stilo
content engineering

Beginner’s Guide to OmniMark

Beginner’s Guide to OmniMark

About OmniMark

OmniMark is a streaming programming language. As a starting point, you can think of OmniMark as either a rule-based language or an event-based language. If you have ever programmed for a graphical user interface such as Windows, the Mac, or Motif, you are used to event-based programming. In these environments, the operating system captures user actions such as keystrokes or mouse clicks, or hardware actions such as data arriving on a serial port, and sends a message to the current program indicating what event has occurred. Programs consist of a collection of responses to events. The order in which program code is invoked depends on the order in which events occur.

OmniMark programs are written the same way, as a collection of responses to events. The difference is that the events an OmniMark program responds to are not user events or hardware events, but data events. Data events occur in streams of data. As a streaming language, the management of streams is built into the heart of OmniMark. OmniMark shields you from the details of stream handling just as good GUI programming languages shield you from the details of user input handling and window management.

What is a data event? Quite simply, a data event is something significant occurring in a stream of data. In a typical GUI environment, it is the operating system and its associated hardware that decides what is an event. There is a defined set of events, and programs simply have to respond to those events that interest them. Who decides what is an event in a stream of data? You do.

This is where the rule-based aspect of OmniMark comes into play. An OmniMark program consists of rules that define data events, and actions that take place when data events occur. Suppose you wanted to count the words in the text "Mary had a little lamb". You would write an OmniMark rule that defined the occurrence of a word as an event:

```
  find letter+
```

This is an OmniMark find rule. Find rules attempt to match patterns that occur in a data stream, and if they match something completely, they detect an event. This rule matches letters. The "+" sign after the keyword "letter" stands for "one or more", so this rule will go on matching letters until it comes to something that is not a letter, such as punctuation or a space. Having run out of letters, it will see if it needs to match anything else. Since it doesn't, the pattern is complete and the rule is fired. Any actions following the rule are then executed. This rule will fire once for every word in the data, so all that remains to do is increment a counter each time the rule is fired.

If you are used to other languages such as C or Visual Basic, you are probably thinking that there is something odd about the find rule above. Sure, it finds words, but what does it find them in? Where is the reference to the file or variable that contains the data?

Because it deals primarily with events happening in data, OmniMark maintains a current input. Rules automatically apply to the current input, so you don't have to specify what each rule applies to.

Similarly, OmniMark has a built-in output. All output goes to current output. If you need to change were output goes, you change the destination of current output.

Of course, you are not restricted to using only a single input or output. You can define and use a variety of inputs and outputs, as well as variables. But in OmniMark you generally do not have to concern yourself with opening files, reading the content into variables, and stepping through the content as you would in other languages. In OmniMark, you just name the desired input source and let the data flow; thus, a complete program to count the words in "Mary had a little lamb" looks like this:

```
global counter wordcount initial {0}

process
   submit "Mary had a little lamb"
   output "d" format wordcount || "%n"

find letter+
   increment wordcount
```

The `output` statement uses the format operator, (`"d" format wordcount`), to convert the value of the counter "wordcount" to a string and uses the concatenation operator `||` to add a new line, represented by "%n".

To try this program, copy it to a file named "test.xom" and type the following on the command line:

```
omnimark -s test.xom
```

The example above introduces a new kind of rule, the process rule. The process rule, as you would expect, is fired when processing begins. Our program consists of one global variable declaration and two rules. Note that, in this program, it doesn't matter in what order the rules appear since each fires only when a specific event occurs. Thus we could just as easily write the program:

```
global counter wordcount initial {0}

find letter+
   increment wordcount

process
   submit "Mary had a little lamb"
   output "d" format wordcount || "%n"
```

This program runs just the same as the first. This is not to say that the order of rules never matters in an OmniMark program. If one event could cause more than one find rule to fire, the rule that occurs first will fire, and the one that occurs later will not. This allows you to put more specific rules before more general rules and have the general rules fire only if the specific rule does not. The following two programs produce different output:

```
global counter wordcount initial {0}

process
   submit "Mary had a little lamb"
   output "d" format wordcount || "%n"

find "had"
   output "*"

find letter+
  increment wordcount

find any
```

The program above prints "*4". The program below changes the order of the find rules and produces a different output.

```
global counter wordcount initial {0}

process
   submit "Mary had a little lamb"
   output "d" format wordcount || "%n"

find letter+
   increment wordcount

find "had"
   output "*"

find any
```

This program prints "5".

Why did we add "find any" as a new rule in both these programs? Actually, it fixes an error in all the earlier versions of our word counting program. The rule "find letter+" matches words. But what about the spaces between the words? What was happening to them? If you actually ran the first program, you might have noticed that it printed its result indented by four spaces. Those are the unmatched spaces from our input. Any input that is not matched by a find rule goes right through to output; "find any" at the end of a set of find rules soaks up any unmatched input. Of course, if you use "find any" it must always be the last find rule.

We said that, in OmniMark, you define data events. This is not always true. Sometimes the data itself contains the definition of the event. Documents written in formal markup languages based on SGML and XML contain tags which break the document up into a set of elements. In such a document, the occurrence of such an element constitutes a data event. Because OmniMark has built-in parsers for XML and SGML, you don't need to worry about how elements are recognized, you just need to write rules to process them when they occur. These are called markup rules.

Pattern and markup processors

The OmniMark language has extensive information processing capability built in. This functionality is packaged into two processors, the pattern processor and the markup processor. In other languages you would have to code this functionality yourself.

The pattern processor provides pattern-matching functionality, and operates on a stream of bytes (which may be text or binary data). The pattern processor works with find rules, detecting an event whenever a pattern defined by a find rule occurs in the input stream. It then fires that rule.

The markup processor provides markup recognition for markup languages created using XML or SGML. The markup processor works with markup rules, detecting an event each time an element or other structure defined by markup occurs in the input stream. It then fires the markup rule associated with the event detected.

Note that the markup processor detects and reports the elements defined by markup, not the text of the markup itself. Thus, in the string "Mary had a little <animal>lamb</animal>" the markup processor will fire the markup rule `element animal` and will report that the data content of the element is "lamb". It will not report that it found the markup strings "<animal>...</animal>". In XML, you can safely assume that it did in fact encounter that markup, but in SGML, which allows for shortened forms of tags and for complete omission of tags in some cases, several different combinations of markup can represent the same element and cause the same rule to fire.

If you wanted to process the text of the markup, as opposed to the structure defined by the markup, you would use the pattern processor. In fact, you can write your own markup processor using the pattern processor. This is useful for converting markup that is not compatible with XML or SGML into XML or SGML form.

How do you use the pattern and markup processors? Simply direct input to the appropriate processor. You direct input to the pattern processor with `submit`. For example, the following code fragment sends a file called names.txt to the pattern processor.

```
  submit file "names.txt"
```

The OmniMark actions `do sgml-parse` and `do xml-parse` direct input to the markup processor. For example, the following code fragment sends a file called myfile.xml to the XML parser of the markup processor.

```
  do xml-parse document
     scan file "myfile.xml"
     output "%c"
  done
```

You can also direct input to the pattern or markup processors using OmniMark's aided translation types.

Once you direct input into one of the processors, OmniMark processes the entire input, firing rules as they occur. If, in responding to an event, you perform an action that submits new input to one of the processors, the current input is suspended, and the new input is processed. When the new input has finished processing, OmniMark resumes processing the original input.

This feature has many uses. For instance, you could read a list of names of files containing XML markup, and open and process each file in turn:

```
process-start
   submit file "names.txt"

find [any except white-space]+=> filename
   do xml-parse document
      scan file filename
      output "%c"
   done

find white-space+
;absorb leftover white space in names.txt

element ...
```

(Note that the find rule for filenames is pretty rudimentary. It's fine if you know the structure of the data you're reading, but don't take this as a general method for identifying filenames!)

Event handling

In OmniMark, data events occur in the processing of the current input. What kind of event occurs depends on which processor, the pattern processor or the markup processor, is presently acting on the current input. Events occurring in the pattern processor cause find rules to fire. Events occurring in the markup processor cause markup rules to fire.

To handle an event, you attach actions to the appropriate find or markup rule. Processing continues until the input is exhausted. During the processing of an event, however, you may do something that starts processing a new piece of input. In this case, processing of the original input is suspended while the new input is processed. This is where things begin to get interesting, because, by default, this new input will be processed by exactly the same rules as were processing the original input. Consider the following program:

```
process
   submit "Mary had a little lamb"

find "had"
   submit "Joe bought a big lamb"

find "lamb"
   output "sheep"
```

This program will output "Mary Joe bought a big sheep a little sheep". The `submit` in the rule `find "had"` suspended processing of the originally submitted data. The find rule for "lamb" fired once for the "lamb" in the second input. When processing of the second submit finished, the first resumed and the "lamb" rule fired once again for the "lamb" in the first input. Can you see why writing code like the following is not a good idea? (The second submit now reads "Joe had a big lamb".) If you feel compelled to try it, save your other work first:

```
process
   submit "Mary had a little lamb"

find "had"
   submit "Joe had a big lamb"

find "lamb"
   output "sheep"
```

When pattern processing, your input is processed completely, unless you decide to pause and begin processing something else in the middle. When processing markup, however, you will have to deal with every level of nesting in the markup. Every element has content, which may include other elements. When an element occurs you have to deal with three things: the start of the element, its content, and its end. To give you the opportunity to decide how and when to deal with the element content, markup processing stalls at each element, and you need to explicitly get it going again.

How do you continue parsing? The OmniMark keyword `%c` causes parsing to continue. You may think of it as equivalent to "continue xml-parse" or "continue sgml-parse". (Since you will always want to continue parsing in the middle of the output of the current element, the parse continuation operator takes a form ("%c") that can easily be dropped into a text string.). A `do xml-parse` or `do sgml-parse` simply sets up the parser in the appropriate initial state to process the input it is receiving. To actually start parsing you must output the parse continuation operator ("%c"):

```
do xml-parse document
   scan file "fred.xml"
   output "Beginning %c End")
done
```

Every markup rule must output the parsing continuation operator ("%c") or its alternative, `suppress`, which continues parsing, but suppresses output of the parsed content. Without them, your program would stall permanently.

For obvious reasons, you cannot use `%c` or `suppress` more than once in a markup rule. Don't fall into the trap of thinking of `%c` as standing for the content of an element. It does not. It is an instruction to continuing parsing that content. Any output that appears in the place where `%c` occurs in an output string is created by the rules which fire as a result of parsing element content.

Can you mix pattern processing and markup processing in the same program? Certainly you can. Consider the following code:

```
element fred
   submit "Element fred contains: %c That's all."
```

The result of parsing the content of element fred will be inserted into the string at the position `%c` occurs. The entire string will then be submitted to the find rules.

Copyright©1988-2004 Stilo International plc

Input and output

Stream processing is at the heart of OmniMark. As a streaming language, OmniMark handles input and output at the core of the language. OmniMark maintains a current input and all recognition of data events is performed on the current input. Similarly, OmniMark maintains a current output at all times, and all output is directed to the current output. This greatly simplifies processing, since you never need to specify what data an action operates on or what destination output goes to. You simply set the current input and output to the appropriate streams and go.

OmniMark defines a number of default streams that correspond to the standard input, standard output, and standard error streams of the underlying operating system.

(Windows programmers may not be familiar with these concepts, which apply in command-line environments. Standard input is where input comes from by default: in most cases, the keyboard. Standard output is where output of a program goes by default, usually to the screen. Standard error is where error messages go by default, usually to the screen. Some operating systems provide sophisticated facilities for manipulating standard input, output, and error. A Windows DOS box provides standard input, output, and error. Under Windows, OmniMark runs in a DOS box.)

In OmniMark, #process-input is a stream bound to standard input, #process-output is a stream bound to standard output, and #error is a stream bound to standard error.

#main-input is a stream that is bound to standard input unless you specify an input file or files on the command line, in which case, #main-input is bound to those files. Similarly, #main-output is bound to standard output unless you specify an output file on the command line, in which case, #main-output is bound to that file.

Of course, you don't have to deal with #main-input and #main-output at all if you don't want to. You can always explicitly assign current input and current output to files from within your code. Since OmniMark has no direct user interface functions, however, the command line is the principal way to pass input and output file names into a batch style OmniMark program.

Server style OmniMark programs communicate with a variety of clients over TCP/IP networks and can receive file names and other instructions from the client. Since OmniMark servers run in the background, there is often no point in dealing with any local input and output streams. If you are writing a server you may wish to disable all the default input and output streams with declare no-default-io.

To explicitly assign current input to a file use submit, do sgml-parse, or do xml-parse with the file modifier:

```
submit file "mary.txt"

do xml-parse document
   scan file "mary.xml"
   output "%c"
```

```
    done
```

You can even assign multiple files to be processed sequentially:

```
    submit file "mary.txt" || file "lamb.txt"

    do sgml-parse document
       scan file "sgmldec.sgm" || file "rhymes.dtd" || file "mary.xml"
       output "%c"
    done
```

All other OmniMark actions that initiate text or markup processing, such as do scan, also accept files in just the same way. If you can process a string, you can process a file by replacing the stream variable or literal string with the keyword `file` and the name of the file. In all cases, doing so sets the current input to the named file.

Copyright©1988-2004 Stilo International plc

Output

All output from an OmniMark program goes to OmniMark's built-in current output. You do not usually need to explicitly state where you want output to go, you just output it and it goes to current output. When you do state a destination for output, you are, in effect, resetting current output to the named destination and outputting to current output.

You can change the destination of current output within a rule:

```
element lamb
   local stream mary
   open mary as file "mary.txt"
   using output as mary
   do
      output "ba ba ba %c"
   done
```

This rule temporarily changes the current output to the file mary.txt. Any output that occurs in the `do...done` block following `using output as` goes to the new destination. Once the block is finished, current output reverts to its original destination. Note, however, that the output statement contains the parse continuation operator (`%c`). Is the new output destination in effect for all the processing that occurs as part of the parsing of the lamb element? Yes it is. Output that is generated by any rules that fire as a result of parsing the lamb element will go to the file "mary.txt".

To understand how this works, consider an XML file, "mary.xml", that contains a valid DTD and the following markup:

```
<line>
<person>Mary</person> had a little <person>lamb</person>
</line>
```

And consider the following OmniMark program:

```
global stream words
global stream people

process
   open words as file "words.txt"
   open people as file "people.txt"
   using output as words
   do xml-parse document
      scan file "mary.xml"
      output"%c"
   done

element line
   output "%c"

element person
   using output as people
```

```
     do
        output "%c "
     done
```

Running this program will leave you with two files: words.txt will contain " had a little ", and people.txt will contain "Mary lamb ". Look closely at this code to make sure you understand which output destination is in effect in the "line" element rule. Make sure you understand why the output ended up in the files it did. If you are comfortable with this, you know most of what you need to know about how OmniMark handles output.

The last rule of the program above can be shortened slightly by using `put` as a shorthand for the `using output as` block:

```
  element person
     put people "%c "
```

OmniMark's current output is a powerful mechanism for simplifying code by eliminating the need to always state where output is going. Once the destination of the current output is set, all output goes to that destination unless you explicitly send it elsewhere. Current output has the additional feature of being able to have more than one destination at a time:

```
  global stream my-file
  global stream my-buffer

  process
     open my-file as file "myfile.txt"
     output-to my-file
     submit "Mary had a little lamb"

  find "had"
     open my-buffer as buffer
     output using my-buffer and #current-output
     do
        output "I've been had!"
     done
```

This code will place "I've been had!" in both the file myfile.txt and in the variable my-buffer. `#current-output` stands for all the current destinations of current output, so you can use it to add a new destination to all those currently active (even if you don't know what they are).

Variables

You can create variables in your OmniMark program. Variables may be of any supported data type.

Variables can be either `global` or `local`. The difference between these is that global variables exist (and can be used) everywhere within a program, and local variables only exist (and can be used) within the rule or function where they are declared.

Since variables cannot be used until they are declared, global variable declarations usually appear at the top of an OmniMark program, and local variables appear at the beginning of the rule or function in which they are to be used. The "scope" of a variable (global or local) must be indicated in the variable declaration.

A variable declaration that creates a global counter variable named "count1" looks like this:

```
  global counter count1
```

Once declared, the variable "count1" can be used to store any positive or negative integer value.

To create a local stream variable named "quotation", you would use the variable declaration:

```
  local stream quotation
```

To store a string in a stream variable, you can use the `set` keyword. For example:

```
  set quotation to "Is this a dagger I see before me?"
```

Counter variable values can be set and changed the same way as stream variables using the `set` action, but counter variables can also be manipulated using the `increment` and `decrement` actions. For example, to increase the value of the `count1` variable by 1, you need only say:

```
  increment count1
```

It is possible to increment or decrement the value of a counter variable by the value of another counter variable. For example, you could decrement the value of "count1" by the value of "count2" with the following code:

```
  decrement count1 by count2
```

The following is a program that makes use of a global switch variable to decide which output action should be executed:

```
global switch question

process
   set question to true

process
   do when question  ;checks if question is true
      output "to be"
   else
      output "not to be"
   done
```

Note that the output of this program will always be "to be".

It is possible to declare a variable with an initial value:

```
global counter count2 initial {3}
global stream quotation2 initial {"A horse!"}
global switch status2 initial {true}
```

You can set a variable to the value of another variable. For example, the process rule in the following program will set the value of the global counter variable "var1" to the value of the local counter variable "var2" and give you the output "8":

```
global counter var1

process
   local counter var2
   set var2 to 8
   set var1 to var2

process
   output "%d(var1)"
```

I/O and variables

Like most languages, OmniMark has actions that assign values to variables (`set`) and actions that read data from and write data to files (`open`, `put`, `close`). Unlike most languages, OmniMark lets you perform file operations with the variable assignment actions, and change the values of stream variables with the file actions. For example, you can place a simple value in a file with the `set` action:

```
  set file "mary.txt" to "Mary had a little lamb"
```

And you can use open, put, and close to set the value of a variable:

```
  local stream Mary
  open Mary as buffer
  put Mary "Mary had a little lamb"
  close Mary
```

How is this magic performed? Simply, in fact, because the variable assignment syntax (`set`) is simply a shorthand version of the file operation syntax. That is, `set Mary to "Mary had a little lamb"` is equivalent to:

```
  open Mary as buffer
  put Mary "Mary had a little lamb"
  close Mary
```

The virtue of using the longer syntax is that you can put off closing the stream until later and write to it many times. This is much easier and more efficient than building up a string by a series of concatenations. So you can replace code like this:

```
  set Mary to "Mary had a little lamb"
  set Mary to Mary || "Its fleece was white as snow"
  set Mary to Mary || "And every where that Mary went"
  set Mary to Mary || "The lamb was sure to go"
```

with code like this:

```
  open Mary as buffer
  put Mary "Mary had a little lamb"
  put Mary "Its fleece was white as snow"
  put Mary "And every where that Mary went"
  put Mary "The lamb was sure to go"
  close Mary
```

You can also make your variable the temporary current output so that everything sent to output goes into that variable:

```
  open Mary as buffer
  using output as Mary
  do
     output "Mary had a little lamb"
     output "Its fleece was white as snow"
     output "And everywhere that Mary went"
     output "The lamb was sure to go"
  done
```

This is an enormously powerful feature of OmniMark. It enables you to choose the type of data assignment mechanism appropriate to the scale of operation you want to perform. You can use `set` for any kind of small scale assignment, whether to a file or a variable, without any of the bother of opening files or buffers. For large-scale operations, you can use file type operations with any file or stream variable and perform multiple updates without the need to specify the destination, or even worry about the kind of destination involved. Choosing the method appropriate to the scale of operation you are performing will greatly simplify your code.

A stream must be closed before it can be read or output:

```
  open Mary as buffer
  put Mary "Mary had a little lamb"
  close Mary
  output Mary
```

You can use the action `reopen` to reopen a closed stream with its original content. However, if you use `open` to open the stream again, the existing content is lost:

```
  open Mary1 as buffer
  open Mary2 as buffer

  put Mary1 "Mary had a little lamb"
  put Mary2 "Mary had a little lamb"

  close Mary1
  close Mary2

  reopen Mary1
  open Mary2

  put Mary1 "Its fleece was white as snow"
  put Mary2 "Its fleece was white as snow"
```

This code will leave the stream Mary1 containing both lines, but Mary2 will contain only "Its fleece was white as snow".

Can you mix the two methods? Yes, but remember that "set" is a shorthand for the sequence "open...put...close" which means that a stream is always closed after a set. This means that you cannot write to it without reopening it. Also remember that a stream is always opened, not reopened, in a set, so the previous content is always lost. In practice, it is not a good idea to mix the two methods. They are really appropriate for different scales of operation. Pick the one that is appropriate to what you have to do and stick to it.

Copyright©1988-2004 Stilo International plc

Arrays

Most programming languages allow programmers to store and manipulate values in arrays, associative arrays, queues, and stacks. Instead, OmniMark provides a data container called a "shelf" which can be used to accomplish all of the tasks normally carried out by these various structures in other programming languages. Like arrays, shelves can be indexed by numeric values that reflect the position of the elements they contain or, like associative arrays, these elements can be given names (keys) and then indexed by those keys.

A shelf is a data structure that is used to store one or more values of a certain type. Stream shelves can be used to store one or more string values, counter shelves to store one or more numeric values, and switch shelves to store one or more Boolean values.

A global stream shelf declaration that creates a shelf of variable size named `quotations` would look like this:

```
  global stream quotations variable
```

A local counter shelf declaration that creates a counter shelf named "count1" that can contain three (and only three) numeric values would look like this:

```
  local counter count1 size 3
```

If you want to create a shelf with initial values that are different from the defaults, you can do this by adding an `initial` keyword to the declaration, followed by the values you want on the shelf being enclosed in curly braces. For example:

```
  global counter count2 size 4 initial {1, 2, 3, 4}
```

This declaration creates a global counter shelf named "count2" that can hold four values with initial values of "1", "2", "3", and "4". You could also create a variable-sized shelf that contains a number of initial values, as follows:

```
  global counter count3 variable initial {1, 2, 3, 4}
```

The only difference between these two shelves (other than their names) is that while "count2" is a fixed-size shelf holding four values, "count3" begins with four values and can be expanded or contracted to hold as many as required. If you're not sure how many values you will need to store on a shelf, it's best to declare it with a `variable` size.

Additionally, shelves of a particular size can be created without having to assign initial values to the shelf items. This is accomplished by using the `initial-size` keyword:

```
  global counter count4 variable initial-size 4
```

This shelf declaration creates a counter named "count4" that starts with space for four items and can be expanded or contracted as required.

To store the string "Now is the winter of our discontent" in the stream shelf "quotations", you would use the following action:

```
  set new quotation to "Now is the winter of our discontent"
```

This begs the question "where on the shelf was this value stored?" Unless you explicitly specify which item on a shelf you want a value stored in, a value will be stored in the current item. A shelf is basically an ordered list of items ranging from 1 to n. The default behavior of a shelf is that all new items are added after n. If you use `set` to store a different value on the shelf without specifying a different item, it will simply replace the n value on the shelf.

To change this default behavior, you can use either of two shelf indexing methods. The first index is based upon the position number of a value on a shelf. For example, the following code sets a value in the third position of the "quotation" shelf:

```
  set quotation item 3 to "Words, words, words."
```

The second index is based upon names or "keys" that are assigned to each value on a shelf. To set the key of the current item on a shelf, you would use the following code:

```
  set key of quotation to "Richard iii"
```

To set the key of a particular item on a shelf, you would use the same code, but adding a position index:

```
  set key of quotation item 3 to "Hamlet"
```

Using the key index of a shelf is very like using the position index, except instead of using the `item` keyword, you use the `key` keyword:

```
  set quotation key "Hamlet" to "To be or not to be?"
```

It is possible to set a key on a shelf item when it is created. This is accomplished by setting the key in the same action in which the new item is created. For example, to create a new item on the "quotes" shelf that has a value of "Alas, poor Yorick." with the key "Hamlet", you would use the action:

```
  set new quotes key "Hamlet" to "Alas, poor Yorick."
```

Up to this point, every time we have created a new item on a shelf, it has been added at the lastmost position of the shelf. If you want to create a new item somewhere else on a shelf, this can be accomplished by using the `before` or `after` keywords in the same action used to create the new item. For example, if you want to create a new item that will exist immediately before the second item on a shelf, you would use the following action:

```
  set new quotes before item 2 to "A horse!"
```

This would create a new item containing the value "A horse!" between the first and second items on the "quotes" shelf. Since the item numbers are based on shelf position, this new item would become item 2, and the item that was number 2 would become number 3. If the values had assigned keys, of course, these keys would not change.

If you wanted to create a new item on a shelf just after an item that had the key "Macbeth", you would use the action:

```
  set new quotes after key "Macbeth" to "A horse!"
```

To illustrate all of this, the following program creates a global stream shelf, and sets the first item on that shelf to a value. Following that, the program gives that first item a key. Then three other items are created: one at the "default" end of the shelf, another before the second item on the shelf, and the third after a value with a set key.

```
  process
     local stream quotes variable
     set new quotes key "Hamlet" to "To be or not to be?"
     set new quotes key "Macbeth" to "Is this a dagger?"
     set new quotes key "Richard iii" before item 2 to "A horse!"
     set new quotes key "Romeo" after key "Richard iii" to "But soft,
what light through yonder window breaks?"

     repeat over quotes
        output key of quotes
            || " - "
            || quotes || "%n"
     again
```

This program will have the following output:

```
  Hamlet - To be or not to be?
  Richard III - A horse!
  Romeo – But soft, what light through yonder window breaks?
  Macbeth - Is this a dagger?
```

Stacks and queues

OmniMark shelves, in addition to having all of the characteristics of arrays and associative arrays, also have the properties of stacks and queues.

A stack is a type of data container which operates under the basic "FILO" (First In Last Out) principle. When you add two items to a stack, for example, you have to remove the second item before you can access the first. The default behavior of the currently selected item on a shelf makes it easy to create stack-like shelves in OmniMark. Quite simply, if you do not explicitly state that actions should be performed on a different shelf item, actions will be carried out on the default currently selected item which is the lastmost item on a shelf.

For example, the following program illustrates how OmniMark shelves act like stacks:

```
process
    local stream name-stack variable initial {"Tom", "Dick"}

    output "The stack now contains%n"
    repeat over name-stack
        output name-stack || "%n"
    again
    output "%n"

    set new name-stack to "Harry"
    output "Pushed "
        || name-stack
        || " on to the stack.%n%n"

    output "The stack now contains%n"
    repeat over name-stack
        output name-stack || "%n"
    again
    output "%n"

    ; Pop all of the items off a stack

    repeat
        exit when number of name-stack = 0

        output "Popping "
            || name-stack
            || " from the stack.%n%n"
        remove name-stack

        output "The stack now contains%n"
        repeat over name-stack
            output name-stack || "%n"
        again
        output "%n"
    again
```

You will notice that this program simply adds and removes items from the "name-stack" shelf at the default item.

A queue is like a stack except it operates under the "FIFO" (First In First Out) principle. If you add two items to a queue, you have to remove the first item before you can access the second. To create a queue-like shelf in OmniMark, you need only specify that all actions are performed on the first item on the shelf (as opposed to the default lastmost item). Any new items should still be added to the shelf at the default lastmost position.

The following program illustrates an OmniMark shelf acting like a queue:

```
process
    local stream name-queue variable initial {"Tom", "Dick"}

    output "The queue now contains%n"
    repeat over name-queue
        output name-queue || "%n"
    again
    output "%n"

    set new name-queue to "Harry"
    output "Pushed "
        || name-queue
        || " on to the queue.%n%n"

    output "The queue now contains%n"
    repeat over name-queue
        output name-queue || "%n"
    again
    output "%n"

    ; Pop all of the items off a stack

    repeat
        exit when number of name-queue = 0

        output "Popping "
            || name-queue item 1
            || " from the queue.%n%n"
        remove name-queue item 1

        output "The queue now contains%n"
        repeat over name-queue
            output name-queue || "%n"
        again
        output "%n"
    again
```

The only real difference between the programs is that the first program removed items from the shelf which were at the default lastmost position, while the second removed items from the shelf which existed at position "1" on the shelf.

Referents

Referents are variables that can be output before their final values have been assigned. With referents you are able to stick "placeholder" variables in your output and then later assign or change their values. These "placeholder" variables are particularly useful in creating hypertext links and cross-references, but they can be used for numerous other tasks. The following program illustrates the "placeholder" quality of referents:

```
process
     output "Goodbye, " || referent "referent1" || "world!"
     set referent "referent1" to "cruel "
```

The result of this program is to output the line, "Goodbye, cruel world!"

Here is a more complex example:

```
process
   local stream foo
   set foo to "Mary%n"
   set referent "bar" to "Mary%n"
   output foo
   output referent "bar"
   set foo to "lamb%n"
   set referent "bar" to "lamb%n"
```

The output of this program is:

```
Mary
lamb
```

Where the output value of stream "foo" didn't change values after being output, the output value of referent "bar" did. The final value of both of the variables did change to "lamb", but only the output of the referent reflected this change.

Notice that while the stream "foo" had to be declared before it was used, the referent "bar" did not. All you need to do to create and use a referent is give it a name and set it to a value. For example, the following code creates a referent named "ref1" and sets it to an initial value of "mary joe":

```
set referent ref1 to "mary joe"
```

Another simple example of the use of referents is in outputting page numbers that include "of n" values, for example, "page 1 of 8". Until a document has been completely processed, there is no way to know for certain how many pages there are going to be. With referents, however, you can simply stick a placeholder where the page numbers will be in the output and, after the document has been completely processed and the number of pages determined, the final values can be plugged into the referents.

The following is a short program that will output a referent when it finds one or more numbers in the input file:

```
global counter num initial {0}

find digit+
   increment num
   output referent "%d(num)ref"

process
   local counter num2
   submit file "test1.txt"
   repeat
      exit when num2 > num
      set referent "%d(num2)ref" to "Play %d(num2) of %d(num)%t"
      increment num2
   again
```

An appropriate plain-text input file for this program would be:

```
1 Hamlet
2 Richard III
3 Macbeth
4 Romeo and Juliet
5 King Lear
```

If this input file were processed by the program shown above, the output would be:

```
Play 1 of 5  Hamlet
Play 2 of 5  Richard III
Play 3 of 5  Macbeth
Play 4 of 5  Romeo and Juliet
Play 5 of 5  King Lear
```

So, what has happened to this output is that OmniMark matched a digit, output a referent as a placeholder, and let any following text (the title of the play) fall through to the output. With each digit encountered, the process is repeated. When the process-end rule fired, the final values of the referents were determined and resolved.

Referents cannot be output to the `#markup-parser` stream. To do so will result in an error. You also cannot change the properties of #markup-parser stream to allow referent processing.

If you need to do referent processing on the text that is to be parsed, open a local referents scope in your process rule with `using nested-referents`. Perform the processing in the scope of "using nested-referents". When processing is complete and the nested referent scope is terminated, the result can be passed as input to the markup processor (the #markup-parser stream).

Copyright©1988-2004 Stilo International plc

Conditional constructs

When you want a program to do one of several possible things in different situations, use a conditional construct. OmniMark provides three different forms of conditional construct, each based upon the basic "do...done" block.

It is important to note that almost anything in OmniMark can be made conditional simply by adding a `when` keyword followed by a test. For example, any rule can have conditions added to it:

```
find "cat" when count1 = 4
   output "I found a cat%n"
```

This rule would only output "I found a cat" if "cat" is found in the input data and the value of count1 is equal to 4.

The simplest of the conditional constructs is the "do when...done" block. This allows you to have an OmniMark program perform various actions based on the results of one or more tests.

```
do when count1 = 4
   output "Yes, the value of count1 is four%n"
done
```

If you want the program to do one thing when a certain condition is true and another if it is false, you can add an `else` option.

```
do when words matches uc
   output "%lg(words)%n"
else
   output words || "%n"
done
```

You can have a "do when" block perform a set of actions if a variable is of more than one value, by adding those conditions to the header using the `or` keyword. For example:

```
do when count1 = 1 or count1 = 5
   output "count1 is one or five%n"
else
   output "the value of count1 is not one or five%n"
done
```

"Do when" blocks can be much more complex than this, however, since "else when" phrases are also allowed.

```
do when count1 = 4
   output "Yes, the value of count1 is four%n"
else when count1 = 5
   output "The value of count1 is five%n"
else when count1 = 6
```

```
      output "The value of count1 is six%n"
   else
      output "The value of count1 is not 4, 5, or 6%n"
   done
```

Another form of conditional construct is the "do select...done" construct:

```
   do select count1
      case 1 to 5
         output "count1 is within the first range%n"
      case 6 to 10
         output "count1 is within the second range%n"
   done
```

The program won't do anything if the value of count1 is less than 1 or greater than 10, however, because there is no alternative that will be executed in these situations. This is quite easily rectified, by adding an "else" phrase to the construct:

```
   do select count1
      case 1 to 5
         output "count1 is within the first range%n"
      case 6 to 10
         output "count1 is within the second range%n"
      else
         output "count1 is out of range%n"
   done
```

Note that while "else" phrases can be used within a "do select" construct, "else when" phrases cannot.

If you want the program to do something when a variable is equal to a particular value, you have to specify that within another "case" phrase. For example:

```
   do select count1
      case 1 to 4
         output "count1 is in the first range%n"
      case 5
         output "count1 is equal to 5%n"
      case 6 to 10
         output "count1 is in the second range%n"
      else
         output "count1 is out of range%n"
   done
```

The final form of conditional constructs is a "do scan". "Do scan" constructs are used to examine a piece of input data for certain patterns. If one of the patterns is discovered in the input data, a set of specified actions is performed. For example, the following program retrieves the name of the current day and scans it. Depending on which pattern is found, the program will output one of several possible phrases.

```
   global stream day

   process
      set day to date "=W"
```

```
do scan day
   match "Monday"
      output "I don't like Mondays.%n"
   match "Friday"
      output "I love Fridays!!!%n"
   else
      output "At least it's not Monday.%n"
done
```

"Do scan" constructs can be used to scan input data in the form of files, streams, or the values of stream variables (as above).

Looping constructs

To have an OmniMark program perform an action or set of actions repeatedly, you will need to create a looping construct of some sort. OmniMark provides three types of looping constructs, `repeat`, `repeat over`, and `repeat scan`.

The simplest is a `repeat...again`. This form of loop will simply repeat the execution of the actions it contains, until an explicit `exit` action is encountered in the loop.

```
  process
     local counter count1
     repeat
        output "count1 is %d(count1)%n"
        increment count1
        exit when count1 = 4
     again
```

This `repeat...again` will execute the `output` action until the counter "count1" equals 4 at which point the `exit` action will execute and the loop will terminate, resulting in the following output:

```
  count1 is 1
  count1 is 2
  count1 is 3
```

The second type of looping construct is a `repeat over...again`. This type of loop is used to iterate over a shelf and perform a set of actions on each item that exists on that shelf. For example, the following program will output the values of each item contained on the stream shelf "some-names":

```
  global stream some-names variable initial {"Bob", "Doug", "Andy",
"Greg"}

  process
     repeat over some-names
        output some-names || "%n"
     again
```

`repeat over` loops can be used to iterate over any type of shelf, and the loop is terminated after the last item on the shelf has been processed.

Arithmetic and comparisons

If you've got two or more values or variables and you want to do something with them or to them, you need an operator.

The most common sorts of operators are arithmetic operators, such as those which perform addition or multiplication. For example:

```
process
   local counter x
   set x to 1 + 1
   output "%d(x)%n"
```

The arithmetic operators available in OmniMark are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `modulo` (the remainder you get when you divide the number by the base value).

OmniMark also provides a full set of operators that are used to compare two or more numeric values. For example:

```
process
   do when x = y
      output "equal%n"
   else when x > y
      output "greater%n"
   else when x < y
      output "lesser%n"
   done
```

The other available numeric comparison operators are `!=` (not equal), >= (greater than or equal to), and <= (less than or equal to).

Other common operators are `&` and `|` (the `and` and `or` Boolean operators). These are usually used to create more complex conditions and tests in OmniMark. For example:

```
process
   do when x = y & z > 4
      output "first test is true%n"
   else when y = z | x = 4
      output "second test is true%n"
   else
      output "neither test is true%n"
   done
```

Copyright©1988-2004 Stilo International plc

Pattern matching

OmniMark allows you to search for particular strings in input data using find rules. For example, the following find rule will fire if the string "Hamlet:" is encountered in the input:

```
find "Hamlet:"
   output "<b>Hamlet</b>: "
```

Using this method, however, you would have to write a separate find rule for each character name you wanted to enclose in HTML bold tags. For example:

```
find "Hamlet:"
   output "<b>Hamlet</b>: "
find "Horatio:"
   output "<b>Horatio</b>: "
find "Bernardo:"
   output "<b>Bernardo</b>: "
```

As you can imagine, this is a pretty inefficient way to program.

This is where OmniMark "patterns" come in. OmniMark has rich, built-in, pattern-matching capabilities which allow you to match strings by way of a more abstract "model" of a string rather than matching a specific string. For example:

```
find letter+ ":"
```

This `find` rule will match any string that contains any number of letters followed immediately by a colon.

Unfortunately, the pattern described in this find rule isn't specific enough to flawlessly match only character names. It will match any string of letters that is followed by a colon that appears anywhere in the text, meaning that words in the middle of sentences will be matched.

Words that appear in the middle of sentences rarely begin with an uppercased letter, while names usually do. This allows us to add further detail to our find rule:

```
find uc letter+ ":"
```

This find rule matches any string that begins with an uppercase letter (`uc`) followed by at least one other letter (`letter+`) and a colon (":").

If we were actually trying to mark up an ASCII copy of "Hamlet", however, our find rule would only match character names that contain a single word, such as "Hamlet", "Ophelia", or "Horatio". Only the second part of two-part names would be matched, so the names of "Queen Gertrude", "Lord Polonius", and so forth, would be incorrectly marked up.

In order to match these more complex names as well as the single-word names, we'll have to further refine our find rule:

```
find uc letter+ (white-space+ uc letter+)? ":"
```

In this version of the `find` rule, the pattern can match a second word prior to the colon. The pattern `(white-space+ uc letter+)?` can match one or more white-space characters followed by an uppercase letter and one or more letters. All of this allows the find rule to match character names that consist of one or two words.

If you wanted to match a series of three numbers, you could use the following pattern:

```
find digit {3}
```

If you wanted to match either a four-digit or a five-digit number, you could use the following pattern:

```
find digit {4 to 5}
```

To match a date that occurs in the" yy/mm/dd" format, the following pattern could be used:

```
find digit {2} "/" digit {2} "/" digit {2}
```

A Canadian postal code could be matched with the following pattern:

```
find letter digit letter " " digit letter digit
```

The `letter` and `uc` keywords that are used to create the patterns shown above are called "character classes". OmniMark provides a variety of these built-in character classes:

- `letter` -- matches a single letter character, uppercase or lowercase
- `uc` -- matches a single uppercased letter
- `lc` -- matches a single lowercased letter
- `digit` -- matches a single digit (0-9)
- `space` -- matches a single space character
- `blank` -- matches a single space or tab character
- `white-space` -- matches a single space, tab, or newline character
- `any-text` -- matches any single character except for a newline
- `any` -- matches any single character

Any pattern can be modified through the use of occurrence operators:

- `+` (one or more)
- `*` (zero or more)
- `?` (zero or one)

So, as shown in the find rules above, for example, `letter+` matches one or more letters, `letter*` matches zero or more letters, and `uc?` matches zero or one uppercase letter.

It is also possible for you to define your own customized character classes. For example:

```
find ["+-*/"]
   output "found an arithmetic operator%n"
```

This find rule would fire if any one of the four arithmetic operators was encountered in the input data.

Compound character classes can be created using the `except` or `or` keywords:

```
find [any except "}"]
```

The find rule above would match any character except for a right brace.

This find rule would match any one of the arithmetic operators or a single digit:

```
find ["+-*/" or digit]
```

This one would match any of the arithmetic operators or any digit except zero ("0"):

```
find ["+-*/" or digit except "0"]
```

Pattern variables

When using patterns to match sections of input data, you must first capture the data in pattern variables for later use. Pattern variables are assigned using the => symbol, and referenced later. For example, in the first `find` rule in the following program the matched input data is assigned to the "found-text" pattern variable.

```
process
   submit "Mary had a little [white] lamb"

find ("[" letter+ "]") => found-text
   output found-text

find any
```

This program outputs "[white]".

What if you want to output only the word in the square brackets, but not the brackets themselves? Try this:

```
process
   submit "Mary had a little [white] lamb"

find "[" letter+ => found-text "]"
   output found-text

find any
```

This program outputs "white". Here, the pattern variable is attached only to the part of the pattern immediately preceding the pattern variable assignment. In fact, this is the default behavior of pattern variables. That's why, to make the previous example work correctly, we had to surround the three elements of the pattern with parentheses to ensure that the text matched by the whole pattern was captured.

You can have more than one pattern variable in a pattern. You can even nest them. For example:

```
process
   submit "Mary had a little [white] lamb"

find  ("[" => first-bracket
   letter+ => found-word
   "]" => second-bracket) => found-text

   output first-bracket
   output found-word
   output second-bracket
   output found-text

find any
```

The output of this program would be "[white][white]". The first "[white]" is the result of the first three output actions, and the second the result of the fourth output action.

Copyright©1988-2004 Stilo International plc

Organizing your program

OmniMark programs have a definite style which reflects the kind of programming you do with OmniMark. OmniMark is used for manipulating and transforming data, either as text or as markup and, in doing so, it responds to events which occur in the data. Since there is no way to predict in advance the order or relationships of data events, there is no way to predict the order of execution of an OmniMark program.

Many programming languages encourage nested code, with functions calling functions calling functions. This helps modularize functionality in a regular programming language. It also makes the execution path rigid and makes it difficult to react to complex sequences of events. OmniMark code is very flat. While you can define and use functions, they are used only within OmniMark's principal execution unit, the rule, and cannot contain rules themselves. All OmniMark rules exist at the base level of the program. In OmniMark you tend to find not nested code, but nested execution.

In processing complex markup, with many nested elements, rules are invoked at each level as appropriate. If you are seven layers of markup deep, seven rules are in mid-execution. This means that you do not have to maintain complex state tables or parse trees. The current execution state of the OmniMark program itself maintains the current parse state for you and makes it easily addressable.

Since you cannot tell in advance the order in which the execution of rules may be nested, nesting the rules themselves would make no sense. Hence the simplicity and flatness of a typical OmniMark program.

Nevertheless, you can and should encapsulate common functionality in your OmniMark programs. OmniMark provides several facilities to do this including functions, groups, macros, and include files.

If you are writing a program that does batch translation, you can save a lot of time and code for initialization and flow control by using one of OmniMark's aided translation types.

Copyright©1988-2004 Stilo International plc

Functions

Like most languages, OmniMark supports functions. Unlike many languages, an OmniMark program is not simply a hierarchy of functions. Rules are the principal structural element of OmniMark programs. Functions are supplementary structures. Functions cannot contain rules (though they can invoke them through `submit`, `do xml-parse`, or `do sgml-parse`). You can use functions to encapsulate code you use commonly within different rule bodies. You can also use functions as pattern matching functions or within patterns to dynamically define a pattern to be matched.

Functions isolate sections of code, but don't isolate you from the current environment, in particular the current output scope. "Output" in a function goes to the current output scope. If a function has a return value, that value goes to the calling action. If a function changes the destinations of the current output scope (with `output-to`), this carries over to the calling environment.

A function that returns a value is defined as follows:

```
define counter function add
   (value counter x,
    value counter y)
   as
   return x + y
```

The return type of the function is declared following the `define` keyword. It may be any OmniMark variable type or any OMX component type. The value is returned using the `return` keyword. `return` exits the function.

Here is how the "add" function can be called:

```
process
   output "d" format add(2,3)
```

Functions can generate output. The following function outputs the value it generates rather than returning it to the calling action. Note that it has no return type in the definition and no `return` is required:

```
define function output-sum
   (value counter x,
    value counter y
   )as
   output "d" format (x + y)
```

This function is called as if it were a regular OmniMark action:

```
process
   output-sum(2, 3)
```

You can also write functions that both return a value and do output:

```
define counter function add
```

```
     (value counter x,
      value counter y
     )as
     output "I will add %d(x) and %d(y)%n"
     return x + y

  process
     local counter z
     set z to add(2,3)
     output "%d(z)%n"
```

While it is certainly possible to program like this, we recommend that you avoid writing functions that both do output and return a value. Not only do they make it hard to follow your code, but they can have unexpected results. In particular, if the return value is directed to current output, you may not get the function's return values and output in the order you expected.

Finally, you can write functions that neither return a value nor create output:

```
  define function clear-flags
      (modifiable switch the-flags
      ) as
      repeat over the-flags
        set the-flags to false
     again
```

This function clears all the switches on a switch shelf that is passed to it as a modifiable argument.

Functions and the rules that call them

Whatever you can do in the rule that calls a function you can do in the function itself. For example, a function can use the `%c` operator, or the "%v" or "%q" format items when called from a rule that supports them. The most useful application of this feature is the ability to write a function that does generic processing on a class of markup element types.

Side effects

The principal job of a function that returns a value is to calculate and return that value. However, a function may have side effects on the global state of the program. While writing functions with side effects is appropriate in some situations, you should exercise caution when using this technique as it can lead to programs that are difficult to debug and hard to read and maintain.

Function side effects can be particularly problematic with functions used in patterns and in the guards of rules. To allow for optimization of pattern matching routines, OmniMark does not define whether a pattern or the guard on a pattern is executed first (a pattern is itself a kind of guard on a statement, so this is sensible). You should never write a program that depends on the order in which a pattern and a guard on that pattern are executed.

In the case of patterns that fail, OmniMark does not guarantee that all parts of the pattern will be tried, or that the same parts will be tried in all circumstances. This allows OmniMark to optimize pattern matching. You should never write a program that depends on the side effects of a function called in a pattern that fails.

Copyright©1988-2004 Stilo International plc

Catch and Throw

You can use catch and throw to manage the execution flow in your OmniMark programs. Catch and throw is a powerful addition to the flow handling features of OmniMark, allowing you to make major redirections of program flow in a safe and structured way.

It is probably easiest to think of catch and throw as an exception handling mechanism. What is an exception? Essentially anything that is an exception to the normal flow of your processing. Some exceptions come whether you want them or not, in the form of run-time program errors or failure to communicate with the world outside your program. Others are simply an expression of the way you choose to solve your programming problem.

It is often the case that a simple piece of code can handle 90% of all cases. The other 10% are exceptions that require significantly different processing. Dividing a problem up into "normal" cases and "exceptions" is a convenient way to simplify and organize your code. This does not mean that the exceptions are errors or even unexpected. Often the exceptional cases are what you are most interested in. Programming with exceptions is simply a technique for designing an algorithm to solve a particular problem.

Catching program errors

The simplest use of catch and throw is to allow your program to recover when something goes wrong. Consider the following code, which contains the server loop of a simple server program.

```
include "omioprot.xin"
include "omtcp.xin"

process
   local TCPService my-service
   set my-service to TCPServiceOpen at 5432
   repeat
      local TCPConnection my-connection
      local stream  my-response
      set my-connection to TCPServiceAcceptConnection my-service
      open my-response as TCPConnectionGetOutput my-connection
      using output as my-response
      do
         submit TCPConnectionGetLine my-connection
      done
   again
```

Any error in the program or with the TCP connection will cause this program to terminate, since there is nothing to handle the error. Server programs should be written to stay running if at all possible, so we need to do something to allow the program to recover:

```
include "omioprot.xin"
include "omtcp.xin"

process
```

```
     local TCPService my-service
     set my-service to TCPServiceOpen at 5432
     repeat
        local TCPConnection my-connection
        local stream  my-response
        set my-connection to TCPServiceAcceptConnection my-service
        open my-response as TCPConnectionGetOutput my-connection
        using output as my-response
        do
           submit TCPConnectionGetLine my-connection
        done
        catch #program-error
     again
```

We have added only a single line, but this version of the program is much more robust. If any error occurs while in the repeat loop of any of the find rules invoked by the submit, execution will transfer to the line "catch #program-error". Along the way, OmniMark will clean up after itself. Local scopes will be terminated, and resources released.

No attempt is made to salvage the work that was in progress when the error happened. That work, and all the resources associated with it, are thrown away. But the server will stay up and running and ready to receive the next request.

Of course, it would be nice to know that the error occurred and why it occurred. #program-error makes information available so that we can act on the error, or at least report it. To ensure that errors get logged, we can rewrite the program like this:

```
  include "builtins.xin"
  include "omioprot.xin"
  include "omtcp.xin"

  process
     local TCPService my-service
     set my-service to TCPServiceOpen at 5432
     repeat
        local TCPConnection my-connection
        local stream  my-response
        set my-connection to TCPServiceAcceptConnection my-service
        open my-response as TCPConnectionGetOutput my-connection
        using output as my-response
        do
           submit TCPConnectionGetLine my-connection
        done
        catch #program-error code c message m location l
           log-message ( "Error "
                      || "d" format c
                      || " "
                      || m
                      || " at "
                      || l
                      || " time "
                      || date "=Y/=M/=D =h:=m:=s"
                       )
     again
```

Note that the catch line guards everything that follows from being executed. The only way into the code of a catch block is to be caught by that catch.

So far, we have seen nothing of the "throw" part of throw and catch. This is because OmniMark itself throws to #program-error. OmniMark can also throw to #external-exception if it has a problem communicating with the external world (such as being unable to open a file or communicate successfully with an OMX component). We don't bother to catch #external-exception in the code above, because the failure of a program to catch an external exception is itself a program error, which causes a throw to #program-error. In other circumstances we might want to use #external-exception explicitly. Suppose one of the find rules in our server program tried and failed to open a file:

```
find "open file " letter+ => foo
   output file foo
   catch #external-exception
   output "Unable to open file " || foo || "."
```

Suppose the attempt to open the named file fails. This causes an external exception at the first output action. We catch the exception and provide alternate output. The program then continues as if nothing had happened.

Now our server is more robust still, since an error opening a file will not cause processing of the current request to be aborted. Instead, the client will receive the error message we output along with any other information the request generates.

Exception handling

When it comes to throwing things, OmniMark does not get to have all the fun. We can create our own exceptions, throw our own throws, and define our own catches. Let's write our own exception to let us shut down the server by remote control (to simplify things, we've left out some of our earlier enhancements):

```
include "builtins.xin"
include "omioprot.xin"
include "omtcp.xin"
declare catch nap-time

process
   local TCPService my-service
   set my-service to TCPServiceOpen at 5432
   repeat
      local TCPConnection my-connection
      local stream  my-response
      set my-connection to TCPServiceAcceptConnection my-service
      open my-response as TCPConnectionGetOutput my-connection
      using output as my-response
      do
         submit TCPConnectionGetLine my-connection
      done
      catch #program-error
   again
   catch nap-time
```

```
find "sleep"
   throw nap-time
```

Here we have a classic case of an exception. Every request to our server is a request for information. Every request except one. The "sleep" request is an instruction to the server to shut itself down. This is the exceptional case and we handle it with an exception.

In the exceptional case that we are shutting down the server, we need to jump out of the connection loop. We do this with a catch outside the loop. The catch is called "nap-time". Catches are named so that we can have more than one and have each one catch something different. Like all names introduced into an OmniMark program, the name of a catch must be declared, which we do in the first line of the program. We place the catch outside the request handling loop. Because OmniMark cleans up after itself while performing a throw, all the resources belonging to the local scope inside the loop are properly closed down. Then, since we are outside the loop and at the end of the process rule, the program simply ends, shutting down the server.

The throw itself is very simple. Having detected the exceptional case (the "sleep" request) we simply throw to the appropriate catch by name. OmniMark handles everything else.

Throwing additional information

When OmniMark throws to #program-error or #external-exception, it adds additional information in the form of three parameters: code, message, and location. It is only fair that we be allowed to pass additional information with our throws as well. We can do this by adding parameters to our catch declaration, following the form of a function definition. Let's add the capability to log the reason for putting a server to sleep:

```
include "builtins.xin"
include "omioprot.xin"
include "omtcp.xin"
declare catch nap-time because value stream reason

process
   local TCPService my-service
   set my-service to TCPServiceOpen at 5432
   repeat
      local TCPConnection my-connection
      local stream  my-response
      set my-connection to TCPServiceAcceptConnection my-service
      open my-response as TCPConnectionGetOutput my-connection
      using output as my-response
      do
         submit TCPConnectionGetLine my-connection
      done
      catch #program-error
   again
   catch nap-time because r
      log-message ( "Shut down because "
                 || r
                 || " Time: "
                 || date "=Y/=M/=D =h:=m:=s"
                  )
```

```
  find "sleep" white-space* any* => the-reason
     throw nap-time because the-reason
```

Here the find rule captures the rest of the "sleep" message and uses it as a parameter to the throw. The catch receives the data and uses it to create the appropriate log message.

Cleaning up after yourself

We said that OmniMark cleans up after itself, and it does. It cleans up everything it knows about. But this may still leave you with cleanup of your own to do. Or there may simply be things that always have to be done, even if an exception occurs. For this, OmniMark provides the "always" keyword. Let's suppose that our server does its own connection logging, while still using OmniMark's logging facility for errors. We want the connection log file closed between requests to make it easy to cycle the log files, so we open and close the log file for each connection.

```
  declare catch nap-time because value stream reason
  include "builtins.xin"
  include "omioprot.xin"
  include "omtcp.xin"
  global stream log-file-name ;should be initialized from the command
line

  process
     local TCPService my-service
     local stream my-log
     set my-service to TCPServiceOpen at 5432
     repeat
        local TCPConnection my-connection
        local stream  my-response
        reopen my-log as file log-file-name
        set my-connection to TCPServiceAcceptConnection my-service
        put my-log TCPConnectionGetPeerIP my-connection || date
"=Y/=M/=D =h:=m:=s"
        open my-response as TCPConnectionGetOutput my-connection
        using output as my-response
        do
           submit TCPConnectionGetLine my-connection
        done
        close my-log
        catch #program-error code c message m location l
           log-message ( "Error "
                      || "d" format c
                      || " "
                      || m
                      || " at "
                      || l
                      || " time "
                      || date "=Y/=M/=D =h:=m:=s"
                       )

     again
     catch nap-time because reason
        log-message ( "Shut down because "
                   || reason
```

```
                     || " Time: "
                     || date "=Y/=M/=D =h:=m:=s"
                      )
```

In this code, a problem processing the request would cause a throw to #program-error. This would mean that the line "close my-log" was never executed and the log file would remain open. (This is something OmniMark can't clean up itself, since the stream my-log belongs to a wider scope which is not being closed.) We want the line "close my-log" to be executed always, whether there is an error or not. To ensure this, we use "always":

```
  declare catch nap-time because value stream reason
  include "builtins.xin"
  include "omioprot.xin"
  include "omtcp.xin"
  global stream log-file-name ;should be initialized from the command
line

  process
     local TCPService my-service
     local stream my-log
     set my-service to TCPServiceOpen at 5432
     repeat
        local TCPConnection my-connection
        local stream  my-response
        reopen my-log as file log-file-name
        set my-connection to TCPServiceAcceptConnection my-service
        put my-log TCPConnectionGetPeerIP my-connection || date
"=Y/=M/=D =h:=m:=s"
        open my-response as TCPConnectionGetOutput my-connection
        using output as my-response
        do
           submit TCPConnectionGetLine my-connection
        done
        catch #program-error code c message m location l
           log-message ( "Error "
                       || "d" format c
                       || " "
                       || m
                       || " at "
                       || l
                       || " time "
                       || date "=Y/=M/=D =h:=m:=s"
                        )
        always
           close my-log

     again
     catch nap-time because reason
        log-message ( "Shut down because "
                    || reason
                    || " Time: "
                    || date "=Y/=M/=D =h:=m:=s"
                     )

  find "sleep" white-space* any* => the-reason
     throw nap-time because the-reason
```

When a throw happens, OmniMark closes scopes one by one until it finds a scope that contains a catch for that throw. As it does so, it executes any code in an always block in each of those scopes, including the scope that contains the catch. So in this example, the "close" line will be executed before the catch block is executed, no matter where or why an error occurs.

Programming with exceptions is a powerful technique that can make your programs both more reliable and easier to read and write. Here is our full server program with all our catch and throw functionality, plus logging of our external exception (but minus the find rules that do the rest of the work):

```
  declare catch nap-time because value stream reason
  include "builtins.xin"
  include "omioprot.xin"
  include "omtcp.xin"
  global stream log-file-name ;should be initialized from the command
line

  process
     local TCPService my-service
     local stream my-log
     set my-service to TCPServiceOpen at 5432
     repeat
        local TCPConnection my-connection
        local stream  my-response
        reopen my-log as file log-file-name
        set my-connection to TCPServiceAcceptConnection my-service
        put my-log TCPConnectionGetPeerIP my-connection || date
"=Y/=M/=D =h:=m:=s"
        open my-response as TCPConnectionGetOutput my-connection
        using output as my-response
        do
           submit TCPConnectionGetLine my-connection
        done
        catch #program-error code c message m location l
           log-message ( "Error "
                      || "d" format c
                      || " "
                      || m
                      || " at "
                      || l
                      || " time "
                      || date "=Y/=M/=D =h:=m:=s"
                       )
        always
           close my-log

     again
     catch nap-time because reason
        log-message ( "Shut down because "
                   || reason
                   || " Time: "
                   || date "=Y/=M/=D =h:=m:=s"
                    )

  find "sleep" white-space* any* => the-reason
     throw nap-time because the-reason
```

```
find "open file " letter+ => foo
   output file foo
   catch #external-exception identity i message m location l
   output "Unable to open file " || foo || "."
   log-message ( "Error "
              || i
              || " "
              || m
              || " at "
              || l
              || " time "
              || date "=Y/=M/=D =h:=m:=s"
               )
```

Rule groups

By default, all OmniMark rules are active all the time. You can change this by bundling your rules into groups:

```
group mary
   find "lamb"
      ...
   find "school"
      ...

group tom
   find "piper"
      ...
   find "pig"
      ...

group #implied
process-start
   using group mary
   do
      submit "Mary had a little lamb"
   done
   using group tom
   do
      submit "Tom, Tom, the piper's son"
   done
```

In this program, only rules in the group "mary" are used to process "Mary had a little lamb". Only rules in the group "tom" are used to process "Tom, Tom, the piper's son".

Why the `group #implied` before the process-start rule? The process-start rule is a rule like any other, so it is affected by groups like any other rule. `group #implied` stands for the default group. (In a program with no groups, all rules are in the default group.) Only the default group is active when a program starts. All other groups are inactive. So, you have to have at least one rule in the default group in order to activate any of the other groups. If we didn't place the process-start rule into the default group, no rules would ever be active in this program.

Any rule that occurs before the first group statement in your program automatically belongs to the default group, but, if you use groups, it is usually a good idea to place your global rules explicitly into group #implied. (Consider what would happen if you included a file that contained group statements at the top of your main program file and didn't explicitly assign your global rules to group #implied.)

All rules in the default group are global. You cannot disable the default group, so rules in the default group are always active. For this reason, you may want to keep the number of rules in the default group to a minimum (but remember, you must have at least one).

Can you have more than one group active at a time? Certainly:

```
  using group mary and tom and dick and harry
```

You can also add a group to the current set of active groups using "#group" to represent all active groups:

```
  using group mary and tom and #group
```

Constants and macros

If you are repeating a section of code or text multiple times in a program, you might want to create a macro to simplify program creation and maintenance.

Essentially, a macro is just a method for creating a shorthand reference that will later be replaced by a larger piece of code or text as specified in the macro definition. For example, if you want to include a piece of debugging code or if you are repeating the name of a company multiple times in a program, you could create a macro that contains that code or company name, and instead of typing the full text or code every time you need it, you could simply use the shorthand version that you have defined in the macro. Not only does this reduce the time required to create a program (by cutting down on typing), it also reduces the number of potential typos. Additionally, if the code or the name of the company should change, rather than searching through an entire program to replace each occurrence, you need only change the text contained in the macro, and it will automatically be changed in the rest of the program.

A macro is created using the `macro` and `macro-end` keywords. The `macro-end` keyword is required because a macro can contain any text or code, so there is no other way for the program to know where the macro definition ends and the rest of the program begins. The following macro definition creates a shorthand reference for the company name "OmniMark":

```
  macro om is
     "OmniMark"
  macro-end
```

All this does is tell the program that every time it encounters the short form "om", it's supposed to replace that short form with the full text OmniMark.

Although the following macro definition looks significantly more complex, the basic principles are the same:

```
  ; Macro to dump a switch shelf (for debugging purposes)
  ;
  macro Dump Switch token s is
     do
        output "Switch %@(s) has " || "d" format number of s || "
items%n"
        repeat over s
           output "  %@(s) @ %d(#item)"
           output " ^ " || key of s when s is keyed
           output " = "
           do when s
              output "true"
           else
              output "false"
           done
           output "%n"
        again
     done
  macro-end
```

This macro will replace each occurrence of the macro name "Dump Switch" with the code that appears between the keywords "is" and "macro-end" in the macro definition.

Once defined, macros are very simple to use. For example, the following program will output "Welcome to OmniMark. OmniMark is located in Ottawa, Ontario, Canada.":

```
  macro om is
     "OmniMark"
  macro-end

  process
     output "Welcome to " || om || ". "
     output om || " is located in Ottawa, Ontario, Canada.%n"
```

Macros don't have to be exact repetitions of a chunk of text. Macros can take "arguments", which are simply values given to the macro and used to change slightly the text that replaces the shorthand reference. The macro "Switch Dump", shown above, takes one token argument ("s").

One thing to note about macros is that all macro references are replaced with the full text of the macro before the program is compiled and run. Each time a macro is used, any actions contained in that macro are counted. So, if you define a macro that contains two actions and you use that macro five times in a program, it will count as ten actions towards the total number of actions in the program.

Including code from other files

OmniMark allows you to include code that is contained in another file in an OmniMark program by way of an `include` declaration. This feature allows you to easily recycle useful bits of code without having to resort to "copy and paste". Additionally, this means that if you decide you want to change something in that particular piece of code, rather than having to track down every individual usage of it, you need only make the changes in the single file that you have "included" in the other programs.

For example, you have defined an OmniMark function that you are particularly proud of, and that is useful in several different programs you're working on. For example, the function "Report", as follows:

```
define function Report
   value stream msg
as
   reopen log-file as file "MyProgram.log"
   put log-file date "xY/M/D h:m:s" || " MyProgram: " || msg || "%n"
   close log-file
```

The function Report simply outputs the value of the stream "msg" into the file MyProgram.log with a time stamp. If this function were the only thing to be in a file called report.xin, that function could then be included in any OmniMark program by naming that file in an include declaration:

```
include "report.xin"
```

Initialization and termination rules

In regular OmniMark programs, there is no real distinction between `process-start`, `process`, and `process-end` rules except that they are performed in that order. Additionally, `process-start` and `process-end` rules can be used in any of the aided-translation-type programs. OmniMark doesn't distinguish what can be done with these rules, but `process-start` and `process-end` rules should be used only for performing processes that must be executed at the beginning or end of a program, respectively. Usually these processes include whole-program initiation and termination functions. `process` rules should be used for the main processing within a program.

`process-start` rules allow you to do processing and produce output at the earliest stages of a program, more or less adjacent to macro and function definitions and global variable declarations. One use of a `process-start` rule would be to allocate handles and connect to a database:

```
  process-start

  local SQL_Handle_type EnvironmentHandle
  local SQL_Handle_type ConnectionHandle
  local SQL_Handle_type StatementHandle
  local counter RetCode

  set RetCode to SQLAllocEnv (EnvironmentHandle)
  output "Allocating environment handle - "
  do when RetCode != SQL_SUCCESS
     output "failed%n"
     halt with 1
  else
     output "passed%n"
  done

  set RetCode to SQLAllocHandle
     ( SQL_HANDLE_DBC, EnvironmentHandle, ConnectionHandle )
  output "Allocating connection handle - "
  do when RetCode != SQL_SUCCESS
     output "failed%n"
     halt with 1
  else
     output "passed%n"
  done

  set RetCode to SQLConnect ( ConnectionHandle, "omodbc", 20, "", 0,
"", 0 )
  output "Connecting to database - "
  do when RetCode != SQL_SUCCESS
     output "failed%n"
     halt with 1
  else
     output "passed%n"
  done
```

Similarly, a `process-end` rule could be used to disconnect from the database and free the handle resources:

```
process-end

set RetCode to SQLDisconnect ( ConnectionHandle )
output "Disconnecting from database - "
do when RetCode != SQL_SUCCESS
   output "failed%n"
   halt with 1
else
   output "passed%n"
done

set RetCode to SQLFreeHandle (SQL_HANDLE_DBC, ConnectionHandle)
output "Freeing connection handle resources - "
do when RetCode != SQL_SUCCESS
   output "failed%n"
   halt with 1
else
   output "passed%n"
done

set RetCode to SQLFreeHandle (SQL_HANDLE_ENV, EnvironmentHandle)
output "Freeing environment handle resources - "
do when RetCode != SQL_SUCCESS
   output "failed%n"
   halt with 1
else
   output "passed%n"
done
```

CGI programming

The Common Gateway Interface (CGI) is a protocol that allows web servers to invoke and communicate with other programs. Through CGI, a web server can call a program, send input to that program, and receive output from that program sent to a web browser.

Writing OmniMark CGI programs is similar to writing other programs in OmniMark, with a few differences in how your program has to handle input and output data.

Before you can begin using OmniMark CGI programs, you may have to make some minor changes to your computer and web server setup. Information about configuring your system should be available in your operating system or web server documentation.

Running an OmniMark CGI program involves three basic steps:

1. Invoking the program
2. Receiving input data
3. Doing the main processing

The third step is, of course, the heart of your program. In OmniMark CGI programs, this third step is exactly the same as in any other type of OmniMark program. You can do anything with OmniMark in a CGI program that you can do in any other OmniMark program, including processing markup, using external function libraries, or interacting with databases.

Invoking an OmniMark CGI program

Before an OmniMark CGI program can receive input data from a web server, the web server has to be able to successfully invoke the OmniMark program. To do this, the web server has to know where it can find the OmniMark executable.

Some web servers (for example, Apache) require that all CGI programs begin with a (hash-bang) directive that tells the web server where to find the software with which to run your CGI program. In an OmniMark CGI program, this line must include the full path and name of your OmniMark executable, followed by a command-line option:

```
#!/usr/bin/omnimark/bin/omnimark -sb
```

The "-sb" command-line option is new to OmniMark 5, and is the functional equivalent of a "-s" combined with "-brief".

The command-line option in the line might look a little strange to people who are used to the OmniMark command line, since the -sb option isn't followed by a filename. This is because the line implicitly refers to the file in which it occurs. For example, assuming that the program above is saved as helloworld.xom, your operating system will interpret the directive in this program as:

Copyright©1988-2004 Stilo International plc

```
     #!/usr/bin/omnimark/bin/omnimark -sb helloworld.xom
```

Note that you can use a line in an arguments file as well:

```
     #!/usr/bin/omnimark/bin/omnimark -f

     -sb helloworld.xom
     -x /usr/bin/omnimark/lib/=L.so
     -i /usr/bin/omnimark/xin/
```

The -f in the arguments file is interpreted the same way as the -sb in the program: the operating system interprets the -f as being followed by the name of the arguments file itself.

Using a line in OmniMark CGI programs does not reduce the portability of your code: OmniMark ignores the line as if it were a comment, as do web servers that don't require it. Note, however, that if you use a line, it must be the first line in the program. Having anything preceding the line will produce an error.

Web servers that don't use the line must be specifically configured to recognize OmniMark CGI programs and to find the OmniMark executable. This is usually done by creating file associations so that the system uses OmniMark to execute .xom and .xar files. For example:

```
  ".xom" = "d:\programs\omnimark -sb %s"
  ".xar" = "d:\programs\omnimark -f %s"
```

Receiving input data

CGI programs receive their input data from the web server in two ways: through environment variables and through standard input. The means used to retrieve the main input data depends upon the method used to send that data, which will usually be either GET or POST.

When you specify the GET method, the web server puts the main input data into an environment variable called QUERY_STRING. When you specify the POST method, the web server sends the main input data to the CGI program through standard input.

The method your OmniMark program uses when retrieving GET data will differ from the method used when retrieving POST data. Using the OmniMark CGI library, however, you can easily create an OmniMark CGI program that will successfully retrieve data sent by either of these methods.

Using the OmniMark CGI library

The OmniMark CGI library contains two functions and a macro:

- `cgiGetQuery` function
- `cgiGetEnv` function
- `crlf` macro

The `cgiGetQuery` function retrieves the data that the web server sends to your OmniMark CGI program, parses it, and puts the data on a keyed shelf of name/value pairs. The `cgiGetEnv` function retrieves the values of a variety of CGI-related environment variables and puts the data on a keyed shelf of name/value pairs. The CRLF macro allows you to easily use `%13#%10#` instead of a `%n` to insert a new line in your program output. Use `%13#%10#` instead of `%n` to ensure the portability of your code.

Because the OmniMark CGI function library uses functions in the OmniMark System Utilities library ("omutil"), you must declare and include the System Utilities library before including the "omcgi.xin" file.

Here's an example of the `cgiGetQuery` function in action:

```
declare #process-input has unbuffered
declare #process-output has binary-mode

include "omutil.xin"
include "omcgi.xin"

process
   local stream input-data variable initial-size 0

   cgiGetQuery into input-data

   output "Content-type: text/plain"
       || crlf
       || crlf

   repeat over input-data
      output key of input-data || " - " || input-data || crlf
   again
```

When called by a web server, the above program retrieves the query string from the QUERY_STRING environment variable if the GET method was used to send the data, or from `#process-input` if the POST method was used. The program parses the query string and puts the name/value pairs on the input-data shelf. The program then outputs a minimal HTTP header (`Content-type: text/plain`), and repeats over the input-data shelf, outputting the name/value pairs.

Note that in the program, `#process-input` is declared as `unbuffered`. Under normal circumstances, all OmniMark streams are buffered. When doing CGI programming, however, this buffering can cause endless amounts of trouble when you're trying to get your input data. If `#process-input` is buffered, your OmniMark program will never be able to get all the data it's waiting for. Therefore, you always have to tell your program to unbuffer `#process-input`. Do this with a declaration at the beginning of the program:

declare #process-input has unbuffered

The `cgiGetQuery` example shown above is an extremely simple CGI program. It does, however, do all the essential things that any OmniMark CGI program must do: it retrieves and parses the input data sent by the web server, and it sends a minimal HTTP header to the web server prior to sending the main bulk of the output.

Sending output data

Data written by your OmniMark CGI program is sent to standard output (which is the default `#process-output` stream) and is then relayed to the web browser. For the web browser to properly format the output, however, your program must output a minimal HTTP header before outputting the data to be displayed.

Setting `#process-output` to `binary-mode` ensures that the system running your CGI program will properly interpret the `%13#%10#` of the CRLF macro. This ensures that your code is portable among systems.

Declare the `#process-output` stream as `binary-mode` with the following declaration:

```
declare #process-output has binary-mode
```

Formatting output data

Here's a simple OmniMark CGI program:

```
; declarations and inclusions
declare #process-input has unbuffered
declare #process-output has binary-mode

include "omutil.xin"
include "omcgi.xin"

process
   output "Content-type: text/plain"
       || crlf
       || crlf
       || "Hello World!"
       || crlf
```

Assuming that this program is saved as "helloworld.xom" and has an accompanying arguments file saved as "helloworld.xar", you can invoke the program by using the path and name of the arguments file in a URL. For example:

```
http://localhost/cgi-bin/helloworld.xar
```

If the web server is properly configured, it will receive the request for the helloworld.xar file which will then call the helloworld.xom program. The program will execute, and the output (an HTTP header followed by "Hello World!") will be sent to the browser. The browser, in turn, will display that output as plaintext ASCII.

Before you send the main content of the page to the browser, you have to send an HTTP header. In most cases, a very minimal HTTP header will suffice, so long as it contains the content-type information for the page you are sending. The two most common types of page content are plaintext and HTML, the minimal HTTP headers for which are:

Copyright©1988-2004 Stilo International plc

```
      "Content-type: text/plain" || CRLF || CRLF
      "Content-type: text/html"  || CRLF || CRLF
```

Notice the two new lines (`|| crlf || crlf`) appended to the end of each of the HTTP header lines above. These new lines are required, because the blank line following the HTTP header indicates to the web browser that the HTTP header is complete, and that everything that follows is part of the page content. If you forget these new lines when sending output to the browser, the web browser will attempt to interpret all of the output as part of the header, which will result in an error.

Other than sending a minimal HTTP header followed by two new line characters, there is nothing special about the output of an OmniMark CGI program. Anything your program sends to standard output (`#process-output`, which is the default output destination) will be sent to the web browser.

Unbuffering #process-output

While you don't have to declare `#process-output` as `unbuffered` in your OmniMark CGI programs, it can sometimes be a good idea to do so. If `#process-output` is unbuffered, users can see responses from your CGI program a little more quickly than if `#process-output` is buffered. In most cases the change in response time is negligible. If your CGI program executes a large number of database queries or some particularly time-consuming processing, however, unbuffering `#process-output` can reduce the perceived "wait time" for the user, and your CGI program will seem more responsive. Again, the change in response time is often negligible, because OmniMark CGI programs tend to be extremely fast.

You can unbuffer `#process-output` with the following declaration:

```
  declare #process-output has unbuffered
```

Error message handling

Dealing with error messages that OmniMark CGI programs generate is a bit more complicated than debugging other OmniMark programs.

When you execute a regular OmniMark program on the command line, any errors that program generates are sent to standard error and displayed in the console window. Since OmniMark CGI programs are executed by the web server rather than through the command line, the error messages are sent back to the web server and usually end up being written in the web server error log file.

If your OmniMark CGI program has errors, the HTTP header that the web browser is expecting doesn't get sent. Instead, the web server receives one or more OmniMark error messages. Since an OmniMark error message does not qualify as a valid HTTP header, the "header" the web server receives (which is actually the OmniMark error message) gets recorded in the server error log as part of a "malformed header" error. The web browser in this case will usually display an HTTP 500 error, indicating an internal server error. To see the error messages, you'll have to open and read the server error log.

If you don't want to tackle the web server error log or if the OmniMark error messages aren't being written to it, you can create an error log for your OmniMark CGI program using the -log or -alog option in the arguments file:

```
#!/usr/bin/omnimark/bin/omnimark -f
-sb helloworld.xom
-alog helloworld.log
```

All error messages that OmniMark generates will be recorded in the file specified after the -log or -alog option. If errors occur in your program, your web browser will display a CGI error message that your CGI program returned an incomplete set of HTTP headers. This occurs because the web browser didn't actually receive anything; the program output (the error messages in this case) was sent to the log file instead.

Note that the -log and -alog command-line options should be used to create a log file only for debugging purposes. If you use these options in your CGI program when it is running in a production environment, your program could encounter concurrency problems if two instances of the CGI program are trying to write to the same log file simultaneously. To avoid these problems, stop using the -log or -alog option after you have finished debugging your program.

CGI-related environment variables

When a web server receives a request for a CGI program, it also stores other CGI-related information in environment variables. You can access these environment variables using the `UTIL_GetEnv` function in the OmniMark System Utilities library ("omutil"). Not all web servers will set all environment variables. You can use the `cgiGetEnv` function to retrieve all of the following environment variable values into a keyed shelf of name/value pairs:

- AUTH_TYPE: The authentication protocol currently being used. This variable is defined only if the server supports, and if access to the CGI program requires, authentication.
- CONTENT_LENGTH: The length, in bytes, of the information that the web server sends to the CGI program as input. This variable is used most often when the CGI program will be processing input being sent from an HTML form using the POST method.
- CONTENT_TYPE: The type of content that the web server sends to the CGI program as input.
- DOCUMENT_ROOT: This variable is set to the value of the DocumentRoot directive of the accessed website.
- GATEWAY_INTERFACE: The version of the Common Gateway Interface that the web server supports.
- HTTP_ACCEPT: A comma-separated list of MIME types that the browser software accepts.
- HTTP_ACCEPT_CHARSET: The character set that the client will accept.
- HTTP_ACCEPT_LANGUAGE: The language that the client will accept.
- HTTP_CONNECTION: The type of connection that the client and server use. For example, "HTTP_CONNECTION = Keep-Alive".

- HTTP_HOST: The IP address or host name of the accessed machine.
- HTTP_REFERER: The URI that forwarded the request to the called CGI program.
- HTTP_USER_AGENT: The browser software and operating system that the client system is running.
- PATH_INFO: Extra path information from the request.
- PATH_TRANSLATED: Maps the CGI program's virtual path (from the root of the server directory, for example) to a physical path that could be used to call the program.
- QUERY_STRING: Contains the encoded data from a form submission when that form is submitted using the GET method. If a form is submitted using the POST method, this environment variable is not set, as the encoded data is passed to the CGI program through standard input (in OmniMark terms, through #process-input).
- REMOTE_ADDR: The IP address of the client machine.
- REMOTE_HOST: The host name of the client machine.
- REMOTE_IDENT: Stores the user identification information returned by the remote identd (identification daemon). Few systems run this type of daemon process, however, so this environment variable is rarely set.
- REMOTE_PORT: The port number the client uses to originate the connection to request the CGI program.
- REMOTE_USER: The authenticated user ID of the user requesting the CGI program. This variable is defined only if the server supports, and if access to the program requires, authentication.
- REQUEST_METHOD: The method by which the CGI program was called (usually "GET" or "POST").
- REQUEST_URI: The URI of the request.
- SCRIPT_FILENAME: The URI of the requested CGI program.
- SCRIPT_NAME: The virtual path to the program.
- SERVER_ADMIN: The value of the ServerAdmin directive, if one is used, to set the email address of the web server in the web server's configuration file.
- SERVER_NAME: The configured host name for the server (usually www.something.com).
- SERVER_PORT: The number of the port on which the server software is listening for requests (usually 80, the default web server port).
- SERVER_PROTOCOL: The version of the web protocol that the server uses (for example, HTTP 1.0).
- SERVER_SOFTWARE: The web server software and version number.

Network programming

Network programming refers to all those operations that happen behind the scenes and involve cooperation between multiple applications running on multiple machines. These are the programs the user does not see, but which do the real work on the Internet and other networked environments.

The workhorse of network programming is the server. A server is a program that provides services to other programs. A server waits for a request from another program, decodes that request, and sends a response. Servers run unattended for days and weeks, so they must be robust.

The program requesting a service from a server is a client. A client issues a request and waits for a response from the server.

A middleware application is a program that acts as a go-between between two other programs. Generally, a middleware program adds value to the transaction. An active content server that receives requests from a web server and queries a database to fill the request is acting as middleware. It manages the communication between the web server and the database server and adds value by adding HTML tagging to the database results.

A middleware application acts as both a client and a server. Multi-tier architectures are possible in which multiple middleware applications broker and add value to communications across a network. Though they sound exotic, servers and middleware applications are easy to program in OmniMark using the connectivity libraries.

When embarking on a network programming project, you will need to know a little bit about protocols. A protocol is simply an agreement between a client and a server about how they will communicate. If you use a common published protocol, or publish your own protocol, you can enable any number of clients to communicate with your server. On the other hand, if you keep your protocol private (or even encrypted) you can help to secure your server against intrusion.

There are two important types of protocol you need to know about:

- Transport protocols
- Application protocols

Transport protocols are used to actually get messages across the network from one machine to another in good order. TCP/IP is the transport protocol used on the Internet, and supported by OmniMark's network libraries and the TCPService and TCPConnection OMX components.

An application protocol is an agreement about what constitutes a message and what the message means. While disk files have natural ends, a network message is just a stream of bytes over an open connection. You have to look at the data itself to determine if you have found the whole message. The OmniMark I/O Protocol library supports all the common methods of delimiting a message.

Once you have a complete message you must decoded it to see what it means, and then generate and send the appropriate response. OmniMark is the ideal language for decoding network protocols. Its streaming features make it very easy to interpret a message and formulate a response quickly.

A simple server

The following is a simple OmniMark server program. This server returns the first line of a nursery rhyme when it receives a message naming the principal character of that rhyme.

```
declare catch server-die
include "omioprot.xin"
include "omtcp.xin"
include "builtins.xin"

define switch function TCPservice-is-working
       read-only TCPService the-service
       as

   do when TCPServiceIsInError the-service
      local stream errorReport variable initial-size 0
      TCPServiceGetStatusReport the-service into errorReport
      log-message "TCPService Error:"
      repeat over errorReport
         log-message errorReport
      again
      return false
   else
    return true
   done


process
   local TCPService my-service
   set my-service to TCPServiceOpen at 5432
   throw server-die unless TCPService-is-working my-service
   repeat
      local TCPConnection my-connection
      local stream my-response
      set my-connection to TCPServiceAcceptConnection my-service
      throw server-die unless TCPService-is-working my-service
      open my-response as TCPConnectionGetOutput my-connection
      using output as my-response
         submit TCPConnectionGetLine my-connection
      catch #program-error
   again
   catch server-die

find "Mary%n"
   output "Mary had a little lamb%13#%10#"

find "Tom%n"
   output "Tom, Tom, the piper's son.%13#%10#"

find "Jack" or "Jill" "%n"
   output "Jack and Jill went up the hill.%13#%10#"
```

```
find "die%n"
   output "Argh! Splat!%13#%10#"
   throw server-die
```

A server operates rather like a telephone. First we place it in service by assigning it a telephone number. Then it must wait for a call. When a call comes it must answer it, listen to the message, and make an appropriate response. The conversation may consist of a single exchange, or of multiple exchanges. When the conversation is over, it hangs up and go back to waiting for the next call.

The essential operation of a server, then, comes down to three things:

- Start up: put the server in service
- Request loop: wait for calls, respond, and repeat
- Shut down: take the server out of service

Because it runs for a long time and has to handle many requests, a server has two overriding performance requirements:

- No matter what happens while servicing a request, the server must not crash. It must stay running.
- No matter what happens while servicing a request, the server must always return to a consistent ready state when the request is complete. If the server was in a different state for each request, its responses would not be reliable.

Let's look at how our sample server meets these requirements, line by line:

```
process
   local TCPService my-service
   set my-service to TCPServiceOpen at 5432
```

This is the code that puts the server in service. It uses a TCPService OMX component to establish a service on port 5432 of the machine it is running on. The server's address (its phone number) will be the machine's network address combined with the port number. Many different servers can run on the same machine using different ports.

```
repeat
      local TCPConnection my-connection
      ...
      set my-connection to TCPServiceAcceptConnection my-service
again
```

This is the code that listens for an incoming call. TCPServiceAcceptConnection waits for a client to connect. When it receives a connection, it returns a TCPConnection OMX which represents the connection to the client. The TCPConnection variable "my-connection" is declared inside the repeat loop so that it will go out of scope at the end of the loop, providing automatic closure and cleanup of the connection.

```
repeat
      local TCPConnection my-connection
      ...
```

```
        set my-connection to TCPServiceAcceptConnection my-service
        ...
           submit TCPConnectionGetLine my-Conn
  again
```

A TCPConnection OMX provides an OmniMark source so that data can be read from the client. Reading data from a network connection, however, is different from reading from a file. While you can either read from a file or write to it, but not both, a network connection, like a telephone connection, is two way. This means that OmniMark cannot detect the end of a message on a network connection the way it detected the end of a file. The connection stays open and there could always be more characters coming. For this reason, all network data communication requires a specific application protocol for determining the end of a message. OmniMark provides support for all the common application protocols used for this purpose through the I/O Protocol library. The TCP library uses the I/O Protocol library to support the common forms. In this case we are using a line based protocol. In our request protocol, the end of a message is signaled by a line-end combination (ASCII 13, 10). The TCPConnectionGetLine function provides a source for line-end delimited data and closes that source when it detects a line end. We submit data from that source to our find rules which will analyze the message and generate the appropriate response.

```
  repeat
        ...
        local stream my-response
        ...
        open my-response as TCPConnectionGetOutput my-connection
        using output as my-response
           submit TCPConnectionGetLine my-connection
        ...
     again
```

Our TCPConnection OMX represents a two way network connection. Not only must we get a source from it to read ddata, we must also attach an output stream to it so that we can send data over the connection to the client. We do this with the TCPConnectionGetOutput function.

Once our submit is prefixed with `using output as my-response`, our find rules are reading from and writing to the network connection.

```
  find "Mary%n"
     output "Mary had a little lamb%13#%10#"

  find "Tom%n"
     output "Tom, Tom, the piper's son.%13#%10#"
```

Ours is a line based protocol, but line ends are different on different platforms (13,10 on Windows, 10 on UNIX). OmniMark's "%n" is a normalized line end. It will match either form. If you output "%n" it will output the form appropriate to the platform the program is running on. Across a network, which can include machines from different platforms, we have to pick one for ourselves. Our protocol specifically requires 13, 10. But for matching purposes, we use, "%n" so that even if the client forgets to send the appropriate line end sequence, we can still read the message. When we send, however, we explicitly send "%13#%10#" rather than "%n". In this we are following an important maxim of network programming: Be liberal in what you accept, conservative in what you send.

```
  find "die"
     output "Argh! Splat!%13#%10#"
     throw server-die
```

This is the find rule that detects the poison pill message. To ensure an orderly shutdown, we provide a method of terminating our server by sending it a message to shut itself down. (In a production system, you might want to pick a slightly less obvious message for the poison pill.)

Shutting down the server is an exception to normal processing. We accomplish it by initiating a throw to a catch named server-die.

```
  process
     ...
     repeat
        ...
     again
     catch server-die
```

We catch the throw to server-die after the end of the server loop. OmniMark cleans up local scopes on the way, ensuring a clean and orderly shutdown. We are at the end of the process rule now, so the program exits normally.

Error and recovery

A server needs to stay running despite any errors that occur in servicing a particular request. On the other hand it should shut down if it cannot run reliably. The following code provides for both these situations:

```
  define switch function TCPservice-is-working
         read-only TCPService the-service
         as

     do when TCPServiceIsInError the-service
        local stream errorReport variable initial-size 0
        TCPServiceGetStatusReport the-service into errorReport
        log-message "TCPService Error:"
        repeat over errorReport
           log-message errorReport
        again
        return false
     else
      return true
     done

  process
     set my-service to TCPServiceOpen at 5432
     throw server-die unless TCPService-is-working my-service
     repeat
        ...
        set my-connection to TCPServiceAcceptConnection my-service
        throw server-die unless TCPService-is-working my-service

        ...
        catch #program-error
     again
```

If there is an error in processing a request, OmniMark initiates a throw to #program-error. We catch the throw at the end of the server loop. This provides for an automatic cleanup of any resources in use in servicing the request in progress, and assures that the server returns to its stable ready state. (No attempt is made to rescue the specific request in which the error occurred. In a production server you would want to provide such error recovery, but make sure you always have a fallback that aborts the current request and returns to a stable ready state.)

In the unlikely event that something goes wrong with the TCPService component, there is nothing much you can do except shut down the server. The current version of the TCP library does not support catch and throw, so you have to do an explicit test for errors in the service whenever you use it. If an error is detected, log it and then throw to server-die to shut down the server.

This simple server program has everything you need for a robust and usable production server. You would need to adapt the code to the protocol you are using, but apart from that, once input and output are bound to the connection, everything else is just regular OmniMark programming.

A simple client

Any client program, written in any language, can use our server as long as they know the protocol. Here is a simple client written in OmniMark:

```
  process
     local TCPConnection my-connection
     local stream my-request
     set my-connection to TCPConnectionOpen
                       on "localhost" at 5432
     open my-request as TCPConnectionGetOutput my-connection
     using output as my-request
        output #command-line-names item 1 || "%13#%10#"
     close my-request

  repeat
     output TCPConnectionGetCharacters my-Conn
     exit unless TCPConnectionIsConnected my-Conn
  again
```

This client is called with the name of the nursery-rhyme character on the command line and prints out the line it receives from the server. Let's go through it line by line:

```
  set my-connection to TCPConnectionOpen
                    on "127.0.0.1" at 5432
```

Like the server program, the client uses a TCPConnection OMX component to create a connection. Unlike the server it does not require a TCPService component, as it is not establishing a service, but simply making a connection to a service established elsewhere. The client takes a more active role than the server, however. While the server waits for a call, the client must take the initiative and make a call. It does this with the TCPConnectionOpen function. The TCPConnectionOpen function takes a network and port address for a server and when the connection is made it returns a TCPConnection OMX, which we can write to and read from just as we did in the server program.

```
repeat
   output TCPConnectionGetCharacters my-Conn
   exit when TCPConnectionIsInError my-Conn
again
```

When we read the data returned from the server we actually have two choices. Since our is a line based protocol, we could use TCPConnectionGetLine to read the response. But we also know that the server will drop the connection as soon as it has finished sending data. (This behavior is part of our protocol as well.) So we choose to keep reading data until the connection is dropped. This way we will get at least partial data even if something goes wrong and the server never sends the end of line. Be conservative in what you send and liberal in what you accept.

Clients for common servers

Most of the client program you write in OmniMark will probably be for well-known servers such a HTTP (Web), FTP, or SMTP/POP (Mail). OmniMark's connectivity libraries provide direct support for these and other common protocols, greatly simplifying the task of retrieving data from these servers.

Financial calculation

You can perform financial calculations in OmniMark using the BCD data type to represent monetary values and fractional numbers such as tax rates. Unlike floating point numbers, BCD numbers provide accurate fractions for financial calculations.

The following code sample shows basic financial calculations using BCD numbers. Note that literal BCD values must be preceded by the keyword `bcd`.

```
include "ombcd.xin"

process
    local bcd price
    local bcd tax-rate
    local counter quantity
    local bcd total

    set price to bcd 19.95
    set tax-rate to bcd 0.07
    set quantity to 28
    set total to price * quantity * (1 + tax-rate)
    output "<$,NNZ.ZZ>" format total
```

The format string "<$,NNZ.ZZ>" uses the BCD template language to create an output string with a leading "$", commas separating digits into groups of three, and two digits after the decimal point.

The following code shows how to read a decimal number from an HTML form and assign the value to a BCD number:

```
include "ombcd.xin"
include "omcgi.xin"

declare #process-input has unbuffered
declare #process-output has binary-mode

process
    local stream form-data variable
    local stream cgi-data variable
    local bcd price
    local bcd tax-rate
    local counter quantity
    local bcd total

    CGIGetEnv into cgi-data
    CGIGetQuery into form-data

    set price to bcd form-data key "price"
    set tax-rate to bcd 0.07
    set quantity to 28
    set total to price * quantity * (1 + tax-rate)
    output "Content-type: text/html"
        || crlf
        || crlf
        || "<html><body>"
        || "<H1>Order confirmation</H1>"
        || "Your total comes to: "
        || "<$,NNZ.ZZ>" format total
```

Markup languages

In OmniMark documentation, almost all references to "markup languages" are actually references to element-based markup languages that have been created using either SGML (Standard Generalized Markup Language) or XML (the eXtensible Markup Language). A markup language is a full set of markup instructions which can be used to comprehensively describe the structural information content of a piece of text. Markup tags are the actual pieces of code that are added to the electronic document.

When you create a markup language using SGML or XML, you are defining a set of tags which can then be used to demarcate the structure of your documents. Because SGML and XML are used to create sets of markup tags, they are "meta languages", languages that describe other languages. The benefits of using SGML and XML to create these markup languages are numerous; since they are internationally recognized standards, this standardization allows the marked up documents to be portable across platforms. Additionally, with SGML or XML you are able to create fully customized languages that will most comprehensively treat your unique markup requirements.

Copyright©1988-2004 Stilo International plc

Markup instructions can be interpreted by applications that use the markup to determine the formatting of a document. When used this way, the markup usually has an immediate and specific effect on the text, either by changing the appearance of the characters (by rendering them in a bold or italic font, for example), or by affecting the positioning of the text (such as by changing the margin, indent, and spacing values).

For example, HTML is a markup language whose elements describe the formatting of a document when that document is processed by an HTML browser or similar application:

```
<html>
<head>
<title>Hamlet</title>
</head>
<body bgcolor="#ffffff" text="#000000">
<div align=center>
<font size=5>
<b>Hamlet</b>
<p><font size=3><b>Act I, Scene I</b>
<p><i>Francisco at his post. Enter to him Bernardo</i>
</div>
<p><b>Bernardo:</b> Who's there?
<p><b>Francisco:</b> Nay, answer me: stand and unfold yourself.
<p><b>Bernardo:</b> Long live the king!
</body>
</html>
```

The elements in this short HTML document affect the alignment, size, and appearance of the text.

When interpreted by applications, specific markup languages also detail the structure of a document, identifying the various internal components of which it is made. These components can include things such as paragraphs, headings, sections, subsections, names, titles, chapters, volumes, articles, and so on. The possible list of document components is endless, but each specific markup language can only be used to identify a small set of these.

For example, the following document is marked up using a very simple language created with XML:

```
<play>
<title>Hamlet</title>
<act><scene>
<scenedesc>Elsinore. A platform before the castle.</scenedesc>
<stagedir>Francisco at his post. Enter to him Bernardo</stagedir>
<char>Bernardo</char>
<line>Who's there?</line>
<char>Francisco</char>
<line>Nay, answer me: stand, and unfold yourself.</line>
<char>Bernardo</char>
<line>Long live the king!</line>
</scene></act>
</play>
```

The elements in this short XML document are used to identify, to an XML application, the components and structure of the information it contains.

Wherever possible, OmniMark uses the same names and terminology as the SGML and XML specifications.

Markup rules

OmniMark provides a complete set of markup rules that can be used to process documents that have been marked up with SGML- or XML-based markup languages. These rules correspond to all the features of SGML and XML, and are as follows:

`data-content` rules, allowing you to capture the parsed character data content of elements.

`document-end` rules, fired immediately after the parsing of an SGML or XML document has been completed in an aided translation type program. `document-end` rules are "termination" rules, meaning that they're useful for doing process cleanup and final processing before a program completes.

`document-start` rules, fired just before the implicit parsing of an SGML or XML document begins. `document-start` rules are "initialization" rules, making them useful for doing any sort of program setup that has to be done before the main processing begins. These rules can only be used in aided translation type programs.

`dtd-end` rules, used in programs that process marked-up documents that contain a DTD. `dtd-end` rules are fired after the DTD has been completely processed.

`dtd-start` rules, fired after the doctype element has been specified in a DTD, but before the main part of the DTD is processed.

`element` rules, used to execute specified actions when the element named in the element rule is encountered in the input document. It is important to note that each element that appears in an SGML or XML document must be uniquely accounted for in an OmniMark program. You must have an `element` rule that can be fired for each individual occurrence of every element in a document.

`epilog-start` rules, used in programs that process marked-up documents that contain a document epilog. `epilog-start` rules fire just before the processing of the document epilog begins.

`external-data-entity` rules, used to specify special processing of external data entities that are encountered in SGML or XML documents. Note that you must have an `external-data-entity` rule that can be fired for each occurrence of an external data entity in a document.

`external-text-entity` rules, used to provide the full-text replacement for each external text entity that appears in the input document.

`invalid-data` rules, used in processing SGML and XML documents and give you control over how erroneous data appearing in the input document is processed.

`marked-section` rules, provided so that you can specify the processing of any type of marked section that appears in an SGML or XML document. Marked sections types include cdata, rcdata, ignore, and include.

`markup-comment` rules, fired whenever a markup comment is encountered in an input document and allow you to control how the content of the comment is processed.

`markup-error` rules, fired if an error is encountered in the markup of an input document.

`processing-instruction` rules, giving you control over how processing instructions that are encountered in SGML and XML documents are processed.

`prolog-end` rules, used in programs that process marked-up documents that include a document prolog. `prolog-end` rules are fired just after the prolog has been completely processed.

`prolog-in-error` rules, fired if an error is encountered in the prolog of a marked-up input document.

`sgml-declaration-end` rules, used in programs that process SGML documents. All SGML documents contain an SGML Declaration, whether it be explicit or implicit, so these rules will always fire if they are used. `sgml-declaration-end` rules fire after the declaration has been completely processed.

`translate` rules, fired when data content matching a specified pattern occurs within an element of an SGML or XML input document.

copyrigh © Stilo International plc 1988-2004