLE border 0
      cellspacing 0
      cellpadding 0
  TABLE-ROW valign :top
    TABLE-CELL
      LINEBREAK number 5
      TEXT "This is the left column"
      LINEBREAK number 5
    TABLE-CELL background-color @button-edge-color
               width 1
      SHIM height 1
           width 1
    TABLE-CELL
      LINEBREAK number 5
      TEXT "This is the right column"
      LINEBREAK number 5
```

SHOPPING-BANNER

The online documentation says: "Inserts the shopping basket banner into the current page. Should be used only in .of files." There is no explanation, however, as to what an .of file is or how to use `SHOPPING-BANNER`.

SORT

This operator sorts a sequence. It takes three parameters, var1, var2, and a sequence. It expects a comparison expression pasted within its body. The comparison expression should compare var1 and var2 and it basically tells the SORT operator how to compare (or sort) any two elements of the sequence. The comparison expression should return a negative number if var1 is less than var2, a positive number if var1 is greater than var2, or zero if var1 is equal to var2. For

this purpose, one of <=>, CMP, STRCMP, or STRCASECMP can readily be used.

Example:

The following example sorts the contents of the current page by price:

```
WITH= variable sorted
      value SORT var1 a
                 var2 b
                 sequence @contents
              <=> n1 WITH-OBJECT a
                       ELEMENT position 0
                               sequence @price
                  n2 WITH-OBJECT b
                       ELEMENT position 0
                               sequence @price
  FOR-EACH-OBJECT sorted
    TEXT @name
    TEXT ", "
    TEXT @price
    TEXT "<br>"
```

See Also: <=>, CMP, STRCMP, STRCASECMP

STRCASECMP

Compares two strings, s1 and s2, case insensitively. It returns a negative number if s1 is less than s2, a positive number if s1 is greater than s2, or zero if s1 is equal to s2. This operator is most commonly used with SORT.

See Also: STRCMP, CMP, <=>, SORT

STRCMP

Compares to strings s1 and s2 case sensitively. Returns a negative number if s1 is less than s2, a positive number if s1 is greater than s2, or zero if s1 is equal to s2. This operator is most commonly used with SORT.

See Also: STRCASECMP, CMP, <=>, SORT

STRING-TRIM

Takes two arguments, *whitechars*, and s. When *whitechars* is set to the intrinsic constant *whitechars*, STRING-TRIM removes any spaces or other non-printing characters from the front and end of the text string *s*.

Example:

```
TEXT STRING-TRIM whitechars whitechars
                 s "    abcde e ef   "
```

The code snippet above would print :

```
abcde e ef
```

SWITCH

The `SWITCH` operator takes one argument, the *switch expression*, and zero or more *key-expression pairs*. It then compares the result of the *switch expression* to each key and if there is a match, the corresponding expression is evaluated and its value returned. If there is no match, `SWITCH` returns nil.

Each expression of a key-expression pair can be exactly one operator. If multiple expressions need to be evaluated when a match is found, use the

MULTI operator to group multiple expressions or put the multiple expressions into a separate template and use the CALL operator to evaluate that template.

Example:

```
SWITCH @page-format
  :top-buttons
  TEXT "You have top buttons."
  :side-buttons
  TEXT "You have side buttons."
```

See also: IF, WHEN

TABLE

Creates a table in the current page. This operator should be used together with TABLE-ROW and TABLE-CELL. If a TABLE doesn't have at least one TABLE-ROW and TABLE-CELL within it, the table will not appear on the current page. This operator is equivalent to the <TABLE> HTML tag.

Only TABLE-ROW operators should be pasted directly within a TABLE, and only TABLE-CELLs within a TABLE-ROW.

The TABLE operator has six parameters covering some but not all the attributes of the equivalent <TABLE> tag in HTML. The parameters are:

Border: a positive integer determining the width of the border of the table. Enter 0 if no border is required. The border attribute is applied to all the cells of the table not only to the outside border. Therefore, if border is other than 0, then the table will look more like a grid.

Align: one of *:left*, *:right*, or *:center*. Determines the alignment of the table relative to the edge of the current page. If *:left* or *:right* is specified, the table will

"float" on the left or right, respectively, which means that any text following the table will be printed on either the left or right side of the table.

Cellspacing: positive integer determining the distance among the table cells in pixels.

Cellpadding: positive integer determining the margin inside the table cells in pixels.

Units: this parameter can be *:em*, *:relative*, or *:pixels*. HTML only uses this attribute if COLSPEC is specified, however, since COLSPEC is not implemented in RTML's `TABLE` operator, the *Units* parameter has no practical use.

Width: specifies the width of the table. Use a positive integer to specify the explicit size of the table in pixels, or enter a percentage inside double-quotes, if a size relative to the browser window is required. For example, to create a table that fills the entire screen; enter "100%" for *width*.

Tables in web sites, and in Yahoo! Store®s in particular, are usually used for layout control, because they allow for the exact positioning of screen elements: banners, navigation buttons, etc.

Example:

```
TABLE border 1
      align :center
      cellspacing 0
      cellpadding 2
      width "80%"
  TABLE-ROW
    TABLE-CELL background-color COLOR red 0
                                      green 255
                                      blue 0
      TEXT "This is a green table cell"
    TABLE-CELL background-color COLOR red 255
                                      green 0
                                      blue 0
      TEXT "This is a red table cell"
```

TABLE-CELL

Creates a table cell inside a table row (see `TABLE-ROW`). Must be pasted within a `TABLE-ROW` operator. `TABLE-CELL` has the following six parameters:

Background-color: color used for the background of the table cell. Must be a variable or property of type color, or the result of the `COLOR` operator.

Align: sets the alignment of the contents (text or images) inside the table cell. Possible values are *:left*, *:right*, and *:center*.

Valign: sets the vertical alignment of the contents (text or images) inside the table cell. Possible values are *:top*, *:middle*, or *:bottom.*

Width: the width of the table cell. This parameter can either be a positive integer specifying the width of the table in pixels, or a percentage (relative to the width of the entire table) within double quotes.

Colspan: determines how many columns the cell should span.

Rowspan: determines how many rows the cell should span.

Example:

See `TABLE` above.

TABLE-ROW

Creates a table row within a `TABLE`. This operator should be pasted within a `TABLE` operator. `TABLE-ROW` has no effect pasted anywhere outside a `TABLE`. This operator has three parameters:

Background-color: color to be used for the entire row. Individual cells inside the row may override this value by specifying their own color. Must be a variable or property of type color or the result of the `COLOR` operator.

Align: sets the horizontal alignment for the cells within the row. Individual cells inside the row may override this value by specifying their own alignment. Possible values are *:left*, *:right*, and *:center*.

Valign: sets the vertical alignment for the cells within the row. Individual cells inside the row may override this value by specifying their own vertical alignment. Possible values are *:top*, *:middle* (this is the default), and *:bottom.*

Example:

See `TABLE` *above.*

TAG-WHEN

Takes two parameters, an HTML tag and a test condition. It evaluates the expressions pasted within its body and if the test condition is true, it encloses the entire output within an open and close tag specified by its *tag* parameter. More simply put, this operator provides a way to conditionally use tags that require an open and a close tag. Examples of such tags include: <CENTER>, <B>, <I>, <SMALL>, etc. By using this operator, it is possible, for instance, to center a paragraph or a sentence if a certain condition is met. The tag parameter may be specified either as a string within double quotes or by using the notation *:tag-name* as in *:center*.

The following example will print the price of the current item in red if the item is on sale. The price will still be printed even if the item is not on sale (no sale-price exists), however, it will not be in red. This example shows another trick when using the TAG-WHEN operator: for the tag parameter, it is possible to provide attributes such as the color attribute of a font.

Example:

```
TAG-WHEN tag "font color=red"
         test @sale-price
  TEXT PRICE number ELEMENT position 0
                            sequence OR
                                       @sale-price
                                       @price
             currency @currency
```

TAXSHIP-MODULE

This operator appears to be experimental. Takes one parameter, the ID of a product. It generates a table detailing the shipping and tax charges for that product from a zip code provided by the user. It shows the shipping charges for all the various shipping methods set up in the store. Unfortunately, TAXSHIP-MODULE only works in the editor (see Figure 18). It generates garbage in the published site (Figure 19).

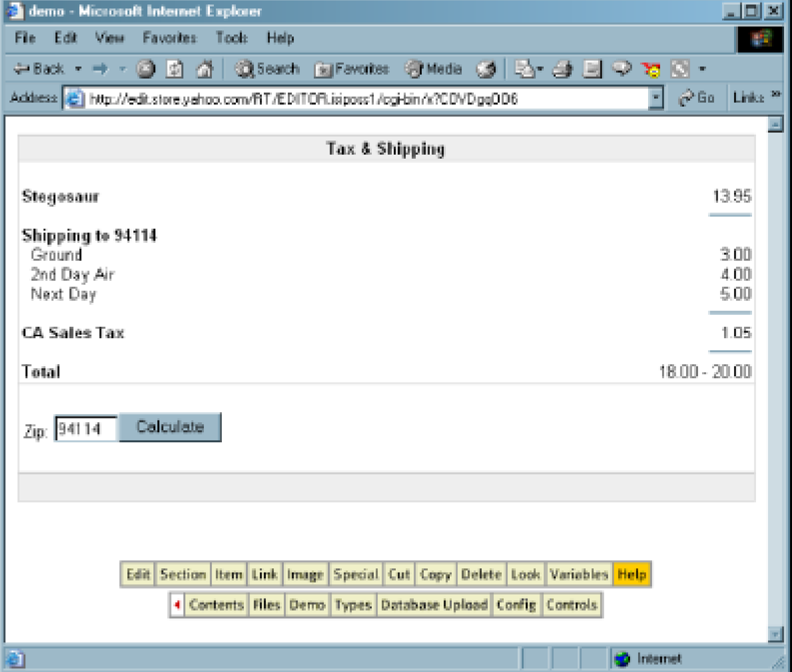


Figure 18 - TAXSHIP-MODULE in the editor

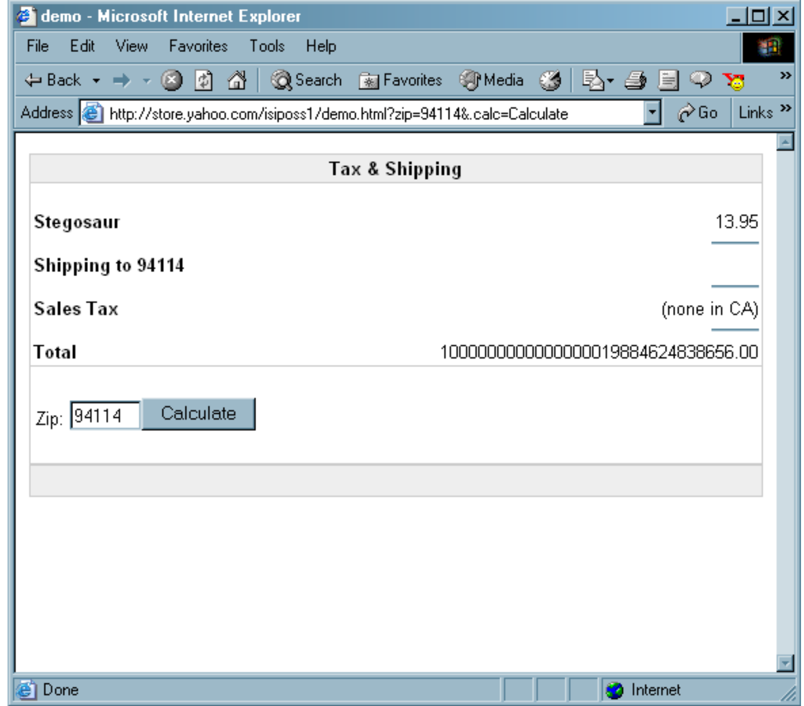


Figure 19 - TAXSHIP-MODULE in the published site

TAXSHIP-ORDER-MODULE

Similar to the `TAXSHIP-MODULE`. Experimental operator, which generates a table detailing the various shipping and tax charges for a product from a user-provided zip code. Takes one parameter, the ID of a product. Unfortunately, this operator only works in the store editor (Figure 20). It produces garbage in the published site (Figure 21).

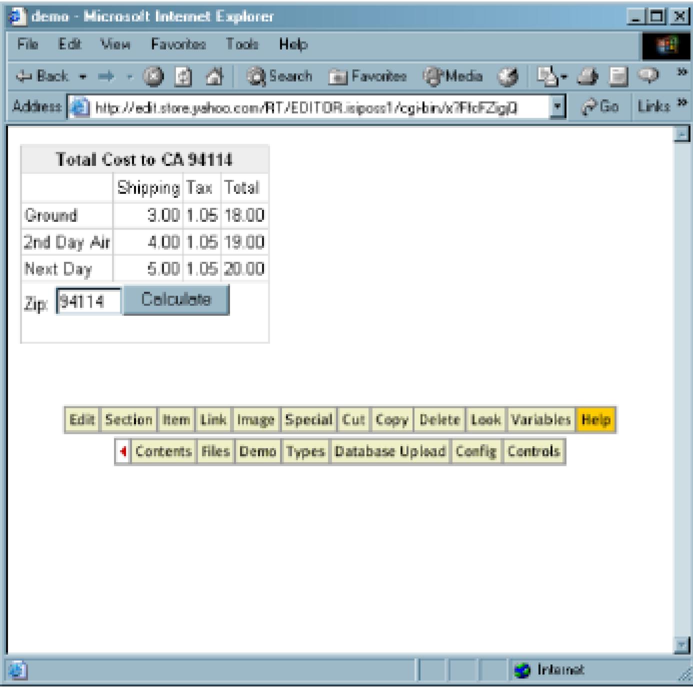


Figure 20 - TAXSHIP-ORDER-MODULE in the editor

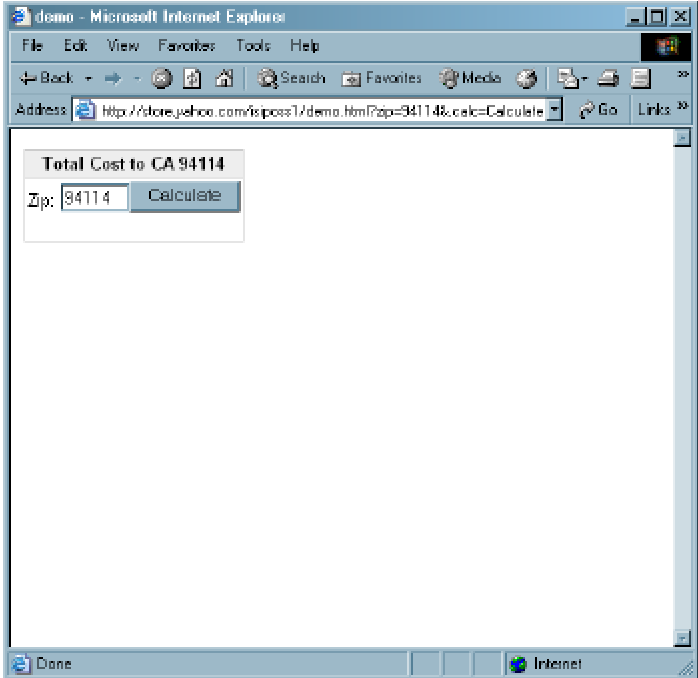


Figure 21 - TAXSHIP-ORDER-MODULE in the published site

TEXT

`TEXT` takes one argument and it prints is on the current page. See **Printing text** on page 90 for more information on the `TEXT` operator.

TEXT-STYLE

Applies an HTML text style to the anything printed by expressions within its body. The possible HTML text styles are: *:bold*, *:italic*, *:typewriter*, *:underline*, *:strike*, *:big*, *:small*, *:subscript*, *:superscript*, and *:blinking*.

The following example will print the regular price crossed out directly above the sale price if the item is on sale. Otherwise, it will simply print the regular price.

Example:

```
IF test @sale-price
   then MULTI
          TEXT @regular-price-text
          TEXT " "
          TEXT-STYLE :strike
            TEXT PRICE number ELEMENT position 0
                                      sequence @price
                       currency @currency
          LINEBREAK
          TEXT @sale-price-text
          TEXT " "
          TEXT PRICE number ELEMENT position 0
                                    sequence @sale-price
                     currency @currency
   else MULTI
          TEXT @regular-price-text
          TEXT " "
          TEXT PRICE number ELEMENT position 0
                                    sequence @price
                     currency @currency
```

See also: FONT

TEXTAREA

Creates a multi-line text box form element. This operator should only be used within the body of a `FORM` operator. Corresponds to the <TEXTAREA> … </TEXTAREA> tags of HTML. The operator takes the following four parameters:

Name: the name of the text box.

Rows: the number of rows the text box is to occupy.

Cols: the number of columns the text box is to occupy.

Wrap: determines how text typed into the text box should wrap. The possible settings are *:soft*, *:virtual, :hard*, *:physical,* and *:off*. The settings *:soft* and *:virtual* will wrap text automatically and will cause a vertical scroll bar to appear on the text area box. These two settings, however will not send actual carriage return characters to the server when the form is submitted. Internet Explorer 4.0 and up assumes *wrap* to be *:virtual* if it is not specified. The settings *:hard* and *:physical* cause text to be wrapped the same way as *:soft* or *:virtual*, but when the form is submitted, the value of the text area box will preserve the actual hard carriage return characters wherever text wrapped. Finally, *:off* causes the text box not to wrap only if the user presses the Enter key. This is the default setting in Netscape.

The `TEXTAREA` operator allows you to paste RTML expressions within its body. The output of any expression pasted within the body of the `TEXTAREA` operator becomes text entered into the text area box when the form is rendered. This method may be used to specify a default value for the `TEXTAREA` field.
The following example demonstrates how to create a simple feedback form. The user can fill in a brief message and when submitted, this message is automatically e-mailed to the e-mail address set in the *email* global variable.

Example:

```
FORM "http://store.yahoo.com/cgi-bin/pro-forma"
  INPUT name :owner
        type :hidden
        value @email
  INPUT name :newnames-to
        type :hidden
        value account
  TEXT "Enter your feedback below:"
  LINEBREAK
  TEXTAREA name "comments"
           rows 10
           cols 30
           wrap :virtual
  LINEBREAK
  INPUT type :submit
        value "Send"
```

See also: FORM, INPUT

TITLE

Names the current page. Corresponds to the <TITLE>…</TITLE> tag of HTML.

See also: HEAD

TO

Takes one parameter, the ID of an object (page) and returns the URL that leads to that page. Possible values are an explicitly specified ID (for example, *:index*), or one of the special variables *id*, *prev*, or *next*. The TO operator is typically used in conjunction with the WITH-LINK operator.

Example:

```
WITH-LINK TO :info
  TEXT "Click here to visit our info page."
```

See also: WITH-LINK, ACTION

TOKENS

The TOKENS operator takes a text string and turns it into a sequence in which each element is a “token” from the original string. Tokens are either single words or phrases enclosed in double quotes and separated by spaces from one another.

The following example will print each word of the sentence “This is how TOKENS works” on a new line:

```
FOR-EACH variable word
         sequence TOKENS "This is how TOKENS works"
  TEXT word
  LINEBREAK
```

At the time this book was written, the only place where `TOKENS` was used in the built-in templates was in the **Order.** template, the template responsible for displaying the price, the options (if there are any) and the order button for each orderable item. In this template, `TOKENS` is used to convert each line of the *options* property into a sequence, so that the second word of each line of the *options* property can be examined whether it includes the word “Monogram” or “Inscription.”

The relevant section of the **Order.** template is listed below.

Example:

```
FOR-EACH variable para
         sequence PARAGRAPHS @options
  WHEN EQUALS value1 @order-style
              value2 :multi-line
    LINEBREAK
  WITH= variable set
        value TOKENS para
    IF test AND
              EQUALS value1 ELEMENT position 0
                                    sequence set
                     value2 "Monogram"
              EQUALS value1 LENGTH set
                     value2 1
       then CALL :monogram.
       else IF test AND
                      > value1 LENGTH set
                        value2 2
                      EQUALS value1 ELEMENT position 1
                                            sequence set
                             value2 "Inscription"
               then CALL :inscription.
                      set
               else MULTI
                      TEXT ELEMENT position 0
                                   sequence set
                      TEXT ": "
                      SELECT name ELEMENT position 0
                                          sequence set
                             options ELEMENTS sequence set
                                              first 1
    TEXT " "
```

See also: FOR-EACH

VALUE

Takes three arguments: *id*, *query*, and *property*. It returns true, if the page whose ID is *id* either has a custom property called *property,* or if that page has

overridden the global variable named *property*. Currently, the only possible value for *query* is *:local*. Property names specified by the *property* parameter must be in the form *:property-name*.

Example:

```
IF test VALUE id id
              query :local
              property :keywords
   then TEXT "The keywords variable has been overridden on this page."
   else TEXT "The keywords variable has not been overridden on this page."
```

WHEN

The `WHEN` operator is very similar to the `IF` operator. It is basically "one half" of the `IF` operator: it says "evaluate the following if this expression is true (not nil.)" It is equivalent to the following `IF` block:

```
IF test <some expression>
   then <some other expression>
   else nil
```

The `WHEN` operator has a single argument, a condition. If the result of the condition is true (not nil) then the expression or expressions pasted ***within*** the `WHEN` block is evaluated. Similarly to the arguments of the `IF` operator, the condition argument of the `WHEN` operator can only be a single operator but may contain any simple or complex expression. To enter a more complex expression (one that consists of more than one operator), use the `CALL` operator to evaluate another template or the `MULTI` operator.

The following example will print, "This item is on sale." if the current page has a sale price entered and will do nothing otherwise.

Example:

```
WHEN @sale-price
  TEXT "This item is on sale."
```

See also: IF

WHOLE-CONTENTS

Returns a list of the IDs of all the regular objects (items and sections) in alphabetical order by their names. The following example (a snippet from the built-in **Index-body.** template) generates a list of all the items that have their own pages (i.e. whose template is not nil). Along the same lines, one could construct a template that would list each item on sale.

Example:

```
FOR-EACH-OBJECT WHOLE-CONTENTS
  WHEN AND
         NONEMPTY @name
         @template
    WITH-LINK TO id
      TEXT @name
    LINEBREAK
```

See also: FOR-EACH

WIDTH

`WIDTH` returns the width of an image in pixels. The image passed to the `WIDTH` operator must be an image that is already rendered, i.e., one that was

returned by the RENDER or FUSE operators. Passing a variable of type image (such as the *name-image*) to WIDTH returns nil.

See also: HEIGHT

WITH-LINK

WITH-LINK creates a hyperlink. It takes one parameter, the URL of a page, which can either be a string (as in "http://shopping.yahoo.com") or the result of a TO or an ACTION operator. The expressions pasted ***within*** the body of WITH-LINK will become the hyperlinked item (image, or text).

Example:

```
WITH-LINK TO :info
  TEXT "Click here to visit our info page."
```

See also: TO, ACTION

WITH-OBJECT

Takes one parameter, the ID of an object. The expressions within `WITH-OBJECT` will be evaluated in the context of that object. This operator can be used to change the context from the current object to any other object.

Example:

```
TEXT "The name of this page is "
TEXT @name
LINEBREAK
TEXT "The name of the search page is "
WITH-OBJECT :nsearch
  TEXT @name
```

See also: FOR-EACH-OBJECT

WITH=

`WITH=` defines a local variable. This operator takes two parameters, a variable name and a variable value. The variable defined this way can be referenced by any expression pasted ***within*** the body of `WITH=`. Local variables are called local because their scope only includes the current template (and current `WIDTH=` block). In other words, a variable defined by the `WITH=` operator is only valid within the template in which it was defined and only within the body of the `WITH=` operator that defined it.

Multiple local variables may be defined by pasting multiple `WITH=` operators within one another.

Example:

```
WITH= variable a
      value 2
  WITH= variable b
        value 5
    TEXT "a * b = "
    TEXT a * b
```

See also: SET

WORDBREAK

`WORDBREAK` is the equivalent of HTML's <WBR> tag. It tells the web browser where the current line is allowed to be broken into the next line. It should only be used within a `NOBREAK` block. Outside of a `NOBREAK` block, `WORDBREAK` has no effect. `WORDBREAK` does not explicitly cause the line to be broken; `LINEBREAK` is used for that purpose.

Example:

```
NOBREAK
  TEXT "This is a long line, which is not going to be wrapped when "
  TEXT "the browser is resized. However, the rest of this sentence "
  WORDBREAK
  TEXT "may wrap if the browser window is resized."
```

YANK

YANK has two parameters: a sequence and an element. It returns the same sequence but with all occurrences of the given element removed. Note that YANK only works on lists; it does not work on strings.

The following example creates a list similar to the "Index" page (site map). Unlike the index, this page will list only sale items—pages that have a sale-price and a template. If an object has no template, it cannot have its own page either, so those will be excluded. Each item will also be hyperlinked to its own page.

Example:

```
FOR-EACH-OBJECT YANK element nil
                     sequence FOR-EACH-OBJECT WHOLE-CONTENTS
                                WHEN AND
                                       @template
                                       @sale-price
                                  id
  WITH-LINK TO id
    TEXT @name
  LINEBREAK
```

See also: APPEND, MAKE-LIST

YFUNCTION

Undocumented. Used to call functions internal to the RTML engine.

Appendices

Appendix A: RTML Resources

- Yahoo! Store Users Group – http://www.ystoreforums.com
- "RTML Tips & Tricks" – a searchable online database at http://www.siposs.com/rtml
- Yahoo!'s own RTML tutorial - http://store.yahoo.com/rtml.html
- Yahoo! Store Templates – http://www.rtmltemplates.com
- Y-Times Publications: www.ytimes.info

Appendix B: Yahoo! Store® Standard Graphical Fonts

The following is a *complete* list of graphical fonts that can be used with the *RENDER* operator or the *vw-img* special tag. There is a list of fonts online at http://store.yahoo.com/vw/fonsam.html, however, that list does not contain all the fonts that can be selected from the standard font selection menu (Figure 14, page 114).

Ad-Lib.

Alexa.

Arnold-Boecklin.

Avantgarde-Oblique.

Avantgarde-Demi.

Avantgarde.

Bank-Gothic-Medium.

BALLOON-EXTRA-BOLD.

BALLOON-BOLD.

Berliner-Grotesk-Bold.

Berthold-Baskerville-Regular.

Berthold-Baskerville-Medium-Italic.

Berthold-Baskerville-Medium.

Berthold-Baskerville-Italic.

Bookman-Demi-Italic.

Bookman-Demi.

Bookman-Light.

Bookman–Light–Italic.

Boton–Regular.

Boton–Italic.

Boton–Medium.

Boton–Medium–Italic.

Boulevard.

Castle-Bold.

Castle-Book.

Chancery–Italic.

Chantilly–Bold.

Chantilly–Bold–Italic.

Chantilly–Regular.

Chantilly–Italic.

City-Medium.

Cooper–Heavy.

COPPERPLATE–GOTHIC.

COPPERPLATE–GOTHIC–BOLD.

COPPERPLATE–GOTHIC–CONDENSED–BOLD.

Courier.

Courier-Bold.

Courier-Bold-Oblique.

Courier-Oblique.

Dom–Bold.

Fette–Gotisch.

Franklin–Gothic–Bold.

Franklin-Gothic-Bold-Italic.

Franklin-Gothic-Cond-Bold.

FRANKLIN-GOTHIC-COND-SM-CAPS.

FRANKLIN-GOTHIC-COND-SM-CAPS-BOLD.

FUNCTION-SM-CAPS-REGULAR.

Garamond.

Garamond-Bold.

Garamond-Medium-Italic.

GARAMOND-SM-CAPS.

GARAMOND-SM-CAPS-BOLD.

Geometric-415-Medium.

Handel-Gothic.

Helvetica.

Helvetica-Bold.

Helvetica-Bold-Oblique.

Helvetica-Narrow.

Helvetica-Narrow-Bold.

Helvetica-Narrow-Oblique.

Helvetica-Narrow-Bold-Oblique.

Helvetica-Oblique.

Humanist-521-Condensed.

Impact.

Impress.

ITC-ANNA.

Kaufmann-Bold.

Kids-Plain.

Legend.

Letter-Gothic-12-Pitch-Bold.

Letter-Gothic-12-Pitch-Bold-Italic.

Letter-Gothic-Bold.

Letter-Gothic-Bold-Oblique.

Letter-Gothic-Extra-Bold.

Liberty.

LITHOS-BLACK.

LITHOS-BOLD.

LITHOS-REGULAR.

MACHINE-BOLD.

MACHINE.

Metaplus-Book-Roman.

Metaplus-Book-Italic.

METAPLUS-BOOK-CAPS-ITALIC.

METAPLUS-BOOK-CAPS.

Metaplus-Bold-Roman.

Metaplus-Bold-Italic.

METAPLUS-BOLD-CAPS-ITALIC.

METAPLUS-BOLD-CAPS.

Mezz

Mona Lisa.

Ocr-A.

Ocr-B.

Palatino.

Palatino–Bold.

Palatino–Bold–Italic.

Palatino–Italic.

Poppl–Laudatio–Italic.

Poppl–Laudatio–Medium.

Poppl–Laudatio–Medium–Italic.

Poppl–Laudatio–Regular.

Post–Antiqua–Roman.

Raleigh–Bold.

STENCIL.

STOP.

Swiss–721–Bold–Rounded.

Symbol.

Syntax–Bold.

Tekton–Mm.

Tempo–Heavy–Condensed.

Tempo–Heavy–Condensed–Italic.

Times.

Times–Bold.

Times–Bold–Italic.

Times–Italic.

Toxica.

Vag–Rounded.

Zapf–Dingbats.

Index

<

< · 121
<= · 122
<=> · 120

>

> · 122
>= · 122

A

Accessories · 20
ACTION · 123
Advanced Interface · 2, 3, 17, 53
AND · 63, 124
APPEND · 125
arguments · xiii
AS-LIST · 125
ASP · xv, 8
AUCTIONURL · 126

B

BASKET · 126
BASKET-MODULE · 126
Big-text · 12
BLUE · 112, 126
BODY · 127

C

`CALL` · 41, 129
CALL operator · 32, 41, 42, 46, 69, 129, 148, 180, 196
CAPS · 129
CENTER · 130
Client-side scripting · xv
CMP · 130, 178, 179
COLOGO · 130
`COLOR` operator · 103, 110, 112, 115, 132, 136, 142, 171, 182, 183
Color type · 13
Colors · 109

COMMENT · 132
Conditionals · 67
Constants · 59
Contents List · 3, 27
context · 54
CSV · xviii, 6, 18, 21
CSV Database Upload · xviii

D

Database · xvii
Database upload · 2, 3
Default property values · 16
DreamWeaver · 8
dynamic content · xvii

E

ELEMENT · 83, 133
ELEMENTS · 82, 133
empty. · 7
EQUALS · 134
EVEN · 135
expression · xiii
Expression Stack · 25, 26, 27, 29
Expressions · 63

F

File upload · 2, 3
FIND-ONE · 78, 135
Font · 115
FONT · 136
Font type · 14
FONT-WIDTH · 137
FOR · 72, 138
FOR-EACH · 75, 139
FOR-EACH-BUT · 76, 139
`FOR-EACH-OBJECT` · 55, 56, 77, 85, 140, 199
FORM · 140
`FUSE` operator · 60, 62, 103, 104, 106, 128, 141, 142, 149

G

GET-ALL-PATHS-TO · 143, 144, 145
GET-PATH-TO · 144, 145
Global variables · 48
GRAB · 145
GRAYSCALE · 112, 146
GREEN · 112, 146

H

HEAD · 146
HEIGHT · 107, 147
HEX-ENCODE · 147

HRULE · 147
HTML · xiii
hyperlinks · 4, 34, 105, 128

I

ID · 4, 53
Ids type · 14
IF · 67, 148
IMAGE · 101, 149
Image type · 15
IMAGE-REF · 151
info page · 3, 6, 33
info. · 9
INPUT · 152
INVENTORY-INFO · 154
ITEM · 154
item. type · 10
ITEM-INVENTORY · 154
Iteration · 72

L

LENGTH · 83, 156
LINEBREAK · 156
LINES · 94, 156
link. type · 10
LIVE · 157
Local variables · 49, 199
LOCAL-LOGO · 157
Logical Expressions · 63
LOOKUP · 157
LOWERCASE · 158

M

main. · 7, 9, 10, 61
MAXIMUM · 158
META · 159
MINIMUM · 159
MODULE · 160
MULTI · 161

N

Nested variables · 52
Next · 56
NOBREAK · 162
NONEMPTY · 84, 162
norder. · 7, 8, 61
NOT · 65, 163
NUMBER · 163
Number type · 13
Numbers type · 13

O

Objects type · 14
OBJID-FROM-STRING · 164

operator · xiii
OR · 64, 165
ORDER · 166
Orderable type · 16
ORDER-FORM · 167

P

PARAGRAPH · 167
PARAGRAPHS · 93, 168
Parameters · 52
PHP · 8
POSITION · 84, 169
Positive-integer · 13
Prev · 56
`PRICE` · 98, 169
Printing Text · 90
privacypolicy. · 9
Properties · 47

R

raw-html. · 7
RED · 112, 170
References type · 15
Regular Interface · 2
`RENDER` operator · 102, 103, 104, 105, 106, 108, 116, 141, 150, 151, 170, 205
RETURN-WITH · 174
REVERSE · 165
RGB · 109, 126

S

search. · 7, 9, 61
SEARCH-FORM · 35, 36, 154, 174
section. type · 10
SEGMENTS · 85, 175
SELECT · 175
Sequences · 82
SET · 175
Sever-side scripts · xv
SHIM · 176
shipping and tax calculation · xvi
shopping cart · xiv, xvi, xvii, 3, 8, 106, 123, 141, 166, 167
SHOPPING-BANNER · 177
Simple Interface · 2
SORT · 120, 121, 130, 177, 178, 179
STRCASECMP · 130, 178, 179
STRCMP · 130, 178, 179
SWITCH · 70, 179
Symbol · 13

T

TABLE · 180
TABLE-CELL · 35, 36, 43, 167, 180, 182
TABLE-ROW · 43, 44, 167, 180, 182, 183
TAG-WHEN · 184
TAXSHIP-MODULE · 185
TAXSHIP-ORDER-MODULE · 187
template · xiii
Template Editor · 22, 29, 32
Template Editor Toolbar · 29
Template parameter · 45
Template property · xvii, xviii, 46, 47
TEXT · 90, 189
Text type · 12
TEXTAREA · 190
TEXT-STYLE · 115, 116, 137, 189
TITLE · 193
TO · 193
TOKENS · 86, 193
Types · 6

U

Up · 56

V

VALUE · 195
VW-IMG · 116

W

web safe colors · 109
WHEN · 69, 196
WHOLE-CONTENTS · 78, 79, 88, 197
WIDTH · 107, 197
WITH= · 199
WITH-LINK · 85, 105, 106, 123, 126, 142, 193, 198
WITH-OBJECT · 55, 56, 199
WORDBREAK · 200

Y

YANK · 88, 201
Yes-No · 14
YFUNCTION · 201