

Inquisit Programmer's Manual

Sean Draine, PhD.

Copyright 2022, Millisecond Software, LLC

Contents

The Inquisit Lab Programming Environment.....	6
Getting Started	7
Editing or Running a Script	8
Using Image, Video, Sound, HTML, and Other Media Files	8
Accessing Data Files	9
Debugging Inquisit Scripts.....	9
Message List.....	9
The Debugger Watch Window	9
Running your code.....	10
The Inquisit Programming Language	12
Inquisit Markup Language (IQML).....	12
Elements and Attributes.....	12
Getting Help.....	13
Inquisit Scripting Language (IQScript).....	13
The IQML Object Model.....	15
Defaults	15
canvasaspectratio.....	15
fontstyle	16
screencolor	16
inputdevice.....	16
windowsize	16
Stimuli.....	17
text	17
picture	18
shape	18
video	19
html	19
clock.....	19
sound	20
Positioning and Layout of Visual Stimuli.....	20
Erasing Stimuli.....	21

Inquisit Programmer's Manual

item	21
Trials.....	23
Presenting Stimuli.....	23
Measuring Responses	25
inputdevice.....	25
validresponse.....	25
correctresponse	26
Measuring Response Time	26
Specialized Trials	26
Openended	27
Slidertrial.....	27
Likert	27
Blocks.....	28
preinstructions	29
trials	29
postinstructions.....	29
Expt (experiment)	29
preinstructions	30
blocks.....	30
postinstructions.....	30
Presenting Instruction Pages	30
instruct	30
page.....	31
htmlpage	32
Data element	32
columns	33
separatefiles	34
Summary Data	34
columns	35
separatefiles	35
Programming a Simple Test with IQML	36
Emotional Dot Probe	36
Defining the Stimuli.....	36

Inquisit Programmer's Manual

Defining the Trials.....	38
Defining the Blocks.....	41
Defining the expt.....	41
Defining the instruction pages.....	42
Default values.....	44
Data.....	44
Programming with IQScript.....	46
IQScript Syntax.....	49
Value Types.....	49
Getting and Setting Object Properties.....	49
Calling Object Functions.....	50
Global Functions.....	51
Operators.....	51
Values element.....	52
Parameters element.....	53
Keywords and Statements.....	54
Variable declarations: var.....	54
Conditional Statements: if, else if, and else.....	55
Conditional Looping.....	57
Return Statement.....	57
Expressions element.....	58
Built-in Elements.....	59
Script Element.....	59
Display Element.....	60
Computer Element.....	60
Inquisit Element.....	61
Mouse Element.....	61
Handling IQML Events with IQScript.....	62
Stimulus Onprepare Event.....	62
Trial Ontrialbegin and Ontrialend Events.....	63
Block Onblockbegin and Onblockend Events.....	64
Expt Onexptbegin and Onexptend Events.....	64
Trial Ivalidresponse and Iincorrectresponse Events.....	65

Inquisit Programmer's Manual

Conditional Branching with IQScript	67
Text Insertion Macros	68
Advanced Stimulus Presentation	70
Selecting Items.....	70
Random Selection	70
Sequential Selection	70
Synchronized Selection Between Stimuli	71
Animation	72
Path Animations.....	72
Points animations	74
Circle animations	75
.....	75
Size Animations	76
Creating Dynamic Stimuli	77
Insertstimulustime() and insertstimulusframe() functions	78
Text insertion macros	78
Working with lists.....	79
List Attributes	79
Using Lists for Advanced Selection	80
Using Lists for Computation.....	81
Trial Duration, Timeouts, and Inter-Trial Intervals.....	83
pretrialpause	83
stimulusframes.....	83
response	83
timeout	84
posttrialpause	84
trialduration	84
Running Multiple Scripts with the Batch Element	85
Between-Group Manipulations	86
Between-Session Manipulations.....	86
Resources for Programming your Own Scripts.....	88

The Inquisit Lab Programming Environment

At the heart of an Inquisit experiment is one or more scripts which contain the commands that instruct Inquisit how a given experiment should run. The script specifies the logical flow of the experiment, which stimuli to present, how participants are meant to respond, and which data should be recorded. Inquisit scripts are standard text files (UTF-8), which enables Inquisit to handle characters from any alphabet in the world, including bidirectional text such as Hebrew and Arabic. You can edit your script using any editor that supports plain text such as Notepad, Visual Studio, or XCode, but if you try to save a script using a proprietary format such as Microsoft Word (e.g. *.docx), it can't be run in Inquisit.

While advanced developers may prefer to use their own editor for programming Inquisit scripts, most will use Inquisit Lab as their development environment. Inquisit Lab has a rich editor with colorization to make your code more readable, autocomplete lists showing which commands are valid in a given context, tooltips showing syntax hints, context sensitive help via the F1 command, a navigation window that allows you to jump to the elements in your script, a validator that reports errors, and debugging features such as the ability to pause a script and inspect the current values of its properties through the Watch Window.

Of course, Inquisit Lab also allows you to run your script and save data, using either a human responder or its built-in automatic test monkey. You can examine the data files to test that randomization is working correctly, the correct number of trials are being run, and that test scores are calculated correctly.

Inquisit Programmer's Manual

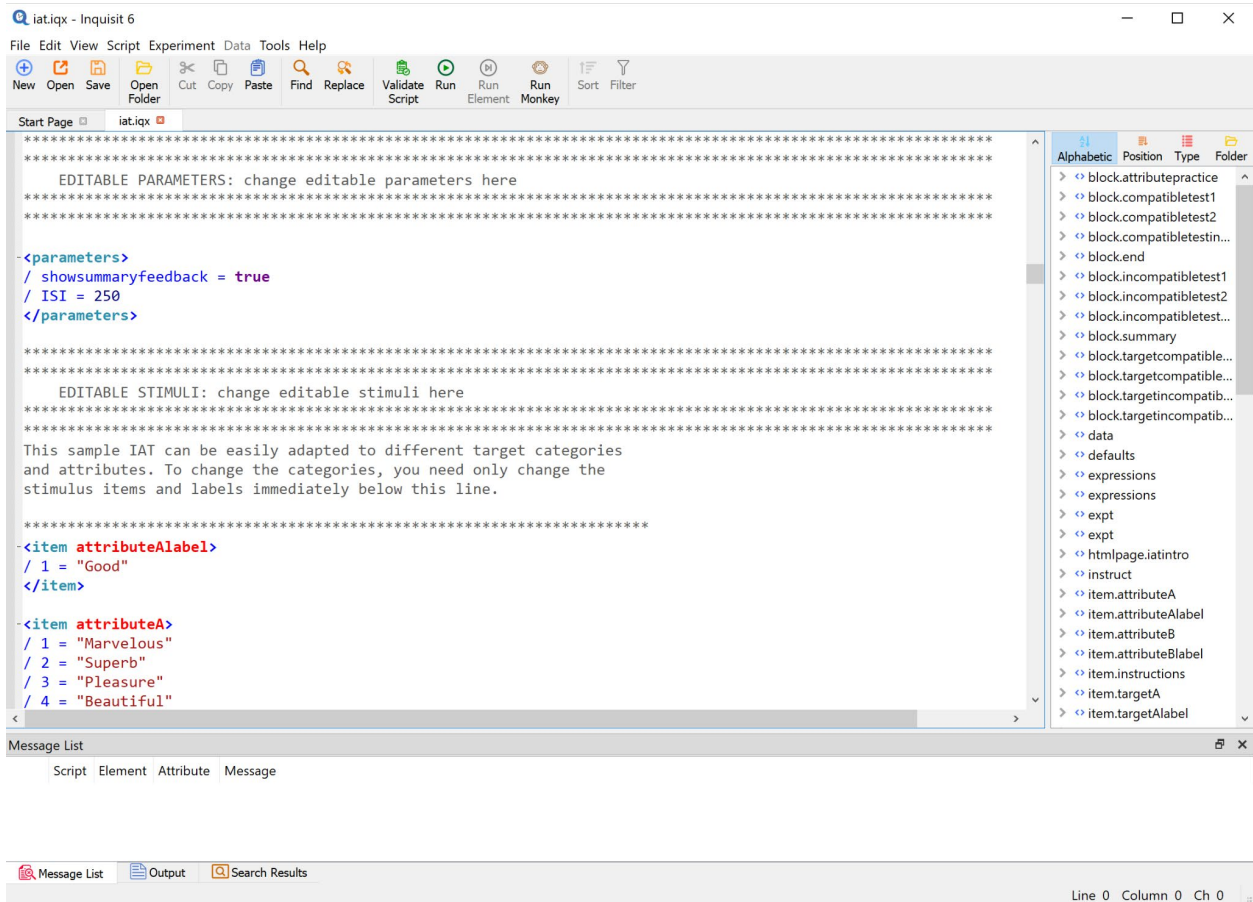


Figure 1. The Inquisit Lab programming environment

Getting Started

The first step for programming your experiment is to download Inquisit Lab from <http://www.millisecond.com/download>. When you run Inquisit Lab for the first time, you'll see a Registration Window appear. If you have purchased an Inquisit Lab license, click the "Get Registration Key" and enter your user id and password to register Inquisit. If you don't have an Inquisit Lab license, click the Run Free Trial button to take advantage of the free 30-day trial.

Inquisit then opens and displays the Start Page, which gives you three options under the Start section. You can click "Download Tests from Millisecond", which will take you to the Millisecond Test Library where you can choose from hundreds of tests to download. If you have an existing script on your computer, you can select "Open at test...", which will enable you to browse to the folder with your script and open it. You can also select "Create a New Test", which will open a blank script for you to start from scratch. In many cases, you will likely start with an existing script that is similar to the procedure you wish to run and then modify that script as needed. Once you get to know the Inquisit scripting language well, it can often be just as efficient to program your ideas from scratch.

Editing or Running a Script

If you are editing an existing script, you can open it in Inquisit Lab in one of two ways.

1. You can simply double-click the script in Windows Explorer or Mac Finder, and Inquisit will show a dialog with options to Run or Edit the script. If you've configured your computer to display file extensions, Inquisit scripts use either the .iqx extension or .iqzip extension. If your computer doesn't display file extensions (default for Mac and Windows), script files will be displayed with the Inquisit stopwatch icon. If you click Run, it will run the test. If you click Edit, it will open the script in the Inquisit Lab editor. If the script has any errors, it will be opened in the editor with the errors displayed below in the message list. Otherwise, the test will run.

2. You can also open a script by first launching Inquisit Lab by double-clicking the Inquisit Lab icon on your desktop. You can then select the "Open at test..." command from the Start Page, and then select your script using the File Open dialog. Inquisit will open the script in the editor, where you can make any desired changes. If you wish to run the script, simply click the Run button on the toolbar at the top (it's the button with the "play" icon). If the script has any errors, these will be displayed below the editor in the message list. Otherwise, the test will run.

Alternatively, if you simply wish to generate a sample data file from our script, you can use the "Run Monkey" command, and Inquisit will automatically run your script providing random responses. This is a convenient way to generate sample data, particularly for tests that take a long time to run.

Using Image, Video, Sound, HTML, and Other Media Files

Many Inquisit tests make use of media such as images, video, sound, or HTML files. When you download a test from the Millisecond Test Library that requires other files, these are all bundled together into a single "iqzip" file. An "iqzip" file is just a standard zip file with the extension changed to "iqzip" so that it can be linked with Inquisit when you double-click it to open it. The first time you open an iqzip file with Inquisit Lab, Inquisit will do the following:

1. Automatically unzip all of the files into a folder named after the iqzip file (for example, "bart.iqzip" will be unzipped into a folder named "bart".)
2. Delete the original iqzip file.
3. Open the script file contained in the iqzip file in the editor

The script file and media files that it uses are thus unzipped to the same folder. By saving the files to the same folder, Inquisit can find the media files when you run the associated script.

When you are creating a new script that uses external media files, it's usually easiest to follow this same convention and keep all of the media files in the same folder as the script. If you try to run a script and you get an error about Inquisit not being able to find a file, that usually means the file is in a separate folder from the script, so Inquisit can't find it.

Accessing Data Files

By default, Inquisit saves the data from a given test in a subfolder of the folder containing the test script. The subfolder is called "data". Additional subfolders called "voicerecordings", "screencaptures", or "photocaptures" may be created for tests that utilize voice recording, screen captures, or that take photos. You can open the data files by simply double-clicking them in Windows Explorer or Mac Finder. Inquisit Lab will display them in a tabular editor similar to Excel. The editor will allow you to sort and filter the data as well as save them as an Excel spreadsheet (Windows only).

If you run your script by first opening it in Inquisit Lab and clicking the Run button, when the script is complete it will return you to the Inquisit Lab editor, and links for each data file saved will be displayed in the Message List below the editor. Simply click the links to open the data files.

Debugging Inquisit Scripts

Inquisit Lab includes a number of tools to help you test and debug your scripts.

Message List

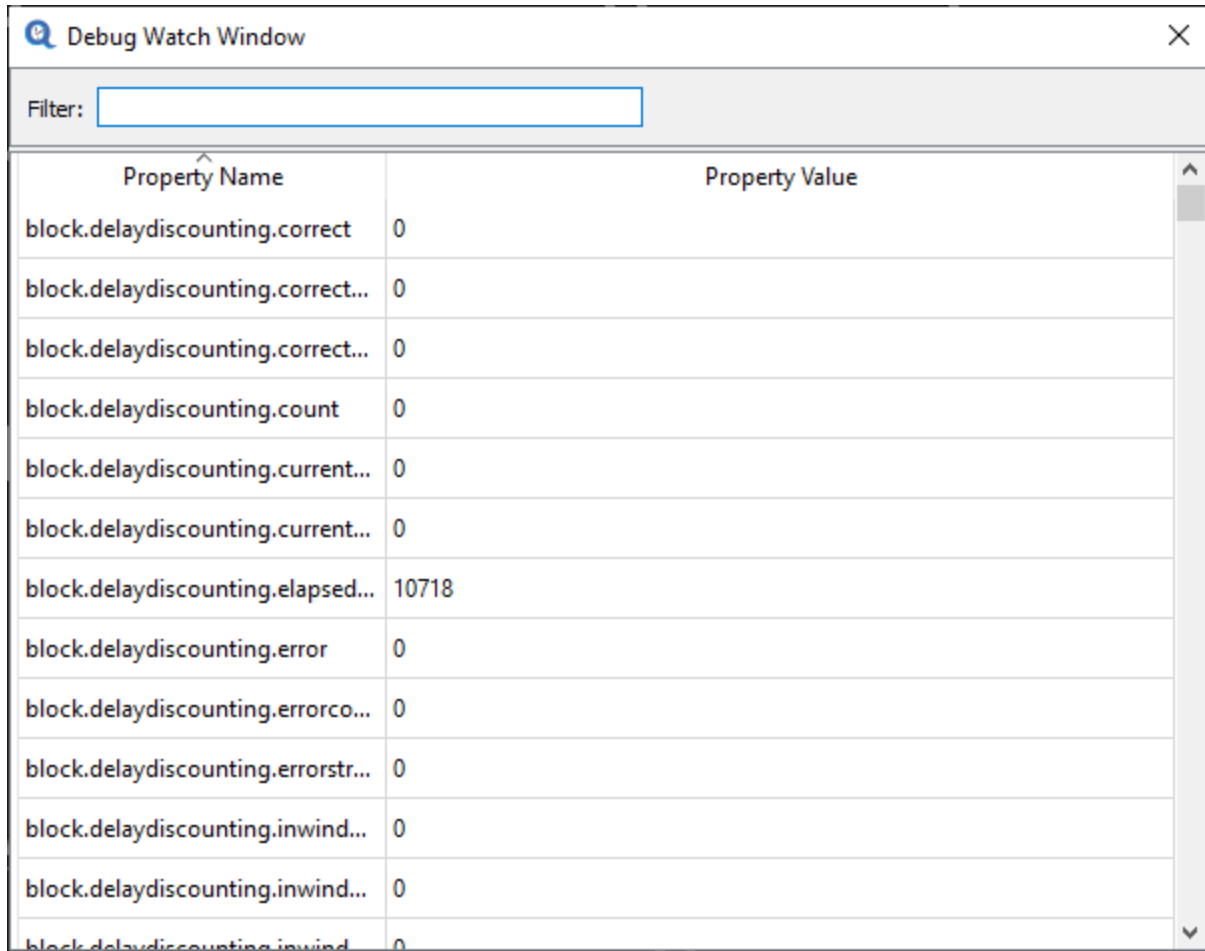
The message list is a critical tool when programming a script, providing feedback when Inquisit encounters problems during a script run. If you select Validate or Run from the menu and your script has errors, Inquisit will output a detailed error message to the Message List window at the bottom of the screen. If you double-click an error message, the editor will jump to the code for the element causing the error.

The image below shows the Message List with a single error in it. The problem is that <values> doesn't contain a definition for "countdelay". In the editor, you can see an error icon near the problematic code.

The Debugger Watch Window

When you are running a script in Inquisit Lab or Web, you can use the Debugger Watch Window to pause the script and check the current values of all of the properties defined for the script. This is handy for troubleshooting errors in calculations. To open the Debugger Watch Window, press the following keys while the script is running:

Ctrl+D



The screenshot shows a window titled "Debug Watch Window" with a close button in the top right corner. Below the title bar is a "Filter:" text box. The main area contains a table with two columns: "Property Name" and "Property Value". The table lists various properties related to "block.delaydiscounting" and their corresponding values.

Property Name	Property Value
block.delaydiscounting.correct	0
block.delaydiscounting.correct...	0
block.delaydiscounting.correct...	0
block.delaydiscounting.count	0
block.delaydiscounting.current...	0
block.delaydiscounting.current...	0
block.delaydiscounting.elapsed...	10718
block.delaydiscounting.error	0
block.delaydiscounting.errorco...	0
block.delaydiscounting.errorstr...	0
block.delaydiscounting.inwind...	0
block.delaydiscounting.inwind...	0
block.delaydiscounting.inwind...	0

The Debugger Watch Window lists the names and values of all properties in the script. To find a given property, you can scroll the list or type the property name in the filter textbox, and only those properties matching your filter are displayed.

The Debugger Watch Window can also be launched programmatically if you wish to inspect property values at a specific point in your script's logic. The IQScript function for opening the window is as follows:

```
script.debugbreak();
```

Running your code

While programming a script, you will typically spend some time jumping back and forth between editing the code and running the script to test your changes. If the task you are programming is particularly long, it can be quite cumbersome to run the entire script just to view a single change. Inquisit provides a number of shortcuts to enable you to run just the code you are interested in testing.

Aborting a Script

You can abort a script in progress by pressing the following keys on the keyboard:

Ctrl + Q

Swipe a Z across the screen with 2 fingers on a touchscreen device.

When this command is entered, Inquisit immediately ends the script and saves any data collected up to that point.

Skipping a Block

You can skip the remaining trials in the current block by pressing the following keys:

Ctrl + B

Running Specific Elements

As a convenience for debugging and testing, Inquisit Lab enables you to run specific elements in your script in isolation rather than running the entire script. For example, you can run a given stimulus element, and Inquisit Lab will show how that stimulus appears on the screen independently of the rest of the script. Similarly, you can run specific trials and blocks in isolation.

To run a given element, simply right-click on the element in the editor, and select the “Run <element name>” command, where <element name> is the name of the element you clicked on.

Test Monkey

One of the most useful tools for testing and debugging Inquisit scripts is the Test Monkey, which automatically runs your script and generates sample data without requiring a human to make any responses. By running the monkey, you can test your script for errors and generate data files that can be used to verify script logic, ensure trial counts are correct across conditions, and prepare for data processing and analysis.

To run the Test Monkey, select the “Run Monkey” command from Inquisit Lab's Tools menu. The monkey will run the test choosing randomly from any responses defined as valid for the task. When the monkey is finished, you'll have data files from its session. For better simulations of human responding, the average response time, accuracy rate, and valid responses for the monkey can be configured in a given test script.

The Inquisit Programming Language

The Inquisit scripting language was designed for psychological experimentation, and much of the terminology used is taken from what you would find in a typical procedural section of a published article.

Inquisit Markup Language (IQML)

The language itself includes an object oriented markup language called IQML, which provides a simple, declarative syntax for specifying the objects or components of an experiment and linking them together. With IQML, you can create elements representing various pieces of an experiment including all of the stimuli presented to be presented, the different types of trials that present stimuli and measure responses, blocks that specify randomized or pre-ordered sequences of trials, and experiment which specifies the sequence of blocks to run. Other elements control presentation of task instructions, the data that are saved, lists for storing arrays of elements or values, and even elements for interacting with eye trackers.

Elements and Attributes

IQML provides a simple, convenient syntax for declaring the various objects of an experiment (referred to as “elements”) and setting properties on those objects (referred to as “attributes”). Elements are declared using open and close tags similar to those used in XML or HTML. For example, Inquisit has a shape element that can be declared as follows:

```
<shape redbox>  
</shape>
```

In this example, “shape” is the type of element, and “redbox” is the name of the element. Other elements that use the shape can refer to it using its name.

Attributes are specified within open and close tags using a “/” followed by the attribute name followed by “=” which is then followed by the values to which the attribute is set. To continue with the shape example, we can set its type, color, and size using the corresponding attributes:

```
<shape redbox>  
/ shape = rectangle  
/ color = red  
/ size = (20%, 20%)  
</shape>
```

As you can see, the syntax for each attribute differs depending on the type of values the attribute takes. The /shape attribute can be set to rectangle, circle, or triangle. The /color attribute can be set to a named color such as red, or to specific values controlling the intensity

of the red, green, and blue components. The `/size` attribute takes width and height values (usually specified as percentages of the screen) listed within parentheses.

IQML is case-insensitive, so “redbox” and “RedBox” are considered the same name.

Getting Help

Within the Inquisit Lab editor, you can use autocomplete to get hints about the various elements available for declaration and the attributes available on a given element. If you type “<” in a free area of the script, you’ll see a list of all Inquisit elements. If you position your cursor inside an element declaration and type “/”, you’ll see a list of attributes supported by the element.

If you hover the mouse above an attribute to see a tooltip showing the syntax and options for the attribute. You can also place the keyboard cursor on any attribute within an element, press the F1 key, and a help topic will appear showing even more details.

Inquisit Scripting Language (IQScript)

IQML provides a simple declarative framework for assembling elements together into an experiment or test. Although the object model is quite flexible and supports countless different paradigms, some procedures require custom calculations, conditional branching, or dynamic flow of events that adapt or change based on a participant’s performance. To support these types of events Inquisit uses a procedural scripting language called “IQScript” that enables you to define custom logic to control the behavior of the elements you’ve declared.

The syntax of IQScript is modeled after that of JavaScript, with IQScript supporting a subset of the keywords and statements supported by JavaScript. As with JavaScript, lines of code are terminated with semicolons, and blocks of code are contained within curly braces `{}`. The language supports arithmetic operators `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. There are numerous built in functions such as `sin`, `cos`, `abs` for computing sine, cosine, and absolute values. IQScript also supports `if-else` blocks for conditional code evaluation, and `while` loops for iteration. Properties of elements can be accessed using the dot `(.)` operator.

The following example shows a trial element that uses IQScript for purposes of branching. For purposes of demonstration, we show only the `/branch` attribute. Other attributes that are typically defined on a trial for presenting stimuli, measuring a response, and determining whether the response was correct are hidden.

```
<trial example>
```

```
...
```

```
/ branch = [  
    if (trial.example.correct == false) {  
        trial.example;  
    }  
]
```

```
    else if (trial.example.latency > 500) {  
        trial.responsetimewarning;  
    }  
]  
...  
</trial>
```

In the example above, the `/branch` attribute, which allows you to conditionally branch to a specific trial after the given trial is finished, is set to a code block contained within brackets `[]`. The code contains an `if` statement checking whether a correct response was made on the trial. If not, the trial branches to itself, and thus the trial is repeated. If a correct response was given, the `else if` statement evaluates whether the response latency (i.e., response time) is greater than 500 milliseconds. If so, the trial branches to another trial called “`responsetimewarning`” which could, for example, display instructions encouraging the participant to respond faster.

Like IQML, IQScript is case insensitive, so `trial.example.latency` is the same as `Trial.Example.Latency`. Programming with IQScript will be covered in more detail later.

The IQML Object Model

IQML is an object-oriented language. An Inquisit test consists of a number of objects (or elements, we'll use these terms interchangeably in this manual) that are defined using IQML, optionally augmented with IQScript, and then combined to form the logical flow of the test. This approach differs from imperative or procedural languages (e.g., C) that use lists of instructions that flow in a sequential order.

Each IQML object is designed to play a specific role, providing a high level programming interface that encapsulates its functionality and internally handles the nitty-gritty implementation details required to execute its role. Objects can be quickly declared (as elements) and configured (using attributes), and they can be combined together using attributes that reference other objects. For example, a trial element, which is responsible for presenting stimuli and recording a response, has attributes that reference stimulus elements. A block element, which is responsible for running sets of trials, can refer to both trial elements and instruction page elements. The first step in mastering IQML is thus to understand the key objects that are available, and how they relate to other objects.

TODO: Insert Object Model Diagram here

Defaults

The defaults element enables you to set both default values and global properties that apply to the entire script. There are two attributes on the defaults element that should always be defined. These two attributes are listed in the example below:

```
<defaults>
/ canvasaspectratio = (4, 3)
/ fontstyle = ("Arial", 2%)
/ screencolor = black
/ inputdevice = mouse
</defaults>
```

canvasaspectratio

The first attribute is `canvasaspectratio`. This attribute is critical if you will be running your experiment on displays of different sizes, resolutions, and shapes, which is to be expected if you are testing over the web. Whatever the particular dimensions of the display running your test, `canvasaspectratio` specifies that the canvas for presenting visual stimuli should be the largest rectangle available on the given display that has the specified width to height ratio. By constraining Inquisit to use the same aspect ratio on all displays, you can ensure that the relative layout and size of visual stimuli on each screen will be the same.

Insert images here

fontstyle

The second attribute is `fontstyle`, which specifies the default font family and font height to use for all text presented by your script. Note the font height is specified as a percentage of screen height. Although Inquisit also supports specifying sizes in pixels, using percentages for all of your size and position settings is critical for ensuring that relative layout and size of your visual stimuli are the same across different screens. If you need a different size for a particular block of text, you can override this setting using the `/fontstyle` attribute of the specific element. Later we'll show how to override the default `fontstyle`, including using bold and italics, for particular elements.

screencolor

The `screencolor` attribute specifies the background color of the screen to use throughout the experiment. By default, the background color is white.

inputdevice

Inquisit supports a variety of response modalities, including keyboard, mouse, voice, and joystick, with keyboard being the default modality. If your test will mostly be using the mouse or voice recording, you can change the default modality using this attribute. For tests that use multiple modalities, this value can be overridden in specific trials or blocks of trials.

Note that Inquisit will automatically adapt the specified modality depending on the type of device it is running on. For example, if `/inputdevice` is set to `keyboard`, when Inquisit runs on an iPad, Android or Microsoft Surface device with no keyboard attached, a response bar will be displayed at the bottom of the screen with buttons corresponding to the valid response keys. Participants can then respond by touching the buttons on the response bar.

window size

Another important attribute of the `defaults` element is `window size`, which we've added to the example below.

```
<defaults>
/ canvasaspectratio = (4, 3)
/ fontstyle = ("Arial", 2%)
/ screencolor = black
/ inputdevice = mouse
/ window size = (95%, 95%)
</defaults>
```

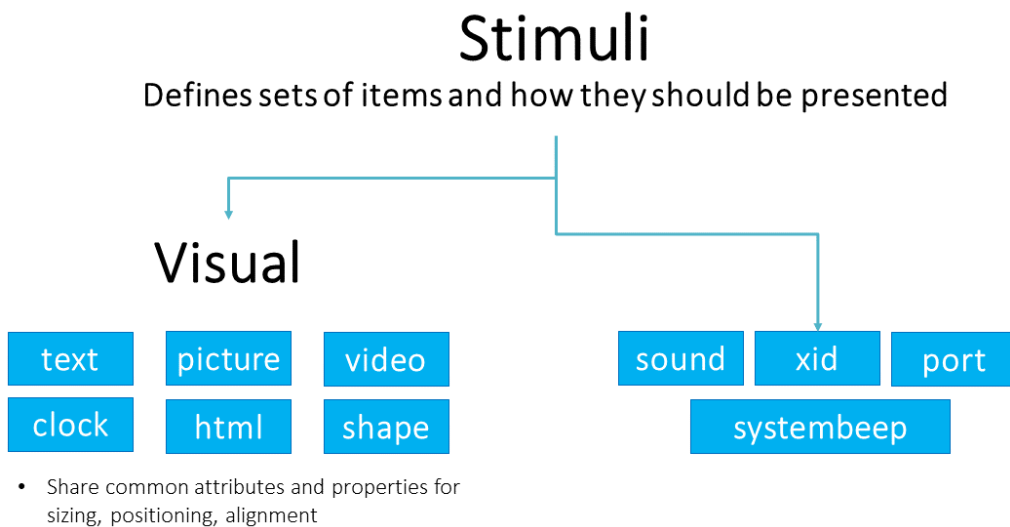
By default, Inquisit runs all tests in fullscreen mode. If `/window size` is specified, the script will be run in a window of the specified size on Windows and Mac (full screen is always used on iOS

and Android devices). This allows participants to switch to other windows or even end the test by clicking the window's close button. Depending on your participants and research requirements, these features may be advantageous or disadvantageous. Note that again we specify the size in percentages, which in this example is a percentage of the display - specifically the window will be 90% of the width and height of the screen.

Stimuli

Inquisit supports presenting a variety of different kinds of stimuli.

Inquisit Object Model - Stimuli



text

The text element enables presentation of text to the screen, and includes attributes controlling the font, position, color, alignment, and whether text should be presented as a single line or multiline with automatic word wrapping. Below is a simple example:

```
<text example>  
/ items = ("Hello World")  
/ fontstyle = ("Arial", 5%)  
/ position = (50%, 25%)  
/ txcolor = white  
/ txbgcolor = black  
</text>
```

The example presents “Hello World” as a single line centered at the point of the screen corresponding to the horizontal center (50%) and at the middle of the upper half of the screen (25%). The origin (0%, 0%) is always the upper left corner of the screen with (100%, 100%) representing the bottom right of the screen. The font is Arial, and the height of the text is 5% of the vertical height of the screen (or window if /windowsize has been specified). The text is printed in white against a black background.

A text element can be used to present a single item of text, or it can select from a set of items. Selection can be configured to be random, sequential, synchronized with the selection of another stimulus or list, or determined by a custom expression.

picture

The picture element enables presentation of image files in a variety of formats, including JPG, PNG, GIF, and BMP.

```
<picture example>  
/ items = (“helloworld.jpg”)  
/ position = (75%, 75%)  
/ size = (20%, 20%)  
</picture>
```

With the picture element, the items specify the file name of the image to display. In the above example, the picture presents an image file named “helloworld.jpg” which is assumed to be located in the same folder as the script. The image is displayed in the bottom right quadrant of the screen (or window). The image is sized to fit within a bounding rectangle representing 20% of the width and height of the screen. The aspect ratio of the image is always preserved.

shape

The shape element presents circles, ellipses, rectangles, and triangles.

```
<shape example>  
/ shape = triangle  
/ position = (25%, 25%)  
/ size = (30%, 15%)  
/ color = red  
</shape>
```

In the above example, a red isosceles triangle is presented in the upper left quadrant of the screen or window. The base of the triangle is 30% of the width of the screen, and the height is 15% of the height of the screen. The /shape attribute controls which shape to present. Options are rectangle, circle, and triangle.

video

The video element plays videos in a variety of formats, although MP4 is the recommended format for presenting videos that will work on different platforms.

```
<video example>  
/ items = ("helloworld.mp4")  
/ position = (75%, 50%)  
/ size = (25%, 25%)  
/ loop = true  
</video>
```

In the above example, a video file named helloworld.mp4 is presented vertically centered on the right half of the screen. The file is assumed to be located in the same folder of the script. The video is sized to fit in a bounding rectangle equal to 25% of the width and height of the screen or window. The aspect ratio of the video is always preserved - i.e., the video is not stretched to the exact size of the rectangle. The /loop attribute specifies that the video should continuously replay whenever it finishes.

html

The html element presents html pages on the screen. Although the element supports older versions of html, the recommended format is HTML 5 for consistent presentation across different platforms.

```
<html example>  
/ items = ("helloworld.htm")  
/ position = (50%, 50%)  
/ size = (90%, 90%)  
/ showborders = false  
/ showscrollbars = true  
</html>
```

In the above example, the element presents an html file called helloworld.htm. The page is centered on the screen and sized to 90% of the height and width of the screen, leaving a 5% margin on all sides. The page is presented without any borders (/showborders = false), and scrollbars will be displayed if the page extends beyond the bounding rectangle (90%, 90%).

clock

For timed tasks, a clock can be presented on the screen in stopwatch mode to indicate how long a participant has spent on a given response, or in timer mode to indicate how long the participant has left to respond in cases where a time limit is imposed.

```
<clock example>
```

```
/ mode = timer  
/ format = ("ss:zzz")  
/ timeout = 10000  
/ position = (90%, 90%)  
/ size = (20%, 20%)  
/ txcolor = white  
/ txbgcolor = black  
</clock>
```

In the above example, the clock functions as a timer counting down from 10000 milliseconds, which is the value specified by the `/timeout` attribute. The `/format` attribute specifies that the clock should display the remaining time in seconds and milliseconds. The clock is positioned in the lower right corner of the screen, with the numbers displayed in white against a black background.

sound

For tasks involving an audio component, the sound element enables you to play wav files. Inquisit uses high performance APIs to present sound with as little lag as can be achieved on a given device.

```
<sound example>  
/ items = ("helloworld.wav")  
/ playthrough = true  
</sound>
```

In the above example, the sound element presents a wav file called `helloworld.wav`. The `/playthrough` attribute specifies that the wav file should be played to completion on a given trial, even if the participant responds before it has finished.

Positioning and Layout of Visual Stimuli

Visual stimuli in Inquisit all share a set of attributes that give you more fine-tuned control over the positioning and layout on the screen. We've already covered the `/position` attribute, which specifies the point on the screen where a stimulus should be displayed. By default, stimuli are horizontally and vertically centered around that point. Using the `/halign` and `/valign` attributes, you can override the default alignment.

The `/halign` attribute has three possible values:

- `center`: Stimuli are horizontally centered around the position point (default)
- `left`: Stimuli are left-aligned to the position point
- `right`: Stimuli are right-aligned to the position point.

The `/valign` attribute also has three options:

- `center`: Stimuli are vertically centered around the position point (default)

- top: Stimuli are top-aligned to the position point
- bottom: Stimuli are bottom-aligned to the position point.

Using the alignment attributes, you can layout stimuli in a variety of ways. For example, if two stimuli are meant to be immediately adjacent to each other, they could be left- and right-aligned on the same position point, or top- and bottom-aligned around a point. If a stimulus should be displayed in the upper left corner of the screen, it could be top- and left-aligned around a position point of (0, 0).

Erasing Stimuli

When stimuli are presented on a given trial, by default they are erased from the screen when the participant responds. Similarly, sounds are stopped when a response is made. Using the `/erase` attribute, you can control whether and how a stimulus should be erased.

By setting `/erase = false`, a stimulus will not be erased. For example, consider the following text element:

```
<text reminder>
/ items = ("Remember to respond as quickly as you can!")
/ fontstyle = ("Arial", 3%)
/ position = (50%, 25%)
/ txcolor = white
/ txbgcolor = black
/ erase = false
</text>
```

This presents a reminder to participants to respond as quickly as they can. This reminder will remain on the screen for subsequent trials unless one of those trials happens to overwrite it with another stimulus.

The `/erase` attribute also allows you to customize the color used to erase a visual stimulus. By setting `/erase = true(blue)`, the visual stimulus is painted over with blue when it is erased.

item

In the examples above, the actual stimulus items to be presented (e.g., words, images, videos, etc.) are defined in the `/items` attribute. In many tests, however, different stimulus elements may refer to the same set of items. For example, consider a procedure in which a set of images could be presented in one of two positions on the screen. You would define two `<picture>` elements, one for each position, that would have the same set of images as in the example below.

```
<picture leftposition>
/ items = ("1.jpg", "2.jpg", "3.jpg", "4.jpg", "5.jpg", "6.jpg")
```

```
/ size = (20%, 20%)  
/ position = (25%, 50%)  
</picture>
```

```
<picture rightposition>  
/ items = ("1.jpg", "2.jpg", "3.jpg", "4.jpg", "5.jpg", "6.jpg")  
/ size = (20%, 20%)  
/ position = (75%, 50%)  
</picture>
```

While this is perfectly valid, it can make a script cumbersome to update. For example, if we decide to replace an image with a different one, we have to remember to replace it in both of the <picture> elements.

Alternatively, we can define the items once using the <item> element, which can then be shared by multiple stimulus elements. The syntax of the <item> element is simple. Each item is simply listed using the item's ordinal position as an attribute. For example, consider the following <item> definition:

```
<item images>  
/1="1.jpg"  
/2="2.jpg"  
/3="3.jpg"  
/4="4.jpg"  
/5="5.jpg"  
/6="6.jpg"  
</item>
```

The <item> element must have a name, in this case "images". The item includes the same 6 image files (1.jpg through 6.jpg) as before, each defined by a corresponding numeric attribute (/1 through /6). Each file name must be enclosed in quotes.

Having declared this <item> element, we can now refer directly to it from the /items command of a given stimulus. For example:

```
<picture leftposition>  
/ items = images  
/ size = (20%, 20%)  
/ position = (25%, 50%)  
</picture>
```

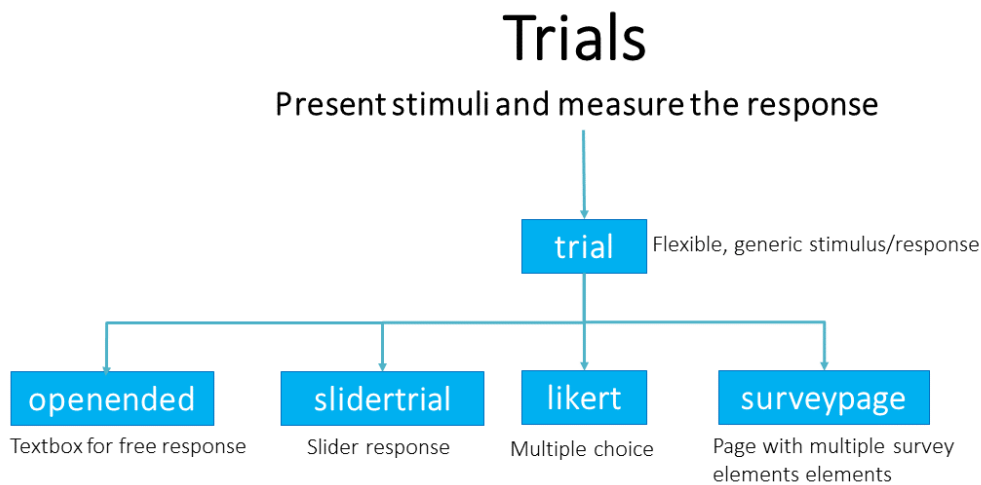
```
<picture rightposition>  
/ items = images  
/ size = (20%, 20%)
```

```
/ position = (75%, 50%)  
</picture>
```

Both <picture> elements now draw from the same set of items, defined once in the <item> element.

Trials

Inquisit Object Model -Trials



The trial element is at the core of the Inquisit language. Trials can play a number of roles, but their primary role is to present stimuli to the screen according to a given schedule, and to measure a participant's response to those stimuli, whether the response is made by pressing a key, clicking a mouse, touching a screen, maneuvering a joystick, or pressing a button on a response box. The trial element has numerous attributes for handling various types of tasks. Think of it as a Swiss Army knife for cognitive tests.

Presenting Stimuli

One of the most powerful features of the trial is the ability to present precisely timed stimuli to participants. The trial element can present any of the stimuli covered above. It can synchronize multiple stimuli to be presented at the same time, and it can present very rapid sequences of stimuli.

To understand how stimulus presentation works in Inquisit, it's helpful to understand the basics of how the ubiquitous LED screens on all of our computerized devices update the screen. Computerized displays do not continuously update the screen. Instead, they draw to the screen using discrete frames, each of which is separated by a constant time interval. In this way, they

are analogous to flip book animations, where each page contains a separate drawing, and when you flip through the pages quickly, the drawings blend together to create a continuous animation. LED screens (and the CRTs that came before them) work much the same way.

A typical LED screen repaints itself once every 16.667 milliseconds, or at a rate of 60 times per second, or 60hz. This rate is referred to variably as the “refresh rate”, “frame rate”, or “vertical retrace rate” of the monitor. Many screens support faster modes, such as 70 or 75hz. Some high-end monitors designed for gaming or research can refresh at 200hz (once every 5 ms).

An important implication of this design is that the refresh rate determines the shortest possible duration a stimulus can be presented. For example, the shortest duration that a visual stimulus can be presented on a 60hz monitor is 16.667 milliseconds, which is the duration of a single frame. To achieve this, a stimulus must be painted to the screen on one frame and then erased from the screen on the very next frame, which happens 16.667 milliseconds later. The 60hz monitor can not erase the stimulus any more quickly than that.

The concept of a refresh rate is built into the Inquisit trial element. Specifically, the trial allows you to schedule stimuli at successive frames, with frame 1 being the first on a given trial. All stimuli, including non-visual stimuli are synchronized with the refresh rate of the monitor. This gives you precise control over the timing of presentations within the physical constraints of the display hardware.

The attribute for specifying the stimulus presentation sequence is /stimulusframes. For each frame starting at 1, one or more stimuli can be scheduled for presentation using the following syntax:

```
<trial auditoryactiontime>  
 / stimulusframes = [1=ready,reminder; 30=tone]  
</trial>
```

In the above example, two stimuli named “ready” and “reminder” are presented on the first frame. On the 30th frame (500 ms on a 60hz monitor), a tone is played after which stimulus presentation is complete, and the trial waits for a yet-to-be-defined response.

The example below shows a subliminal priming procedure:

```
<trial priming_pos_pos>  
 / stimulusframes = [1=fixation; 30=forwardmask;31=prime;32=backwardmask;33=target]  
</trial>
```

A fixation stimulus is presented on the first frame. At the 30th frame, a forward mask is presented, which is overwritten on the next frame by a prime, which is overwritten on the next frame by a backward mask. The prime stimulus is thus sandwiched in time between the forward

and backward mask. On the 33rd frame, a target stimulus is presented which remains on the screen until a response is given.

As a convenience, the trial element also allows you to schedule stimuli in terms of milliseconds instead of frames using the /stimulustimes command. Using /stimulustimes, the masked priming example above could be specified as follows:

```
<trial priming_pos_pos>  
/ stimulustimes = [0=fixation; 500=forwardmask; 517=positiveprime; 533=backwardmask;  
550=positivetarget]  
</trial>
```

If stimuli are specified at a time that falls between the boundaries of a refresh interval, the actual presentation time is rounded up or down to the nearest frame boundary. If, for example, two stimuli are presented at 10 and 15ms on a 60hz monitor, both presentation times would be rounded up to 16.7 ms - i.e., the second frame - and would thus be presented simultaneously.

Measuring Responses

The trial element includes a number of attributes that specify the response modality, which responses are recognized as valid, and which responses are considered correct. Continuing with the masked priming example, we'll add three new attributes to the trial to control these parameters.

```
<trial priming_pos_pos>  
/ stimulustimes = [0=fixation; 500=forwardmask; 517=positiveprime; 533=backwardmask;  
550=positivetarget]  
/ inputdevice = keyboard  
/ validresponse = ("e", "i")  
/ correctresponse = ("e")  
</trial>
```

inputdevice

The /inputdevice attribute allows you to specify the response modality used for the trial. If this attribute is not specified, the trial will use the /inputdevice value specified in the <defaults> element. If /inputdevice is not specified on the <defaults> element, it falls back to the keyboard as the default.

validresponse

The /validresponse attribute specifies which responses are considered meaningful or valid responses on the given trial. The responses are listed within parentheses and can include any number of responses. The values that can be listed here depend on the /inputdevice. Inquisit supports multiple input devices, including keyboards, mice, touchscreens, game controllers,

voice key, speech, Cedrus devices, custom response boxes, and even gaze points from an eye tracker.

With `/inputdevice=keyboard`, you can specify the character (in quotes) corresponding to the key on the keyboard. The example above shows a 2-choice response paradigm, where the two valid responses are the “e” and “i” key. Once either of these responses is made, the trial is complete and the next trial is run.

For keys that don't have a character value (e.g., Delete, Shift), you can alternatively specify the numeric scan code of the keys in `/validresponse`. To determine the scan code of a key, open Inquisit Lab and select the “Keyboard scancodes...” command from the Tools menu. You can then press any key on the keyboard, and it will display its scancode.

With `/inputdevice=mouse`, you can specify the names of any visual stimuli presented on that trial. If the participant clicks on any of the listed stimuli, it is considered a valid response for the trial. You can also use the mouse as a simple buttonbox where clicking the left or right button is considered a valid response.

Note that if `/inputdevice` is set to keyboard or mouse, Inquisit automatically adapts these response modes to run on touchscreen devices that don't have a separate keyboard or mouse plugged in. For keyboard input, Inquisit shows a response bar at the bottom of the touchscreen with valid keys represented as buttons. Participants can thus touch a button instead of pressing a key. For mouse input, participants can touch target stimuli on the screen instead of mouse-clicking them.

correctresponse

The `/correctresponse` attribute specifies which (if any) responses should be scored as correct. The syntax is similar to the `/validresponse` command, with correct responses listed within parentheses. Here, too, the values that can be listed depend on `/inputdevice`. In the example, the “e” key is considered the correct response of the two choices.

Measuring Response Time

By default, a trial starts listening for responses as soon as its last stimulus frame is presented. Responses made before all frames have been presented are simply ignored. The trial automatically measures the response time, or “latency” as it's referred to within the Inquisit language, starting from the point at which the last frame is presented. Response times are measured in milliseconds. In the above example, if the participant presses the “i” key exactly 435 milliseconds after the `positivetarget` stimulus is presented, the latency is recorded as 435.

Specialized Trials

Inquisit offers a number of specialized trials to support different forms of responding, such as moving a slider, entering text, or selecting from a fixed number of choices. These specialized

trials can be thought of as subclasses of the trial element, and as such, they can be defined and referenced and used anywhere in the script as trials.

Openended

The openended element is a specialized trial designed for text entry. The trial presents a text box where participants can type responses, and a button to submit their response. Responses can be constrained by length or format such as valid numbers and dates.

```
<openended example>  
/ stimulusframes = [1=questions]  
/ multiline = true  
/ required = true  
/ position = (50%, 70%)  
/ size = (30%, 20%)  
</openended>
```

In the above example, the openended trial presents a stimulus called "questions". It then presents a multiline textbox at the coordinates corresponding to the specified size and position. The /required attribute is set to true, indicating that participants must enter at least one character of text in order to advance.

Slidertrial

The slidertrial element enables participants to select a value by moving an indicator along a slider scale.

```
<slidertrial example>  
/ stimulusframes = [1=questions]  
/ orientation = horizontal  
/ range = (-50, 50)  
/ labels = ("cold", "hot")  
/ showticks = true  
</slidertrial>
```

The above slidertrial presents a stimulus called "questions" and a horizontal slider control. The numeric range of the slider is from -50 to 50, left to right, with 0 as the center point. Two labels are presented, "cold" on the left end of the scale and "hot" on the right end. Ticks are displayed along the slider control in increments of 1 unit.

Likert

On likert trials, participants respond by selecting one of an array of options.

```
<likert example>
```

```
/ stimulusframes = [1=questions]
/ anchors = [1="Never"; 3="Sometimes"; 5="Always"]
/ numpoints = 5
/ buttonvalues = [1="-2"; 2="-1"; 3="0"; 4="1"; 5="2"]
/ position = (50%, 75%)
</likert>
```

The above likert trial presents a single stimulus called “questions” along with a 5-point likert scale on the lower half of the screen with values ranging from 1 to 5. Text anchors “Never”, “Sometimes”, and “Always” appear at the first, third, and fifth positions on the scale. Scales are presented as a line of buttons which participants can click to respond and move to the next trial.

Blocks

The block element is responsible for running sets of trials. The block can be configured to run trials representing the same condition of an experiment, or it can support mixed block designs in which trials representing different conditions are run. The order of trials can be a fixed sequence or random, a combination of the two, or random with constraints such as avoiding runs of the same type of trial or repeating trials on which an error is made. Blocks can present instructions pages to be displayed before the trials begin or after they are finished.

A typical experiment may run a set of practice blocks that progressively instruct participants how to perform a task. For example, a two-choice priming task might start with a practice block in which only targets are presented in order to familiarize them with the basic categorization task. The practice block could be configured to repeat itself until the participant's performance meets criterion. The next practice block would introduce the primes, instructing participants to ignore them and continue responding to the targets as before. Once performance meets criterion, the task can move on to test blocks in which priming effects are measured.

The example below continues the categorical priming task we started above. The block implements a mixed-block design that runs trials representing all four combinations of primes and targets, including congruent trials (priming_pos_pos/priming_neg_neg) and incongruent trials (priming_pos_neg, priming_neg_pos). A total of 40 trials are run in random order. If possible, Inquisit will ensure that each type of trial is run the same number of times. In this case, since we are running 20 trials that are randomly selected from 4 types of trials, each type will be run exactly 5 times in the block.

```
<block evaluative_priming_test>
/ preinstructions = (page1, page2)
/ trials = [1-20 = random(priming_pos_pos, priming_neg_neg, priming_pos_neg,
priming_neg_pos)]
/ postinstructions = (rest)
</block>
```

preinstructions

The `/preinstructions` attribute lists a set of instruction pages to present at the beginning of the block before the trials are run. The pages are named “page1” and “page2” and are presented in the given order (we'll cover how to define instruction pages later). Instruction pages can consist of paragraphs of plain text or html pages consisting of richly formatted text, images, etc.

trials

The `/trials` attribute specifies the set of trials to run in the block. Not only does the trial specify how many trials to run, but also which trials and in what order. Sequences of trials such as “1-20” as in the above example can be assigned to a single trial or to a set of randomly selected trials (as above). Alternatively, if the order is meant to be fixed, each trial can be specified individually, for example:

```
/trials = [1=priming_pos_pos;2=priming_neg_neg;3=priming_pos_neg;4=priming_neg_pos,...]
```

Or more conveniently:

```
/trials = [1, 5, 9, 13, 17 = priming_pos_pos;  
2, 6, 10, 14, 18 = priming_neg_neg;  
3, 7, 11, 15, 19 = priming_pos_neg;  
4, 8, 12, 16, 20 = priming_neg_pos]
```

Note that the entire trial sequence is contained within square brackets “[]”, and that each subset in which trial numbers are assigned to trials must be separated by a semi-colon “;”.

postinstructions

The `/postinstructions` attribute specifies any instruction pages to be presented after all trials have been run. The syntax is the same as `/preinstructions`.

Expt (experiment)

The `expt` element allows control of sequences of blocks. The order of blocks can be a fixed sequence or random, a combination of the two, or random with constraints such as avoiding runs of the same type of block. Expts can present instructions pages to be displayed before the blocks start and after they are finished.

The example below continues with the evaluative priming task. A total of seven blocks are run. The first block allows participants to practice classifying target stimuli only. The second block gives participants a chance to practice with prime stimuli. The remaining blocks run test trials of the evaluative priming task.

```
<expt evaluativepriming>
```

```
/ preinstructions = (intro)
/ blocks = [1=targets_practice;2=evaluative_priming_practice; 3-7=evaluative_priming_test]
/ postinstructions = (thankyou)
</expt>
```

preinstructions

The `/preinstructions` attribute lists a set of instruction pages to be presented at the beginning of the experiment before any blocks are run. An instruction page called “intro” is presented in the example.

blocks

The `/blocks` attribute specifies the set of blocks to run in the experiment. Not only does it specify how many blocks to run, but also which blocks and in what order. Sequences of blocks such as “3-7” in the above example can be assigned to a single block (as above) or to a set of randomly selected blocks. Alternatively, if the order is meant to be fixed, each block can be specified individually, as in the first and second practice blocks.

The syntax of `/blocks` is the same as for `/trials`. The entire block sequence is contained within square brackets “[]”, and each subset in which block numbers are assigned to blocks must be separated by a semi-colon “;”.

postinstructions

The `/postinstructions` attribute specifies any instruction pages to be presented at the end of the experiment. In the above example, a “thankyou” page is displayed which informs the participant the test is over and thanks them for participating. The syntax is the same as `/preinstructions`.

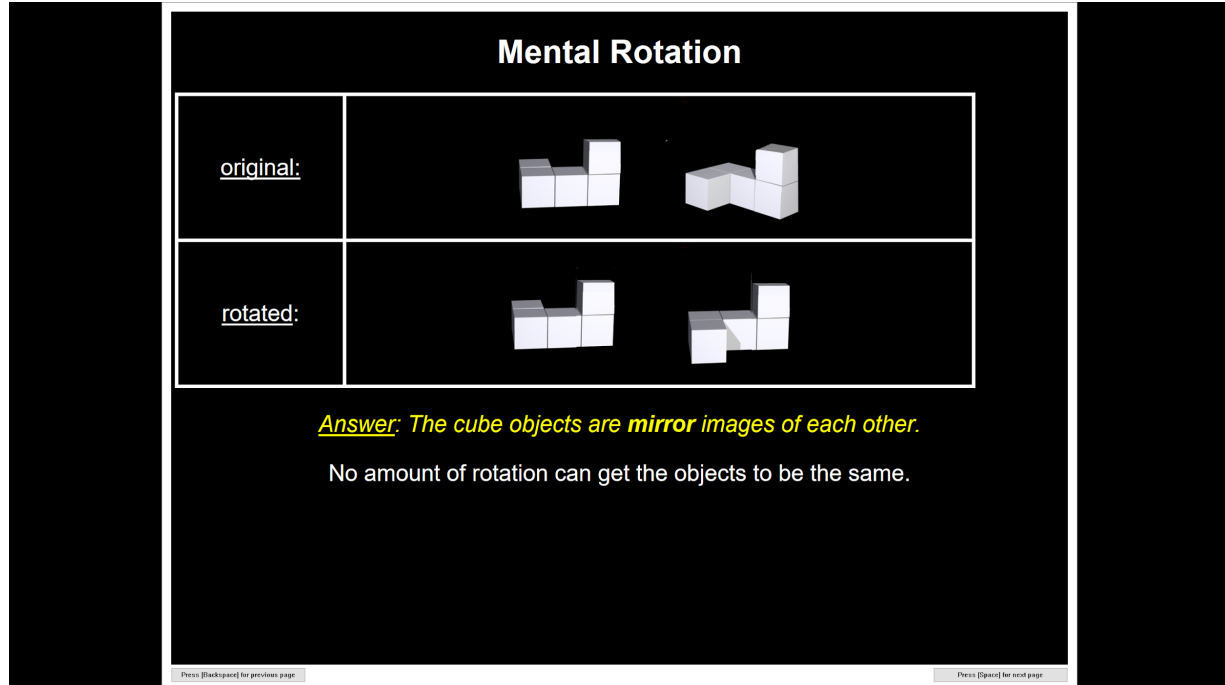
Presenting Instruction Pages

The IQML language includes a set of elements designed for easy presentation of task instructions pages. The pages are presented with built-in navigation buttons for moving back and forth between sequences of instructions. No data are recorded while participants browse the instructions.

instruct

The `<instruct>` element provides central control over how all task instruction pages are displayed on the screen during the test. The element controls the size of instruction pages, the font used for text pages and navigation buttons, whether a back button is displayed for returning to previous pages, and whether participants navigate using the keyboard or mouse.

Figure 3. An example of an HTML instruction page in Inquisit.



The following example sets the font, text and background color, and size of instruction pages.

```
<instruct>
/ fontstyle = ("Arial", 5%, true)
/ txcolor = white
/ windowsize = (90%, 90%)
/ screencolor = black
</instruct>
```

page

The page element defines a single page of plain text instructions. The text is automatically wrapped to fit into the instructions window. Line breaks can be inserted to separated blocks of text. The entire page of text is presented in the color and font specified in the /txcolor and /fontstyle attributes of the <instruct> element. To present text with different colors, fonts, and/or styles, you can format your instructions using HTML and present them using <htmlpage> element.

An example of the element is given here:

```
<page intro>
Welcome to the Evaluative Priming Task.
```

The approximate duration of this task is 10 minutes.

When you are ready to start, click the Continue button.

```
</page>
```

Unlike other elements, `<page>` does not have any attributes. Rather, the text that should be displayed on the page is simply entered between the open and close tags. Line breaks appearing within the tags will also appear when the page is displayed. You can also insert line breaks using the “~” character.

Finally, with Inquisit 6, page text can also be formatted using a simple, limited subset of HTML.

htmlpage

The `htmlpage` element supports presenting instruction pages formatted with HTML. The element allows you to utilize the full power of HTML to present rich instructions with different colors and styles, tables, media such as images and video, and anything else you might find on a web page.

The `<htmlpage>` element has a single attribute, the `/file` attribute which is set to the file name of the HTML page to present. For example, the following `<htmlpage>` displays a file called “intro.htm”.

```
<htmlpage>  
/ file = “intro.htm”  
</htmlpage>
```

Data element

The data element specifies which data should be recorded at the end of each trial. The data file thus contains a single line/row of data for every trial that is run. We refer to this data file as the “raw” data file. There can be only one data element defined in a given script.

By default, the raw data file is saved into a folder called ‘data’ that is located in the same folder as the script. The file name will be that of the script with “_raw” appended to it and the “iqdat” file extension. For example, if the script file name is “racismiat.iqx”, the raw data file will be named “racismiat_raw.iqdat”.

The contents of this file are defined by the data element, as in the example:

```
<data>  
/ columns = (date, time, subject, group, blockcode, trialcode, trialnum, response, latency,  
correct, stimulusitem1, stimulusitem2, stimulusitem3, stimulusitem4, stimulusitem5)  
/ separatefiles = true
```


</data>

columns

The columns attribute defines the fields or columns of data that are saved. At a minimum, the raw data file should record the columns listed in the example above. Starting from left to right, the columns in the example are defined as follows:

date	The date the session was started
time	The time the session was started
subject	A unique identifier of the participant
group	The group number used in this session (for tests involving between-subject variables)
blockcode	The name of the current block
trialcode	The name of the current trial
trialnum	The ordinal number of the trial
response	The response made by the participant
latency	The response time in milliseconds
correct	Whether or not the response was correct
stimulusitem1 through stimulusitem5	The particular item presented on the trial for the first, second, third, fourth, and fifth stimulus.

The stimulusitem columns deserve a bit more explanation as they work somewhat differently than the rest. Consider the masked priming trial as an example:

```
<trial maskedpriming>
/ stimulustimes = [0=fixation; 500=forwardmask; 517=positiveprime; 533=backwardmask;
550=positivetarget]
/ inputdevice = keyboard
/ validresponse = ("e", "i")
/ correctresponse = ("e")
</trial>
```

Trial presents 5 different stimuli - a fixation, forward mask, a positive prime, backward mask, and a positive target. The stimulusitem1 column shows the current item of the first stimulus defined in the /stimulusframes, the fixation stimulus, which might be a plus sign "+" on each trial. The stimulusitem2 column records the second stimulus item in the sequence, which is the

current forward mask, and which might be a row of Xs "XXXXXXXXXX" on each trial. The `stimulusitem3` column records the current item of the third stimulus in the sequence, the prime, which might be randomly selected from list of evaluatively positive words, for example "LOVE" on one trial, "WONDERFUL" on the next trial, "HAPPY" on the next trial, and so forth. `Stimulusitem4` records the current item of the backward mask, which might be another row of Xs "XXXXXXXXXX". `Stimulusitem5` records the current item of the target stimulus, which might be randomly selected from a list of evaluatively positive words, e.g., "SMILE" on one trial, "LAUGHTER" on the next, "BEAUTIFUL" on the following, and so forth.

For stimuli that select from a set of items, the `stimulusitem` row thus captures the particular item presented. By recording these data, it would be possible to do item level analysis of particular primes and targets.

separatefiles

Inquisit Lab will allow you to save data from different sessions in separate files or appended into a single data file. Inquisit defaults to saving data to separate files because this minimizes data loss due to file conflicts. Although a single file might be a convenient format for data analysis, we strongly recommend against using this option to avoid data loss. For example, if the single file is accidentally deleted or corrupted, then all data from all subjects is lost. If separate files are used, accidentally deleting a single file means that only a single subject's data is lost. If a data file is opened and locked by another application, data from a session might not be saved when using a single file, but with separate files, a new file is created for each session so data recording can proceed.

When you are ready to analyze the data, you can use Inquisit's "Merge Data Files..." command available on the File menu to combine the separate files into a single file.

Summary Data

The `<summarydata>` element allows you to record a single line of summary scores for each participant at the end of the test. Typically, `<summarydata>` includes the final metrics for the test, whatever they happen to be, along with the group id, subject id, date and time the session started, and whether the entire test was completed. When the `<summarydata>` from multiple participants are combined, the result is a table of data that is conveniently formatted for statistical analysis.

While the `<data>` element includes built-in columns (e.g., response, latency, `stimulusitem`), the `<summarydata>` element only supports recording the property values of various elements as columns (more on element properties later).

In the example below, the `<summarydata>` element records a standard set of summary variables, including the date and time, subject id, group id, total time spent on the test, the computer platform (type of device used), and a value indicating whether the entire test was

completed. In addition, a set of custom expressions called “da”, “db”, and “dc” that return the final scores for the test. The last expression, “percentcorrect”, records the overall percent correct for the task and can be used as a quick screener for participants who weren't following instructions.

```
<summarydata>  
/ columns = (script.startdate, script.starttime, script.subjectid,  
script.groupid, script.elapsedtime, computer.platform, values.completed,  
expressions.da, expressions.db, expressions.d, expressions.percentcorrect)  
/ separatefiles = true  
</summarydata>
```

columns

The columns attribute defines which properties are saved to the summary data file. Unlike the data element for raw data, the summary data element does not have any built-in columns. Only properties are supported. So, for example, instead of specifying “date”, you would specify “script.startdate”.

separatefiles

The separatefiles attribute works the same as in the data element. If true (default), the summary data from each session is saved to separate files. If false, Inquisit saves data to a single data file. For reasons spelled out above, you should always use the default setting so that separate files are used.

Programming a Simple Test with IQML

IQML provides a simple, declarative language for specifying the various components of a test and connecting them together. While most tests use IQScript in combination with IQML, it is certainly possible to program simple tests using only IQML. Having reviewed the language, we'll now put the concepts together to program a working test - an Emotional Dot Probe.

Emotional Dot Probe

The Emotional Dot Probe we'll program uses the following procedure. Pairs of negative and positive words are presented on the screen, one above the other, with negative and positive words randomly varying between upper and lower positions. The words are then replaced on screen with a dot appearing randomly in the upper or lower position, and participants must press one of two response keys to indicate which position it's in. Accuracy and latency of responses are recorded. The test measures the extent to which negative or positive stimuli draw the participants' attention, causing them to respond more quickly and accurately when the dot appears in the same position as a negative or positive word.

Defining the Stimuli

When creating a new script, you can start with any element. We generally find it easier to start by defining the core stimuli to be presented. In this case, the stimuli consists of two sets of words, one positively valenced and the other negatively valenced, which are defined below:

```
<item pleasant>  
/1 = "Marvelous"  
/2 = "Superb"  
/3 = "Pleasure"  
/4 = "Beautiful"  
/5 = "Joyful"  
/6 = "Glorious"  
/7 = "Lovely"  
/8 = "Wonderful"  
</item>
```

```
<item unpleasant>  
/1 = "Tragic"  
/2 = "Horrible"  
/3 = "Agony"  
/4 = "Painful"  
/5 = "Terrible"  
/6 = "Awful"
```

```
/7 = "Humiliate"  
/8 = "Nasty"  
</item>
```

Next, we'll define the text elements that refer to these item sets. Recall that both item categories can be presented in the upper or lower position, which requires us to define $2 \times 2 = 4$ different `<text>` elements. The positive `<text>` elements are defined as follows:

```
<text pleasanttop>  
/ items = pleasant  
/ position = (50%, 45%)  
</text>
```

```
<text pleasantbottom>  
/ items = pleasant  
/ position = (50%, 55%)  
</text>
```

Both refer to the "pleasant" item set, presenting them at the horizontal center (50%) of the screen. The "pleasanttop" stimuli, however, are presented at 45% of the height of the screen (upper position) and the "pleasantbottom" stimuli are presented at 55% of the height of the screen (lower position).

Next we'll define the negative stimuli:

```
<text unpleasanttop>  
/ items = unpleasant  
/ position = (50%, 45%)  
</text>
```

```
<text unpleasantbottom>  
/ items = unpleasant  
/ position = (50%, 55%)  
</text>
```

Both elements now refer to the "unpleasant" item set, presenting those words in the upper and lower positions respectively.

The Emotional Dot Probe must also present a dot in either of the two positions. The dot stimuli can be defined using the `<shape>` element as follows:

```
<shape probetop>  
/ shape = circle  
/ size = (.66%, 1%)
```

```
/ color = grey  
/ position = (50%, 45%)  
</shape>
```

```
<shape probebottom>  
/ shape = circle  
/ size = (.66%, 1%)  
/ color = grey  
/ position = (50%, 55%)  
</shape>
```

Both <shape> elements present a grey circle, the height of which is .66% of the screen width and 1% of the screen height. (We use different percentages because most screens are wider than they are tall). Both dots are positioned in the same spot as the upper and lower word stimuli we previously defined.

During the practice phase of our dot probe, we want to show error feedback to participants in the form of an error message. The error message is defined as follows, which presents the word "error" in red:

```
<text errormessage>  
/ items = ("error")  
/ txcolor = red  
</text>
```

Finally, we'll define a fixation point that appears in the center of the screen between the upper and lower stimuli so that participants can be instructed not to focus attention on the upper or lower positions until the dot is presented. The fixation point is quite simple:

```
<text fixation>  
/ items = ("+")  
</text>
```

The fixation uses the default positioning for stimuli, which is in the center of the screen. It also uses the default font and color, which we'll define below.

Defining the Trials

The trial elements are responsible for presenting the dot probe stimuli and measuring participants' responses. A given trial can present the positive word in the upper or lower position (with negative words in the corresponding position), representing two conditions. The trial can also present the dot probe in the upper or lower position, representing another two conditions. We'll thus need $2 \times 2 = 4$ different trial elements to capture the combination of conditions. The

trial for the first combination with both the positive and dot probe in the upper position is defined below:

```
<trial pleasanttop>  
/ stimulustimes = [0=fixation; 500=clearscreen, pleasanttop, unpleasantbottom;  
1500=clearscreen, probetop]  
/ validresponse = ("e", "i")  
/ correctresponse = ("i")  
/ posttrialpause = 500  
</trial>
```

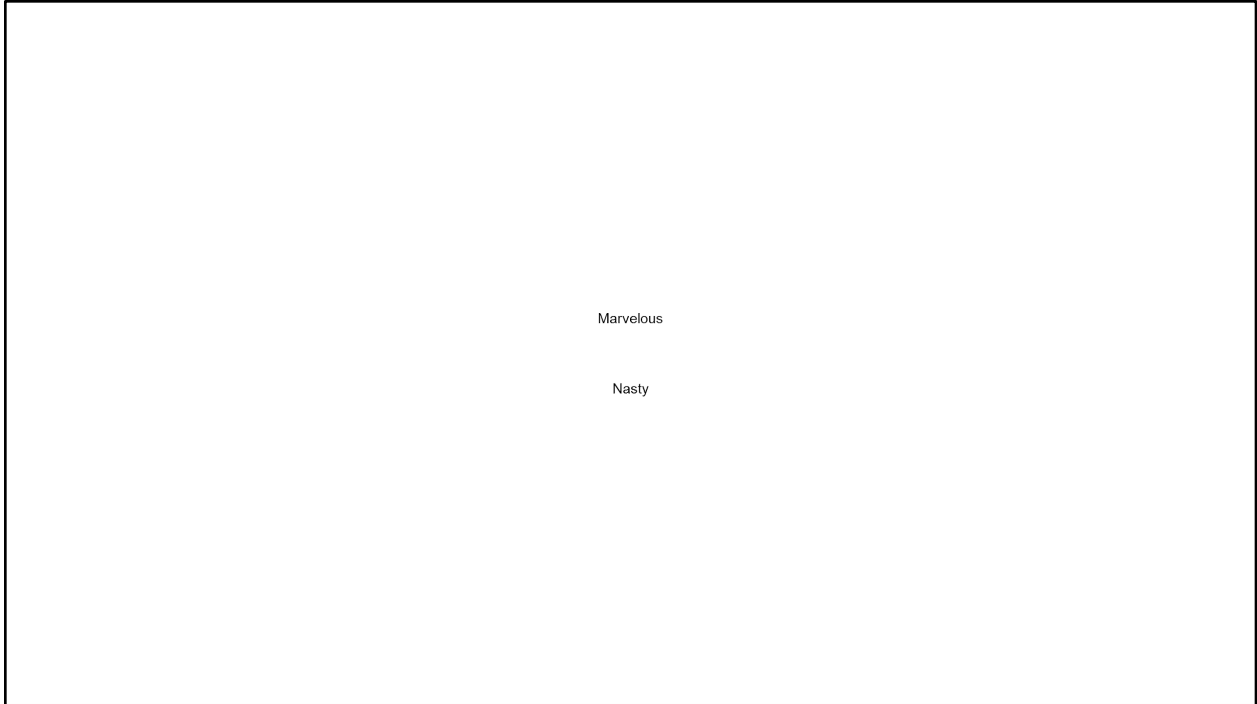
First let's look at the /stimulustimes attribute. At 0 milliseconds (the start of the trial), the fixation point is presented on the screen. 500 ms later, a special built-in stimulus called "clearscreen" is presented along with the pleasanttop and unpleasantbottom stimuli. The clearscreen stimulus simply paints the entire screen to the background color, which erases our fixation point. At the same time the screen is erased, both the negative and positive words are presented. These remain on screen for 1000ms, before they are overwritten at the 1500 millisecond mark by another clearscreen accompanied by "probetop", the dot probe shape in the upper position.

Immediately after the last stimulus ("probetop") is presented, Inquisit begins polling for a response from the participant (responses made prior to the dot probe presentation are simply ignored). The /validresponse attribute defines two possible responses for the trial, the "e" and "i" keys, indicating that the probe was in the lower or upper position respectively. All other key presses are ignored. The /correctresponse attribute defines the "i" key as correct for this trial, given that the probe is presented on top.

The last attribute /posttrialpause specifies that at the end of the trial, Inquisit will pause for 500 ms before advancing to the next trials. This defines our inter-trial interval, or ITI.

The next 3 trial definitions are similar, varying only in the positions of the stimuli and which response is defined as correct. Note that on trials where the probe is presented in the lower position, the /correctresponse is "e" instead of "i".

The following figure captures the screen at the 500 millisecond time point when both the positive and negative words are displayed on the screen.



```
<trial pleasantbottom>  
/ stimulustimes = [0=fixation; 500=clearscreen, pleasantbottom, unpleasanttop;  
1500=clearscreen, probebottom]  
/ validresponse = ("e", "i")  
/ correctresponse = ("e")  
/ posttrialpause = 500  
</trial>
```

```
<trial unpleasantbottom>  
/ stimulustimes = [0=fixation; 500=clearscreen, unpleasantbottom, pleasanttop;  
1500=clearscreen, probebottom]  
/ validresponse = ("e", "i")  
/ correctresponse = ("e")  
/ posttrialpause = 500  
</trial>
```

```
<trial unpleasanttop>  
/ stimulustimes = [0=fixation; 500=clearscreen, pleasantbottom, unpleasanttop;  
1500=clearscreen, probetop]  
/ validresponse = ("e", "i")  
/ correctresponse = ("i")  
/ posttrialpause = 500  
</trial>
```


Defining the Blocks

The test will entail two blocks, a shorter practice block enabling participants to learn the task, and a longer test block in which data collection trials will be run. The practice block is defined as follows:

```
<block practice>  
/ preinstructions = (practice)  
/ trials = [1-20=noreplace(pleasanttop, pleasantbottom, unpleasanttop, unpleasantbottom)]  
/ errormessage = (errormessage, 300)  
/ recorddata = false  
</block>
```

The /preinstructions attribute specifies that a single instruction page called “practice” should be presented at the start of the block (we’ll define this page later).

The /trials attribute determines how many and which trials are run during the block. In this case, a total of 20 trials are run, randomly selected from the 4 different types of trials we defined above that make up the task. Since we’re using the “noreplace” option for randomization, Inquisit will ensure that each type of trial is run the same amount of times (5 times in this case).

Since this is a practice block, we’ll want to give error feedback to participants so they can more easily learn the task. This is accomplished using the /errormessage attribute, which indicates that whenever a participant responds incorrectly, the “errormessage” stimulus defined above is presented for 300 milliseconds.

Finally, because this is only practice, the /recorddata option is set to false so that data isn’t recorded during this block.

Next, we’ll define the test block, which is quite similar to the practice block:

```
<block test>  
/ preinstructions = (test)  
/ trials = [1-40=noreplace(pleasanttop, pleasantbottom, unpleasanttop, unpleasantbottom)]  
</block>
```

As with practice, the block randomly selects from the 4 types of trials. However, it runs a total of 40 trials instead of 20. Also, a different instruction page called “test” (defined below) is presented at the start of this block.

Defining the expt

The <expt> element combines the blocks into a complete test or experiment. Below is the <expt> element for our dot probe task:

```
<expt>
/ preinstructions = (begin)
/ blocks = [1=practice; 2=test]
/ postinstructions = (summary, end)
</expt>
```

The /blocks attribute specifies the blocks to run -- first the practice and then the test block. In addition, the /preinstructions attribute presents an instruction page called “begin” at the start of the test, and two pages “summary” and “end” at the end of the test.

Defining the instruction pages

For our dot probe, we'll use text instructions rather than HTML pages. To control the font, we'll use the <instruct> element as follows:

```
<instruct>
/ fontstyle = ("Arial", 2%, false, false, false, 5, 0)
</instruct>
```

We'll simply use the defaults setting for the font color, background color, page size, etc.

The “begin”, “practice”, “test”, and “end” pages are all defined below. Note the use of the “^” character to force additional line breaks in the text.

```
<page begin>
Dot Probe Demo
^^This sample demonstrates a simple dot probe task. On each trial, a fixation point '+' is
presented for 500 ms, followed by two words, one pleasant and the other unpleasant,
presented above and below the fixation point for 1000 ms. Both words are erased from the
screen, and one is replaced with a gray dot.
^^If the dot is in the upper position, press the 'i' key on the keyboard.
^^If the dot is in the lower position, press the 'e' key on the keyboard.
^^Respond as quickly and accurately as possible.
</page>
```

```
<page practice>
^^The next 20 trials are practice trials so you can learn the task. An error message will be
shown if you respond incorrectly.
^^ As a reminder:
^^If the dot is in the upper position, press the 'i' key on the keyboard.
^^If the dot is in the lower position, press the 'e' key on the keyboard.
</page>
```

```
<page test>
```

^^The next 40 trials are test trials. Please respond as quickly and accurately as possible. The error message will no longer be displayed after an incorrect response.

^^ As a reminder:

^^If the dot is in the upper position, press the 'i' key on the keyboard.

^^If the dot is in the lower position, press the 'e' key on the keyboard.

</page>

<page end>

^^^Test complete!

</page>

The “summary” page is intended to show the participant their average response times in the 4 different conditions of the task. To do this, the page uses a feature that enables us to dynamically inject property values into the page. To insert a particular property value, simply add the full property name to the page surrounded by “<%” and “%>”. When the page is displayed, the entire block including the delimiters will be substituted with the value of the given property.

With the summary page, we display 4 different property values, each of which returns the average response latency for the 4 different trial types. The full property names start with the type of the object “trial”, followed by the name of the particular trial (e.g. “pleasanttop”), followed by the name of the property on that trial (“meanlatency”).

<page summary>

^^Performance summary:

^^The following are average reaction times for 4 different conditions of the experiment:

^^Dot replaces pleasant word on top: <% trial.pleasanttop.meanlatency %> milliseconds.

^Dot replaces unpleasant word on top: <% trial.unpleasanttop.meanlatency %> milliseconds.

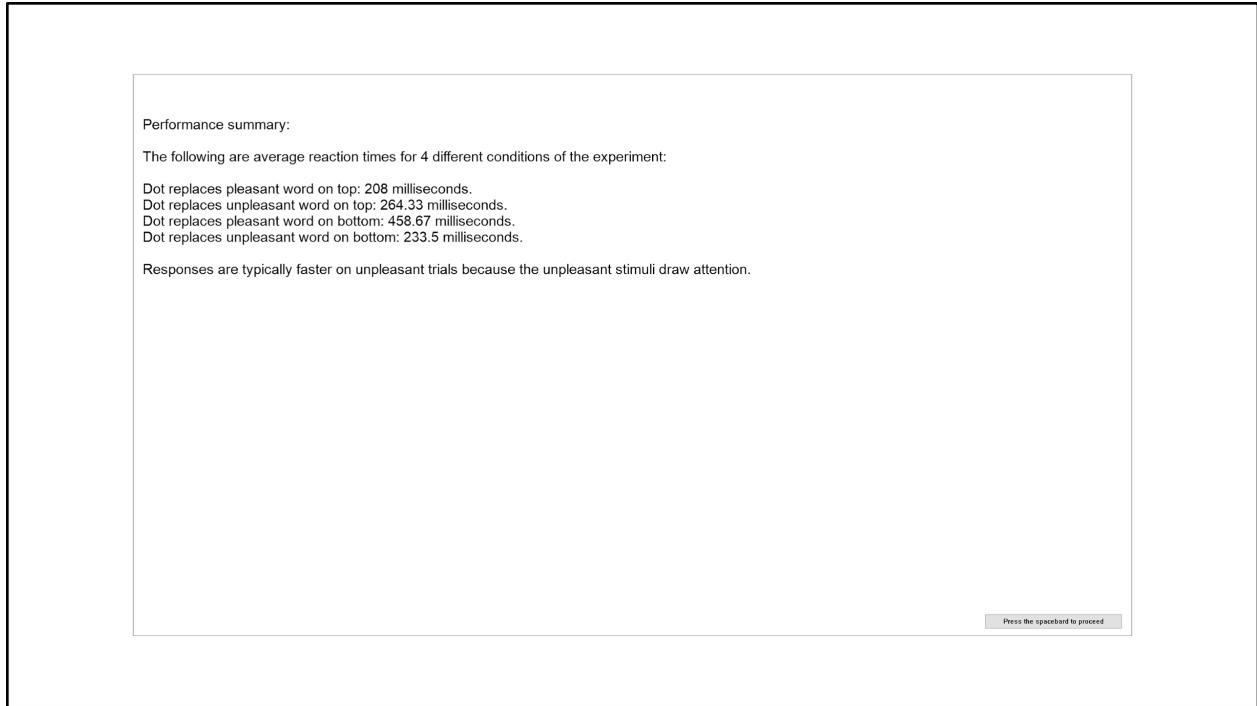
^Dot replaces pleasant word on bottom: <% trial.pleasantbottom.meanlatency %> milliseconds.

^Dot replaces unpleasant word on bottom: <% trial.unpleasantbottom.meanlatency %> milliseconds.

^^Responses are typically faster on unpleasant trials because the unpleasant stimuli draw attention.

</page>

The end result looks like the following:



Default values

Next, we'll define default settings for the dot probe:

```
<defaults>  
/ fontstyle = ("Arial", 2%, false, false, false, 5, 0)  
/ canvasaspectratio = (4, 3)  
</defaults>
```

The `/fontstyle` attribute specifies the default font to use for all text presented in the text. The `/canvasaspectratio` specifies the relative width and height of the subset of the screen to use as a canvas for presenting all stimuli.

Data

Finally, we'll define the raw and summary data to record for this task.

For the raw data, we'll record the typical variables, including the type of trial (`trialcode`), the response that was given (`response`), whether the response was correct (`correct`), the response time (`latency`), as well as the particular stimulus items presented on each trial. Additionally, the `/separatefiles` attribute is true so that data from each session is saved in a separate file to minimize potential for data loss:

```
<data>
```

```
/ columns = (date, time, subjectid, groupid, sessionid, trialcode, trialnum, response, latency,
correct, stimulusitem, stimulusitem, stimulusitem, stimulusitem)
/ separatefiles = true
</data>
```

For summary data, we'll record the average response times for each of the 4 types of trials. We'll also record the overall percent correct to flag participants who didn't follow instructions.

```
<summarydata>
/ columns = (script.startdate, script.starttime, script.subjectid, script.groupid,
script.sessionid, trial.pleasanttop.meanlatency, trial.unpleasanttop.meanlatency,
trial.pleasantbottom.meanlatency, trial.unpleasantbottom.meanlatency,
block.test.percentcorrect)
/ separatefiles = true
</summarydata>
```

That's it! Our dot probe is finished. Of course, there are numerous ways we could enhance the test, such as automatically calculating the dependent measures or giving participants an opportunity to repeat the practice block. Such enhancements would be made possible using IQScript, which is designed to support dynamic and adaptive test procedures.

Programming with IQScript

In the previous chapters, we covered the IQML language and demonstrated how it could be used to program a simple dot probe test. As a declarative, object-oriented language, IQML provides an easy-to-use framework in which test procedures are created simply by declaring objects and setting their attributes. The IQML object model provides the building blocks of an experiment. The programmers task is to configure and assemble those blocks.

The simplicity of IQML, however, comes at the expense of flexibility. With IQML, you are limited to the algorithms and procedures supported by the various IQML objects. What if your test requires logic that isn't built into these objects? This is where IQScript comes in. IQScript is a simple imperative programming language that can be used in combination with IQML to add sophisticated custom logic to a given test procedure. With IQML + IQScript, you can program almost any test procedure in Inquisit.

For those of you with any experience programming web pages, IQML + IQScript is directly analogous to HTML + Javascript. HTML provides a simple formatting language for static web pages. If all you need are pages with links to other pages, HTML has you covered. However, soon after Netscape released the first version of its browser back in 1994 and the world wide web started to take off, it quickly became clear that the web had potential to be much more than a network of interlinked documents. It could also be a means of building intelligent, interactive applications for shopping, banking, gaming, and more. Towards this vision, Netscape released the first version of Javascript in December of 1995. In 1997, Javascript was standardized by the ECMA and ISO standards organizations, and Microsoft added support for it to Internet Explorer. With Javascript support becoming widely available in browsers, static web pages evolved into applications with dynamic user interfaces that perform sophisticated tasks.

Similar to Javascript and HTML, IQScript can be embedded into certain IQML attributes so that the code can be run in order to calculate performance metrics, change the procedural flow of an experiment, or implement any logic that isn't built into IQML. Consider the following example:

```
<text tooslow>
/ onprepare = [
    text.tooslow.skip = true;
    if(block.practice.meanlatency > 1000)
    {
        text.tooslow.skip = false;
    }
]
/ items = ("Please respond more quickly!")
/ txcolor = red
/ position = (50%, 75%)
</text>
```

At the beginning of each trial, Inquisit prepares each stimulus in the trial for presentation. At the beginning of this preparation process, Inquisit executes the IQScript code (if any) defined in the `/onprepare` attribute of the stimuli. It then finishes up preparation and proceeds to the stimulus presentation phase, omitting any stimuli with the “skip” property set to true. Thus, our message is only presented if mean latency is greater than 1000, and the “skip” property is not set to true.

The purpose of the `<text>` element is to display a message in red encouraging participants to respond more quickly. However, we only wish to display this message if the participant's average response time is greater than 1000 ms. This is accomplished through the `/onprepare` attribute, which executes the block of IQScript code defined between the square brackets when the text element is prepared for presentation. By default, the message is skipped. The code then checks the mean response time of all trials within the block called “practice”. If the mean response latency is greater than 1000 ms, the code sets the “skip” property of the `<text>` element to false. In this case, the message will be presented.

IQScript can also be used to calculate custom metrics of task performance. Consider a Go/No Go task with two types of trials: 1) a “go” trial in which participants are presented a target stimulus and are instructed to press the spacebar, and 2) a “nogo” trial in which participants see a distractor stimulus and are instructed not to respond. Using the terminology of signal detection theory, if participants correctly respond to a “go” target, this is scored as a “hit”. If they do not respond on a “go” trial, the trial is classified as a “miss”. For “no go” trials, if they correctly do not respond, the trial is considered a “correct rejection”. If they mistakenly respond on a “no go” trial, the trial is scored as a “false alarm”.

The example below shows two trial definitions, one for “go” and the other for “no go”. The trials include custom logic to tally the overall counts of hits, misses, correct rejections, and false alarms. These values could be used for numerous purposes, such as to calculate “d prime”, a signal detection measure of sensitivity.

```
<trial gotrial>
/ stimulusframes = [1=target]
/ validresponse = (57, noresponse)
/ correctresponse = (57)
/ trialduration = 1000
/ ontrialend = [
    if(trial.gotrial.response == 57)
    {
        values.hitcount += 1;
    }
    else if(trial.gotrial.response == 0)
    {
        values.misscount += 1;
    }
]
```

```

</trial>

<trial nogotrial>
/ stimulusframes = [1=distractor]
/ validresponse = (57, noresponse)
/ correctresponse = (noresponse)
/ trialduration = 1000
/ ontrialend = [
    if(trial.gotrial.response == 57)
    {
        values.falsealarmcount += 1;
    }
    else if(trial.gotrial.response == 0)
    {
        Values.correctrejectioncount += 1;
    }
]
</trial>

```

This example introduces some new concepts that require explanation. First, note that the `/validresponse` attributes contain two responses, “57” and “noresponse”. The number 57 is the scan code of the space bar on a computer keyboard. (Hint - to get the scan code for a given key on your keyboard, select the “Keyboard Scancodes...” command on the Inquisit Lab Tools menu, and press the key of interest.) The “noresponse” entry is a special keyword indicating that not responding on this trial is valid. The `/correctresponse` attribute defines the correct response for the trial, which is the space bar for go trials and no response for no-go trials.

Given that noresponse is valid for both trials, the trial duration must be set to a fixed duration so that a lack of response can be registered. Without limiting the duration of the trial, the trial would wait indefinitely for a response before advancing to the next trial. Using the `/trialduration` attribute, we thus set the duration to 1000 ms (1 second). The `/trialduration` attribute causes the trial to end after 1 second if no response is given. On the other hand, if a response is given before 1 second passes, the trial waits out the remainder of the 1 second before advancing. Thus, the `/trialduration` ensures the trial is exactly 1 second long regardless of whether a response is given or not.

Finally, both trials make use of the `/ontrialend` attribute to calculate the custom metrics. The attribute takes a block of IQScript code (contained with square brackets as always). This code is executed after the trial is complete, but before data for the trial is recorded. Thus, it's an excellent place to run performance metric calculations.

For the go trial, the code checks whether the response was a spacebar press (57) or no response. If the space bar was pressed, it increments a property called “values.hitcount” by one. If no response was made, it increments a property called “values.misscount” by one. In contrast,

the no-go trial increments “values.correctrejection” if no response was given, and “values.falsealarmcount” if a response was incorrectly given.

These are just two examples of how IQML can be combined with IQScript to implement more sophisticated algorithms. This should give you a general idea of how IQScript can be used to enhance Inquisit's power and flexibility. With this larger picture in mind, we'll next drill down into the syntax of the IQScript language.

IQScript Syntax

Value Types

String, numeric, constants, booleans (true and false)

Getting and Setting Object Properties

Each object in the IQML object model has a set of properties representing the current state of the object. For example, visual stimuli such as <text> and <picture> have “vposition” and “hposition” properties representing the horizontal and vertical screen coordinates at which the stimulus is presented. The <trial> element has a “response” property representing the last response given on that trial, and a “latency” property representing the response time of the last response on that trial.

Some properties are read/write, meaning that their values can be programmatically retrieved and modified (e.g., the “hposition” and “vposition” of stimulus objects). Others are read-only, meaning that values can be retrieved via IQScript, but they can not be programmatically modified (e.g., “response” and “latency” properties of trials).

The syntax for referencing properties uses the following format for properties of elements that are named (e.g., <text>, <trial>):

type.name.property

Type is the type of object, such as “trial”, “text”, or “picture”. Name is the name given to the element. Property is the name of the property being referenced. For example, consider the following shape element:

```
<shape highlight>  
/ shape = rectangle  
/ color = yellow  
/ size = (10%, 10%)  
/ position = (33%, 33%)  
</shape>
```

The shape element offers a number of different properties which can be referenced using the following format:

```
shape.highlight.color  
shape.highlight.height  
shape.highlight.width  
shape.highlight.height  
shape.highlight.monitor
```

These examples are by no means all of the properties supported by shape. For a complete list, see reference topic for the shape element.

The IQML object model also includes singleton elements that do not have a name. For example, a script can only have one <defaults> and <data> element, so no name is required for these objects. To reference properties of unnamed elements, the format is similar to named elements except without the name component:

```
type.property
```

As with named elements, type is the type of the object (e.g., “defaults”, “data”), and property is the name of the property being referenced.

To illustrate properties on unnamed objects, we'll introduce the “display” element that is implicit to the IQML object model. The display object represents the physical display of the device that is running the test. The element can not be declared in the script, and it has no attributes. Its purpose is to expose the properties of the display so that these can be incorporated into task logic through IQScript. Below are examples of how to reference some of the properties of the display object:

```
display.height  
display.width  
display.refreshrate  
display.colordepth
```

Calling Object Functions

In addition to properties, some objects also have member functions that can be called from IQScript code. For example, the <display> object has a “getpixelsx” that returns the horizontal pixel value corresponding to the percentage value that is passed in. Stimulus elements such as <text>, <picture>, <sound>, etc. have an “appenditem” that adds another stimulus item to the set.

The syntax for calling functions is similar to that of properties. Here are a few examples:

```
text.example.hposition = display.getpixelsx(50);
```

```
picture.selecteditems.appenditem("item1.jpg");
```

Note that some functions (such as "getpixelsx") return a value, whereas others do not. Functions may also take one or more parameters, as the above examples illustrated, with parameters passed in between parentheses.

Global Functions

IQScript has dozens of built-in global functions for doing string operations, mathematical computations, statistical calculations, and more. Because the functions are global, they can be called directly without reference to a parent object. The following examples show the syntax for global functions:

```
abs(trial.foo.response);
```

The "abs" function returns the absolute value of the argument.

```
trim(openended.name.response);
```

The "trim" function trims whitespace from a string argument.

For a complete list of global functions in Inquisit, see the [Global Functions help topic](#).

Operators

Inquisit supports a number of arithmetic, logical, and comparison operators that are standard among programming languages.

Assignment Operator

=	Sets the value of the left operand to the value of the right	shape.dot.vposition = 50%;
---	--	----------------------------

Arithmetic Operators

+	Adds two operands	trial.condition1.correctcount + trial.condition2.correctcount
-	Subtracts the right operand from the left	100 - trial.mytrial.percentcorrect
*	Multiplies two operands	trial.test.correctcount * 5
/	Divides the left operand by the right	trial.mytrial.percentcorrect / 100
+=	Sets the left operand to the sum of itself and the right operand	values.trialcount += 1;

-=	Sets the left operand to the difference of itself minus the right operand	values.points -= 10;
*=	Sets the left operand to the product of itself multiplied by the right operand	shape.bar.width *= 5;
/=	Sets the left operand to the quotient of itself divided by the right operand	shape.bar.width /= 2;

Comparison Operators

==	Returns true if two operands are equal, and false otherwise	trial.iat.trialcount == 100
!=	Returns true if two operands are not equal, and false otherwise	trial.iat.trialcount != 1
<	Returns true if the left operand is less than the right, and false otherwise	trial.test.latency < 500
<=	Returns true if the left operand is less than or equal to the right, and false otherwise	trial.test.latency <= 500
>	Returns true if the left operand is greater than the right, and false otherwise	block.compat.percentcorrect > 50
>=	Returns true if the left operand is greater than or equal to the right, and false otherwise	trial.test.latency >= 500

Logical Operators

&&	Logical AND: returns true if both the left AND right operators are true, and false otherwise	block.test1.percentcorrect == 100 && block.test2.percentcorrect == 100
	Logical OR: returns true if either the left OR right operator is true, and false otherwise	trial.test.latency < 100 trial.test.latency > 1000
!	Logical NOT: Returns true if the operand is false, and false if the operand is true.	!trial.test.correct

Values element

The IQML object model includes a special <values> element, the purpose of which is to store values that can be set or retrieved with IQScript. The <values> object has no predefined

attributes. Instead, arbitrary attributes can be defined to serve as variables that can be used in the rest of the script. Consider the following example:

```
<values>
/ trialcount = 0
/ condition = ""
/ totalearnings = 0.0
</values>
```

In this example, we've defined three values. The "trialcount" value is initialized to 0, the "condition" value is initialized to an empty string, and the "totalearnings" value is initialized to decimal value 0.0. Since values are initialized before the rest of the script is parsed, they can only be set to literal constants such as integers, decimals, and strings. They can not be initialized to the values of other properties.

Having defined these values, they can now be used in IQScript as variables for storing metrics and state information as in the example below.

```
<trial collect>
/ ontrialbegin = [
    if(values.condition == "high")
    {
        values.totalearnings += 100;
    }
    else if (values.condition == "low")
    {
        values.totalearnings += 10;
    }
]
...
</trial>
```

Parameters element

The <parameters> element is similar to <values> in that its purpose is to declare variables that can be used throughout the rest of the script. The difference is that variables defined in the <parameter> element are read-only. They can not be changed from their initialized value. Parameters are a good place to define global settings controlling the test procedure, where they can be easily changed. For example, a test procedure could be programmed with the option to show performance feedback depending on whether a parameter is set to true or false.

```
<parameters>
/ showfeedback = true
/ testtrialcount = 50
```

```
/ practicetrialcount = 10  
</parameters>
```

As with the <values>, each parameter must be initialized to a simple value such as a number, string, or built-in constant. In the example above, 3 parameters are defined: “showfeedback” determines whether performance feedback is presented to the participant, “testtrialcount” determines the total number of test trials, and “practicetrialcount” determines the total number of practice trials. By defining these as parameters (and implementing the rest of the script to use them), we can change certain aspects of the procedure simply by changing the parameter value.

Keywords and Statements

Variable declarations: var

Variables are a fundamental tool of programming languages. A variable can be thought of as a named placeholder or box where values can be stored, updated, and retrieved. Variables can be used to store the results of complex calculations along with any intermediate values used in those calculations. They can store almost any kind of data, such as scores measuring a participant's performance, special instructions to present to a participant, or values indicating the participant's progress through a task.

A variable must first be declared and optionally initialized before it can be used. Inquisit provides two ways to declare variables. The first method is by adding them to the <values> or the <parameters> element using IQML as described previously. Such variables are global to the script and thus can be used anywhere else in the script. The key difference between <values> and <parameters> is that <values> are read/write and can thus be assigned different values, whereas <parameters> are read-only and will always have whatever value they were initialized to.

The second way to declare a variable is with the “var” statement in IQScript. The “var” statement allows you to define temporary variables that are local to the block of IQScript in which they are declared. These variables are read/write and can thus be assigned different values.

Variables declared with either method can be assigned values of any type, including numbers, string, and objects. So which of the two methods should one use? Generally speaking, values that are referenced multiple times during execution of the script, or across multiple code blocks, as well as values that should be recorded to the data file should be declared in the <values> element. Examples include performance metrics that are updated by different trials or blocks, and trial or block counters that track the state of the currently running task.

Values that are used temporarily within a particular code block, in contrast, should be declared with “var”. Examples include counters that track how many iterations of a loop have been run, and temporary intermediate variables used in complex calculations.

The following examples illustrate the syntax for “var” declarations:

<trial example>

```
...
/ ontrialend =
[
    var i = 0;
    while (i < 10) {
        list.conditions.appenditem(i);
    }
]
...
</trial>
```

In the first example, a variable named “sum_of_squares” is declared but not initialized. The value of an uninitialized variable is undefined, so the variable should not be used until it has been explicitly assigned a value. In the second example, a variable named “i” is declared and initialized to 0. As you can see, initialization variables in the declaration statement are optional.

Once a variable is declared, it can be referenced in code as in the lines following the declarations above.

Conditional Statements: if, else if, and else.

IQScript includes the statements “if”, “else if”, and “else” for implementing conditional logic. The “if” statement can be used standalone as in the following example:

```
if (block.test1.percentcorrect == 100)
    values.perfectscore = true;
```

The logical expression to be evaluated (i.e., the condition) must be included in parentheses. If the expression is true, the assignment expression immediately following the condition is executed. Note that standalone expressions such as the assignment must be terminated by semi-colons. To execute a block of statements if the condition is true, we can group those statements in curly braces as follows:

```
if (block.test1.percentcorrect == 100)
{
    values.perfectscore = true;
    values.stoptest = true;
}
```

Conditions can include more complex expressions such as logical AND, arithmetic operators, and function calls, as in the following example:

```
if (block.test1.percentcorrect == 100 && trim(trial.test1.response) == "red")
{
    values.perfectscore = true;
    values.stoptest = true;
}
```

The "if" statement can be combined with the "else" statement to specify expressions to run if the condition is not true. If the condition is true, the "if" block is executed, otherwise the "else" block is executed:

```
if (block.test1.percentcorrect == 100)
{
    values.perfectscore = true;
    values.maxblocks -= 2;
}
else
{
    values.perfectscore = false;
    values.maxblocks += 4;
}
```

The "if" statement can also be combined with the "else if" statement to execute blocks based on multiple conditions as in the following example:

```
if(trial.mytrial.response == 1)
{
    shape.option1.skip = false;
}
else if (trial.mytrial.response == 2)
{
    shape.option2.skip = false;
}
else if (trial.mytrial.response == 3)
{
    shape.option3.skip = false;
}
else
{
    shape.option4.skip = false;
}
```


The statement starts by comparing a trial response to the value 1. If this is true, the enclosed block is executed. If it's not true, the next condition comparing the response to 2 is evaluated. If it is true, the corresponding code block is executed. Otherwise the next "else if" statement is evaluated. If none of the conditions are true, the code in the "else" statement is executed.

Conditions aren't restricted to operations that return true or false. Numeric expressions can also be used, with 0 evaluating to false and all other numbers evaluating to true.

As a reminder, all standalone expressions must be terminated with a semicolon. Semicolons should not be used for expressions passed as arguments to functions or used in conditional statements like "if" and "else if".

Conditional Looping

Looping is a fundamental programming concept in which a block of code is repeatedly executed as long as a particular condition is met. When that condition is no longer true, the loop exits and the next line of code is run.

IQScript supports looping using the "while" statement. The "while" statement specifies a logical statement that is evaluated as true or false and a block of code to be executed, repeatedly, until the logical statement is false. The logical statement must appear in parentheses. The code block must be contained with curly braces if it has more than one line of code. Consider the following example:

```
var i = 0;
var x;
while(i < 100)
{
    x = randgaussian(100, 10);
    list.normaldistribution.append(x);
    i += 1;
}
```

Two variables are declared, *i* serving as a counter and *x* to store randomly selected values. The while loop executes repeatedly as long as the value of the *i* counter is less than 100. Inside the loop, a random value is selected from a gaussian distribution with a mean of 100 and standard deviation of 10 and stored into variable *x*. That variable is then added to a list. Finally, the counter variable *i* is incremented by 1. This loop executes exactly 100 times before the value of *i* is 100, and the while condition is no longer true.

Return Statement

IQScript also supports a return statement that ends a block of statements and optionally returns a value. Consider the following example, which illustrates branching logic selecting which trial should run next depending on which conditions are met:

```

<trial choice>
...
/ branch = [
if(trial.choice.response == "a")
    return trial.a;
else if(trial.choice.response == "b")
    return trial.b;
else if(trial.choice.response == "c")
    return trial.c;
]
...
</trial>

```

The example evaluates the response given on trial "choice", and it returns a different trial ("a", "b", or "c") depending on the value of the response.

Expressions element

The expressions element allows you to define global calculations that can be reused and referenced in IQScript. Expressions allow you to encapsulate calculations so you can reference them by an expression name rather than coding up redundant copies of the formula everywhere it is used.

Consider the following example:

```

<expressions>
/ mean_responsetime = values.sum_responsetime / values.n_responsetime
</expressions>

```

The example defines an expression called "mean_responsetime", which tracks the average response time and is computed by dividing the sum of response times by the count. With the expression so defined, we now have a variable "expressions.mean_responsetime" that we can use throughout the rest of the script. Whenever the expression is referenced, the corresponding equation is evaluated and the result returned.

The expression can be reported in data columns:

```

<data>
/ columns = (date, time, subject, group, session, expressions.mean_responsetime)
</data>

```

Or it can be used in IQScript, such as in the following example, in which a block branches to a feedback block if the average response time exceeds 600 milliseconds:

```
<block simple_reactiontime>
/ trials = [1-20 = noreplace(leftkey, rightkey)]
/ branch = [
    if(expressions.mean_responsetime > 600)
        return block.showfeedback;
]
</block>
```

Expressions can contain more complicated equations, such as in the following example, taken from the IAT test, which computes the standard deviation of response times.

```
<expressions>
/ sda = sqrt((((values.n1a - 1) * (expressions.sd1a * expressions.sd1a) + (values.n2a - 1) *
(expressions.sd2a * expressions.sd2a)) + ((values.n1a + values.n2a) * ((expressions.m1a -
expressions.m2a) * (expressions.m1a - expressions.m2a)) / 4) ) / (values.n1a + values.n2a - 1)
)
</expressions>
```

The syntax for defining expressions is relatively straightforward. The expression name is defined right after the “/” character and is followed by an equal sign “=”, after which the equation to be evaluated is defined.

Built-in Elements

IQScript includes a number of elements that are built into the object model rather than declared with IQML. These elements have properties and functions that can be called in IQScript or saved to data files, just like with IQML elements.

Script Element

The script element has properties that return values related to the current run of the script. The most commonly used properties and functions of the script element are listed below (see Inquisit's documentation for a complete list):

script.completed

This property returns a 1 at the end of the script if the participant completed the entire script, or 0 if they aborted the script before finishing. This can be stored in the summary data file to flag incomplete data sets.

script.elapsedtime

This property returns the number of milliseconds that have elapsed since the script started running. This property is useful for implementing custom timing algorithms.

`script.groupid`

This is the group number of the current session, which can be used for between-subject manipulations and counterbalancing.

`script.subjectid`

This property returns the participant's identifier for this session.

`script.sessionid`

This property runs the count of the current session, tracking which iteration of the test the given participant is on in a repeated-measures design.

`script.starttime`

This property returns the starting time of the session.

`script.startdate`

This property returns the start date of the session.

`script.abort()`

This function allows a script to be programmatically aborted. This could be useful, for example, in creating a custom consent mechanism that includes an option for participants to quit.

Display Element

The display element offers data about the graphics system on the current device. This data can be recorded to data files or used by a script to check whether the display meets minimum criteria for a given test.

`display.height`

This property returns the height of the display in pixels.

`display.width`

This property returns the width of the display in pixels.

`display.refreshinterval`

This property returns how long in milliseconds it takes the display to repaint the screen during each refresh cycle. On a 60hz monitor, for example, the value would be 16.667.

Computer Element

`computer.haskeyboard`

This property returns true if the device has a physical keyboard attached, and false otherwise.

`computer.languagecode`

This property returns the two-character, ISO 639-1 language code of the current device, such as “en” for English, “de” for German, and “jp” for Japanese. This is the default language for the device and can thus indicate the native language of the participant in online studies.

`computer.os`

This returns the name and version of the operating system the current device is running.

`computer.platform`

This returns the platform of the current device, which can be “win”, “mac”, “ios”, or “android”.

`computer.touch`

This returns true if the current device has a touchscreen, and false otherwise.

Inquisit Element

The inquisit element has information about the version of Inquisit that is currently running the script. For long term studies that span multiple Inquisit updates, this can be used to record which version of Inquisit was used for a given session.

`inquisit.releasedate`

This property returns the date the current version of Inquisit was released.

`inquisit.version`

This property returns the full version number of Inquisit that is running.

Mouse Element

The mouse element allows programmatic access to the mouse's location on the screen.

`mouse.x`

This property returns the horizontal pixel of the mouse cursor's current location. The property can also be set to programmatically change the position of the mouse. For touch screens, it returns the location of the last touch.

`mouse.y`

This property returns the vertical pixel of the mouse cursor's current location. The property can also be set to programmatically change the position of the mouse. For touch screens, it returns the location of the last touch.

Handling IQML Events with IQScript

Many elements in IQML have events that can be linked with blocks of IQScript such that whenever the event occurs, the corresponding script is executed. These events allow you to run IQScript at specific times in the flow of the procedure. This allows creation of flexible, dynamic procedures in which elements can be configured on the fly based on the state of the test or the participant's performance.

Stimulus Onprepare Event

All stimulus elements in IQML have a single event called "onprepare" that is executed immediately before the stimulus is prepared for presentation. While the attributes of a given stimulus can be configured statically in IQML, the onprepare event is useful for dynamically setting stimulus properties depending on other factors.

The following example covers a <text> element that shows a warning to participants to respond more accurately if their percent correct falls below threshold.

```
<text warning>
/ items = ("Please respond more accurately")
/ onprepare = [
    if(block.test.percentcorrect < 70)
    {
        text.warning.textcolor = red;
    }
    else if(block.test.percentcorrect < 85)
    {
        text.warning.textcolor = yellow;
    }
    else
    {
        text.warning.textcolor = black;
    }
]
/ position = (50%, 25%)
/ erase = false
</text>
```

When Inquisit prepares the stimulus for presentation, it calls the onprepare event and executes the script within. In this example, the script checks the percent of correct responses within a given block. If the accuracy is less than 70%, the warning message is changed to red. If accuracy is less than 85%, the warning message is yellow. Otherwise, the warning message is presented in black.

Trial Ontrialbegin and Ontrialend Events

All trial elements in IQML (<trial>, <openedend>, <likert>, and <slidertrial>) execute two events, ontrialbegin and ontrialend. The ontrialbegin event is fired immediately before the trial is prepared. This is useful for initialization logic for the trial, among other things. The ontrialend event is fired after the trial is complete and just before data is saved. The ontrialend event is great for scripts that update variables based on the participant's response.

The <trial> in the following example shows a trial with blocks of IQScript running in the ontrialbegin and ontrialend events:

```
<trial iowagamblingtask>
/ ontrialbegin = [
    picture.deck1.item.1 = "deck.jpg";
    picture.deck2.item.1 = "deck.jpg";
    picture.deck3.item.1 = "deck.jpg";
    picture.deck4.item.1 = "deck.jpg";
]
/ stimulusframes = [1=deck1, deck2, deck3, deck4]
/ validresponse = (deck1, deck2, deck3, deck4)
/ ontrialend = [
    if (trial.iowagamblingtask.response == "deck1") {
        picture.deck1.item.1 = "deckon.jpg";
    } else if (trial.iowagamblingtask.response == "deck2") {
        picture.deck2.item.1 = "deckon.jpg";
    } else if (trial.iowagamblingtask.response == "deck3") {
        picture.deck3.item.1 = "deckon.jpg";
    } else if (trial.iowagamblingtask.response == "deck4") {
        picture.deck4.item.1 = "deckon.jpg";
    };
]
</trial>
```

The trial presents four decks of cards on the screen using four different picture elements. In ontrialbegin, the first item in each picture is set to the same image file, "deck.jpg", which depicts the back of a deck of cards. Thus, the trial presents four decks on the screen for the participant to choose from. In the ontrialend event, the code evaluates which deck the participant selected and then changes the image file used by the corresponding picture element to "deckon.jpg"; an image that shows the back of the card similar to "deck.jpg" but with an outline around it to indicate it was selected.

Block Onblockbegin and Onblockend Events

The <block> element supports two events, onblockbegin and onblockend. The onblockbegin event fires just before the block is prepared, providing an opportunity to initialize variables at the start of a block. The onblockend event fires after the block is complete, providing a good place for cleanup code that should run at the end of the block. The following example block shows IQScript running in both the onblockbegin and onblockend events.

```
<block ANT_practice>
/ onblockbegin = [
    values.practice = true;
]
/ trials = [1-12 = noreplace(nocue, centercue, spatialcue)]
/ onblockend = [
    list.accuracy.reset();
    list.latencies.reset();
]
</block>
```

The block runs practice trials of the Short ANT procedure. In onblockbegin, a single value named “practice”, which serves as a global flag indicating the script is in practice mode, is initialized to true. In onblockend, two lists storing the accuracy and latency of each response in the block are reset - i.e., they are emptied and all selection state is cleared.

Expt Onexptbegin and Onexptend Events

The <expt> element supports two events, onexptbegin and onexptend. The onexptbegin event fires just before the experiment is prepared, providing an opportunity to initialize variables at the start of a script. The onexptend event fires after the experiment is complete but before summary data is recorded, providing a good place for cleanup code as well as calculating final performance metrics.

The following example from the Corsi Block Test shows IQScript running in both the onblockbegin and onblockend events.

```
<expt corsitask>
/ onexptbegin = [
    text.1.erasecolor=parameters.blockcolor;
    text.2.erasecolor=parameters.blockcolor;
    text.3.erasecolor=parameters.blockcolor;
    text.4.erasecolor=parameters.blockcolor;
    text.5.erasecolor=parameters.blockcolor;
    text.6.erasecolor=parameters.blockcolor;
    text.7.erasecolor=parameters.blockcolor;
```



```

text.8.erasecolor=parameters.blockcolor;
text.9.erasecolor=parameters.blockcolor;

text.1.textbgcolor=parameters.blockcolor;
text.2.textbgcolor=parameters.blockcolor;
text.3.textbgcolor=parameters.blockcolor;
text.4.textbgcolor=parameters.blockcolor;
text.5.textbgcolor=parameters.blockcolor;
text.6.textbgcolor=parameters.blockcolor;
text.7.textbgcolor=parameters.blockcolor;
text.8.textbgcolor=parameters.blockcolor;
text.9.textbgcolor=parameters.blockcolor;

shape.board.color=parameters.boardcolor;
]
/ blocks = [1=instructionsblock; 2-9=corsiblock; 10=scoreblock;]
/ onexptend = [
  if (values.ageGroup == 1){
    expressions.calculate_totalscore_byAgegroup1;
  } else if (values.ageGroup == 2){
    expressions.calculate_totalscore_byAgegroup2;
  } else if (values.ageGroup == 3){
    expressions.calculate_totalscore_byAgegroup3;
  };
]
</expt>

```

In the onexptbegin event, which fires once at the beginning of the script, the background color and the color used to erase the stimuli representing the blocks are all set to a parameter defining the block color. By storing the color in a single parameter, it can be easily changed to another color. In the onexptend event, which fires when the experiment is complete but before data is written to the summary data file, one of three different expressions is used to calculate normalized scores depending on the age of the participant.

Trial Ivalidresponse and Iscorrectresponse Events

All trial elements (<trial>, <openended>, <likert>, and <slidertrial>) support two events for evaluating responses given by participants, isvalidresponse and iscorrectresponse.

The isvalidresponse event fires immediately after a participant responds. This event works in combination with the /validresponse attribute (if it's defined) to determine whether a response is valid. If the IQScript in the isvalidresponse event returns true, and either /validresponse is not defined or the response is listed in the /validresponse attribute, the response is considered

valid. If `isinvalidresponse` returns false, or the `/validresponse` attribute is defined but does not contain the response, the response is considered invalid and is ignored.

In most cases, response validity can be handled by the `/validresponse` attribute alone. The `isinvalidresponse` event can be useful in cases requiring more complex validation logic.

In the following example from the picture story exercise, participants are instructed to type a story to go with a picture. There are no validation rules for the text they enter - they can type whatever they like. However, in order to ensure they don't simply race through the task without making an effort, the task requires them to remain on each question for a minimum amount of time before they can submit their response and proceed to the next trial. This rule is enforced by the `isinvalidresponse` event, which checks the latency of the submitted response and returns true if it is greater than or equal to the minimum time, captured by the "parameters.minstorytime" parameter, and false otherwise.

```
<opened story>
/ size = (80%, 60%)
/ stimulustimes = [0=dir, question]
/ isinvalidresponse = [
    return (    opened.story.latency >= parameters.minstorytime);
]
/ timeout = parameters.maxstorytime
</opened>
```

The `isincorrectresponse` event fires immediately after the `isinvalidresponse` event. This event works in combination with the `/correctresponse` attribute (if it's defined) to determine whether a response is valid. If the IQScript in the `isincorrectresponse` event returns true, and either `/correctresponse` is not defined or the response is listed in the `/correctresponse` attribute, the response is considered correct. If `isincorrectresponse` returns false, or the `/correctresponse` attribute is defined but does not contain the response, the response is considered incorrect.

In the example below, an `opened` trial displays a math problem and participants type the answer into a textbox. The string representing the math problem (e.g., "7 + 4") is stored in `values.mathProblem`. When the participant submits their response, the `isincorrectresponse` event fires and the math problem is evaluated on the fly using Inquisit's "evaluate" function. The result is compared to the participant's response, returning true if they are the same and false if not.

```
<opened math>
/ stimulusframes = [1 = mathProblem]
/ isincorrectresponse = [
    return (opened.math.response == evaluate(values.mathProblem));
]
/ buttonlabel = "submit"
/ mask = integer
```

</openended>

Conditional Branching with IQScript

Inquisit supports conditional branching through the `/branch` attribute, which is supported on all `<trial>` and `<block>` elements as well as their derivatives. Much like events, the branch attribute takes IQScript expressions. The script within the attribute is evaluated at the very end of the trial or block. If the expression returns a trial or block, the procedure runs that trial or block next. Those trials or blocks may in turn branch to other trials and blocks. If the `/branch` expression returns nothing, control is returned to the parent element to determine what if anything is run next. A block can only branch to a block, including itself. A trial can only branch to a trial, including itself. A block can not branch to a trial, nor a trial to a block.

The following example shows a block from a visual scanning task that includes conditional branching. The block runs 21 training trials for the task. Each time the block runs, a block counter `values.counttrainingblocks` is incremented by 1. In the branch attribute, this counter is compared against “`parameters.nr_trainingblocks`”, which indicates the total number of training blocks to run. If the counter is less than the parameter value, the branch expression returns `block.training` and the block is repeated. If the counter is greater or equal to the parameter, the branch returns `block.reset`, which clears all variables and starts the actual test trials.

```
<block training>
/ onblockbegin = [
    values.counttrainingblocks += 1;
]
/ trials = [1-21 = fixation]
/ branch = [
    if (values.counttrainingblocks < parameters.nr_trainingblocks){
        return block.training
    } else {
        return block.reset;
    };
]
</block>
```

The following example taken from the Affective Go/No Go task demonstrates branching at the trial level. This is a practice “go” trial in which participants are expected to press the go response key before the trial times out. The `/branch` attribute evaluates the `trial.go_practice.correct` property, which is true if a correct response was made on the trial and false otherwise. If the participant did respond correctly, the branch returns a trial called “`correctfeedback`” that informs the participant they responded correctly. If the participant does not respond, the branch returns the “`errorfeedback`” trial, which presents feedback indicating an error was made,

```

<trial go_practice>
/ stimulustimes = [0 = practice]
/ validresponse = (parameters.goKey)
/ correctresponse = (parameters.goKey)
/ timeout = parameters.responseDuration
/ branch = [
    if (trial.go_practice.correct){
        return trial.correctfeedback;
    } else {
        return trial.errorfeedback;
    }
];
]
</trial>

```

Text Insertion Macros

Inquisit provides a simple mechanism for inserting variables into text stimuli, instruction pages, survey captions and response options, and almost any command in IQML that takes a string. This can be useful in a number of situations, such as giving participants feedback on their performance, or for using a participant's response input as stimuli.

The following example shows the syntax for inserting variable values into a string of text for an instruction page

```

<page one>
Put your left finger on the '<%parameters.leftkey%>' response key for items that belong to
the category <%expressions.leftTarget%>. Put your right finger on the
'<%parameters.rightkey%>' response key for items that belong to the category
<%expressions.rightTarget%>.
</page>

```

The variables to be inserted appear between the delimiters “<%” and “%>”. Any valid IQScript expression can be specified within the delimiters. When Inquisit displays the page, the entire section (including delimiters) is replaced by the result of the expression contained within the delimiters. After variables are substituted, the page might look like the following:

Put your left finger on the 'E' response key for items that belong to the category Pleasant. Put your right finger on the 'I' response key for items that belong to the category Unpleasant.

Variables can be inserted almost anywhere that text can be specified. Variables can be inserted into HTML files, for example, as presented by <html> and <htmlpage>. They can be inserted

into anchor text for <likert> scales. They can also be inserted into standard <text> elements, such as in the following example:

```
<text scores>  
/ items = ("Results:  
Block Span = <%values.blockspan%>  
Total Score = <%values.totalscore%>")  
/ size = (50%, 50%)  
</text>
```

Advanced Stimulus Presentation

Selecting Items

Stimulus elements in Inquisit can be defined to represent a single stimulus item, such as a `<text>` element containing a standard error message to be displayed whenever an erroneous response is made. Or they can be defined to represent a set of related items, such as a `<picture>` element containing 52 images of standard playing cards to be selected and presented in sequence over a series of trials. For stimulus elements containing multiple items, only one item at a time can be selected and presented on a given trial. The method by which items are selected is controlled by the `/select` attribute.

Random Selection

By default, stimulus elements in Inquisit randomly select items without replacement. Once all items have been selected, they are all returned to the selection pool. This can also be explicitly specified by setting `/select = noreplace`. The following two `<text>` elements are thus functionally identical:

```
<text animals>
/ items = ("dog", "cat", "bird", "horse")
</text>
```

```
<text animals>
/ items = ("dog", "cat", "bird", "horse")
/ select = noreplace
</text>
```

To select with replacement, the `/select` attribute should be set to "replace" as in the following example:

```
<text animals>
/ items = ("dog", "cat", "bird", "horse")
/ select = replace
</text>
```

Sequential Selection

To present the item list in the order they were specified, the `/select` attribute can be set to "sequence". Once all the items have been presented, the stimulus starts over at the beginning of the item list:

```
<text animals>
/ items = ("dog", "cat", "bird", "horse")
```

```
/ select = sequence  
</text>
```

Synchronized Selection Between Stimuli

In some scenarios, the items from one stimulus set might be related to the items of another. For example, a lexical priming task might include a set of prime words (“doctor”, “cat”, “money”) and a set of semantically related targets (“nurse”, “kitten”, “wallet”). In this case, selection of the primes and targets must be coordinated so that when a given prime is selected, its corresponding target is also presented.

This can be accomplished by setting the /select attribute of one one <text> element to the name of the other:

```
<text primes>  
/ items = (“doctor”, “car”, “strength”, “hammer”)  
/ select = noreplace  
</text>
```

```
<text targets>  
/ items = (“nurse”, “automobile”, “power”, “nail”)  
/ select = primes  
</text>
```

In this example, the primes words are selected randomly without replacement. The selected target depends on which prime is selected. Specifically, the target selected is the same index as the selected prime. For example, if the second item (“car”) is randomly selected as the prime, then the second item (“automobile”) is selected as the target.

It is also possible to set up the inverse of the above example, where all target items are considered valid selections EXCEPT for the currently selected prime. Consider an evaluative priming task involving pleasant and unpleasant primes and targets, both of which are drawn from the same pool of words. On congruent trials, any target word can be paired with any prime word except for itself. We can capture this constraint by setting /select=noreplacenot, as in the following:

```
<text pleasant_prime>  
/ items = (“love”, “peace”, “happy”, “wonderful”)  
/ select = noreplace  
</text>
```

```
<text pleasant_target>  
/ items = (“love”, “peace”, “happy”, “wonderful”)  
/ select = noreplacenot(pleasant_prime)
```

</text>

In this example, primes are once again selected randomly without replacement. Targets are also selected randomly with the condition that the target word must be different from the prime. This is specified as `/select=noreplacenot(pleasant_prime)`. This command ensures that the target item at the same index as the currently selected prime is not selected.

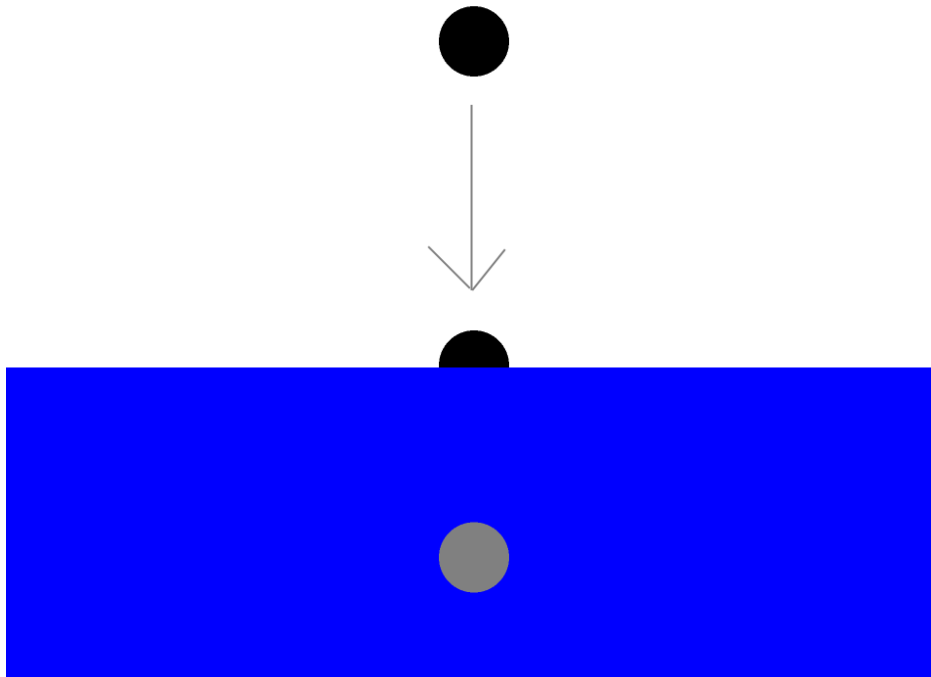
Animation

Numerous psychological testing paradigms, particularly in the areas of motion perception and time estimation, involve visual stimuli that move across the screen. For example, the Time Wall Estimation Task presents a disk that moves at a constant velocity from the top to the bottom of the screen and disappears behind a wall half way down. Participants are required to estimate when the hidden disk reaches a certain position on the wall. Another example is the Pursuit Rotor Task, in which participants manually track a disk that moves in a circle on the screen.

All visual stimuli in Inquisit (with the exception of `<html>`) can easily be animated along the dimensions of position, size, and rotation. Animated stimuli can be programmed to move along particular paths, including circles, lines, and custom sets of coordinates. Stimuli can also be programmed to change size and rotation. The speed of animations can of course be customized. Animations can also be configured to run once or to loop.

Path Animations

Path animations enable you to move stimuli along a path specified by a series of points on the screen. Consider, for example, the Time Wall Estimation Task. As illustrated below, this task presents a black disk that falls downward at a constant speed until it disappears behind a blue wall. Participants are required to estimate when the position of the falling black disk would exactly overlap the gray circle on the wall.



In this task, the disk is defined using a simple <shape> element, and its movement is controlled by the animation attribute, as in the following code.

```
<shape fallingdisk>  
/shape = circle  
/color = black  
/size = (parameters.circleradius*2*display.canvasheight/display.canvaswidth,  
parameters.circleradius*2)  
/animation = path(parameters.totalfallduration, 1, 50%, expressions.start_y, 50%,  
expressions.end_y)  
/erase = false  
</shape>
```

Note that the /shape is defined as a circle (taking into account the current canvasaspectratio), /color is set to black, and the /size defines the width and height according to the “circleradius” parameter.

Movement of the disk is specified by the /animation command. The animation is defined as a “path” with the following parameters.

- 1) Duration of the animation in milliseconds. This value determines the time it takes the animation to complete, or conversely, the speed at which the disk falls, with higher durations corresponding to slower speeds. The above example uses “totalfallduration” parameter to specify the duration.

- 2) Loop count of the animation. This determines how many times the animation will loop, with -1 indicating continuous looping. The above example sets loop count to 1, which runs the animation once.
- 3) Starting horizontal coordinate. This specifies the horizontal coordinate at which the animation starts. This is set to 50% above, which corresponds to the center of the screen.
- 4) Starting vertical coordinate. This specifies the vertical coordinate at which the animation starts. This is set to an expression called "start_y" above, which returns the top of the screen less the circle radius, so that the disk starts exactly adjacent to the top of the screen.
- 5) Ending horizontal coordinate. This specifies the horizontal coordinate of the destination point of the animation. This is set to 50% above, which is the same as the starting point, causing the disk to fall straight down.
- 6) Ending vertical coordinate. This specifies the vertical coordinate of the destination point of the animation. This is set to an expression called "end_y" above, which returns the bottom of the screen less the circle radius, so that the disk ends up exactly adjacent to the bottom of the screen.

Thus, the disk is animated to fall straight down from the top to the bottom of the screen along a path in the middle of the screen. The speed of the animation is set such that it moves from start to finish in exactly the duration specified by the first parameter.

The above example shows how to move a stimulus in a straight line between two points. With path animations, you can specify more complicated routes by including an arbitrary number of points in the path, in which case the stimulus will move from point to point until it reaches the final point.

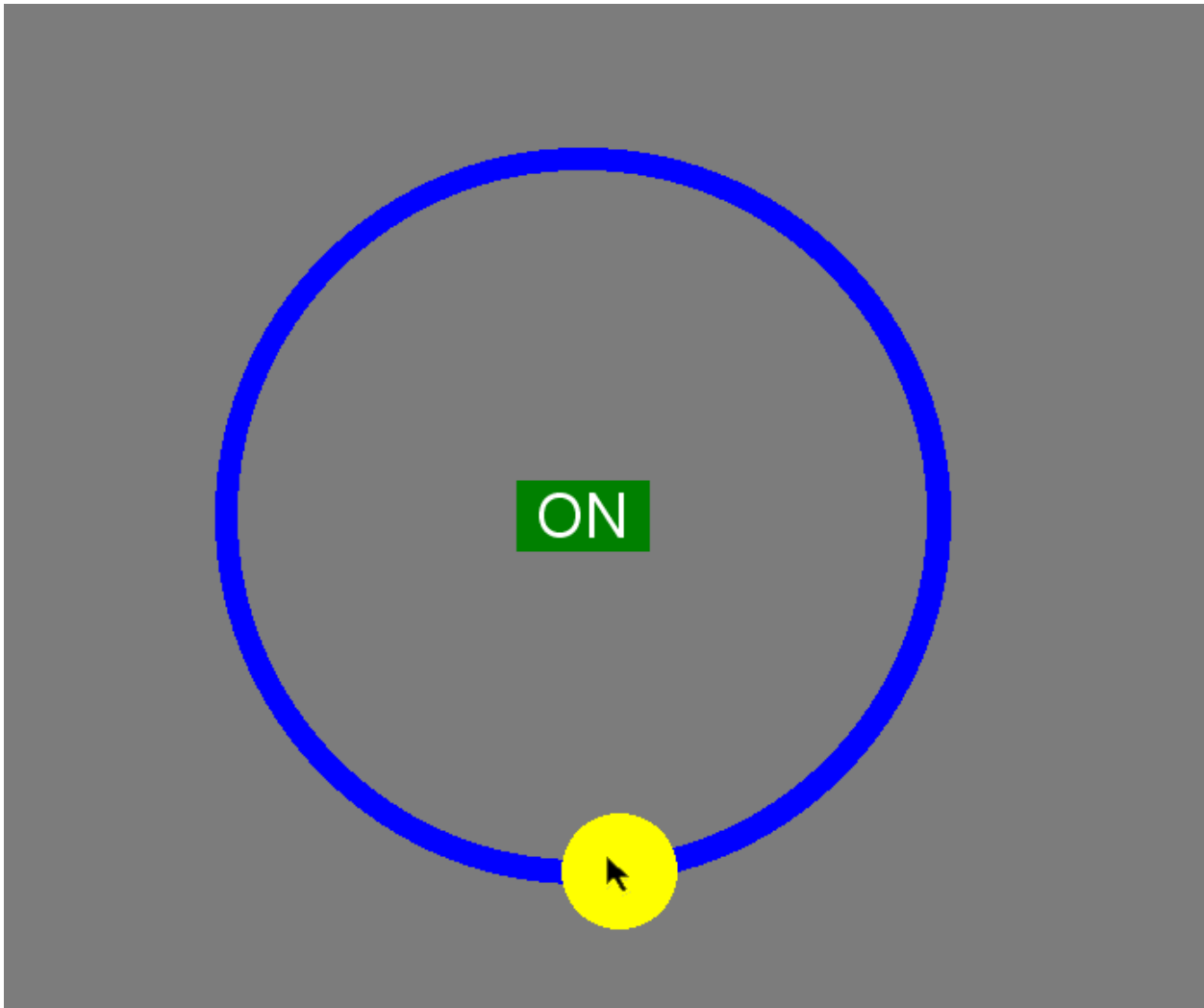
Points animations

Points animations are similar to path animations with the following exception. Whereas path animations cause a stimulus to move continuously along the lines between a set of points, the points animation simply relocates the stimulus, causing it to jump from point to point. The parameters are identical to path animations, as shown in the example below:

```
<shape jumpingdisk>  
/shape = circle  
/color = black  
/size = (parameters.circleradius*2*display.canvasheight/display.canvaswidth,  
parameters.circleradius*2)  
/animation = points(parameters.totalfallduration, 1, 50%, expressions.start_y, 50%,  
expressions.end_y)  
/ erase = false  
</shape>
```

Circle animations

Circle animations provide a simple way to move a stimulus in a circle. Consider the Pursuit Rotor task, depicted in the image below, in which participants must try to position their mouse points over a disk as it moves in a circle. When the cursor is above the circle, the feedback message “ON” is presented as feedback to the user. If the cursor moves off the circle, the message “OFF” is presented.



The tracking disk is defined in the following code.

```
<shape animatedCircle>  
/ shape = circle  
/ color = yellow  
/ size = (200%*parameters.discRadius, 200%*parameters.discRadius)
```

```
/ animation = circle(parameters.trialDuration, 1, 0, 50%, 50%,  
expressions.radius_track_inpercent)  
/ erase = false  
</shape>
```

The /shape is set to circle, /color is set to yellow, and the /size is set using the “discRadius” parameter.

Movement of the disk is specified by the /animation command. The animation is defined as a “circle” with the following parameters.

- 1) Duration of the animation in milliseconds. This value determines the time it takes the animation to complete, or alternatively, the speed at which the disk revolves, with higher durations corresponding to slower speeds. The above example uses the “trialDuration” parameter to specify the duration.
- 2) Loop count of the animation. This determines how many times the animation will loop, with -1 indicating continuous looping. The above example sets loop count to 1, which runs the animation once.
- 3) Starting point. With circle animations, the starting point is the percentage of the circular arc, ranging from 0 (top of the circle) to 100 (back to the top of the circle). This is set to 50% above, which corresponds to the bottom of the circle. Negative values can be specified for counter-clockwise animations. Values greater than 100 can be used to animate beyond a single rotation.
- 4) Horizontal center. This specifies the horizontal coordinate of the center point of the circle. This is set to 50% above, which is the center of the screen.
- 5) Vertical center. This specifies the vertical coordinate of the center point of the circle. This is set to 50% above, which is the center of the screen.
- 6) Radius. This specifies the radius of the circle, which is set to an expression called “radius_track” above.

Thus, the disk is animated to revolve around the specified circle. Its speed is set such that it completes 1 revolution in exactly the duration specified by the first parameter.

Size Animations

In addition to controlling movement of stimuli, animation can also be used to change stimulus size. The syntax for animation size is similar to that of position, except that scaling values are provided instead of screen coordinates. Consider the following example, which displays a picture that steadily increases in size.

```
<picture feedback>  
/ item = (“feedback.jpg”)  
/ size = (10%, 10%)  
/ animation = size(1000, -1, 10%, 10%, 2)  
</picture>
```

Growth of the picture is specified by the /animation command. The animation is defined as a "size" with the following parameters.

- 1) Duration of the animation in milliseconds. This value will determine the speed at which the picture grows, with higher durations corresponding to slower speeds. The above example sets the duration to 1000 milliseconds.
- 2) Loop count of the animation. This determines how many times the animation will loop, with -1 indicating continuous looping. The above example sets loop count to -1, which repeats the animation continuously.
- 3) Horizontal starting size. This specifies the horizontal starting size of the picture, which is 10% in the example above.
- 4) Vertical starting size. This specifies the vertical starting size of the picture, which is 10% in the example above.
- 5) Scale. This specifies the scaling value for the animation. In the above example, the value is 2, which indicates that the picture should double in size.

Thus, the picture is animated to double its size in 1 second. This animation is looped continuously such that it creates a pulsing effect, which can be useful for drawing attention to important stimuli.

Creating Dynamic Stimuli

As previously described, the stimuli to be presented on a given trial are specified by the /stimulusframes or /stimulustimes attributes. With these attributes, each stimulus is assigned to either a fixed frame or onset time to be displayed. This raises the question, how can the stimulus presentation time, or the actual stimuli to be presented, be varied across trials?

One method is to simply create different trial elements for each condition. For example, imagine a procedure in which one of two stimuli, an arrow pointing left or right, is presented. To accomplish this, we can create two trial elements, one that presents the left arrow and the other that presents the right, as follows:

```
<trial left>  
/ stimulusframes = [1=leftarrow]  
/ validresponse = ("e", "i")  
/ correctresponse = ("e")  
</trial>
```

```
<trial right>  
/ stimulusframes = [1=rightarrow]  
/ validresponse = ("e", "i")  
/ correctresponse = ("i")  
</trial>
```

Using two trial elements makes sense in this case because we only have two different conditions. Moreover, not only do the stimuli differ across these conditions, but the correct response is different as well.

Insertstimulustime() and insertstimulusframe() functions

Defining different trial elements may not always be feasible, however. For example, imagine a priming task in which the interstimulus interval (ISI) between prime and target is randomly varied between 300 and 500 milliseconds. In theory, we could define different trials for each of the 200 possible ISI values, but that would be quite cumbersome. Instead, we can randomly select an ISI at the beginning of each trial and use the trials's functions for inserting and removing stimuli at the corresponding times. Trial has two functions for inserting stimuli, "insertstimulusframe", which inserts a stimulus into the specified refresh frame, and "insertstimulustime", which inserts the stimulus into the specified time. We'll use the latter in the following example.

```
<trial priming>
/ ontrialbegin = [
    values.ISI = rand(300, 500);
    trial.priming.insertstimulustime(text.target, values.ISI);
]
/ stimulustimes = [0=prime]
/ validresponse = ("e", "i")
/ ontrialend = [
    trial.priming.resetstimulusframes();
]
</trial>
```

This trial always presents the prime at the beginning of the stimulus presentation sequence. In the ontrialbegin event, we've inserted code that randomly selects an integer from 300 to 500 and assigns it to a value called "ISI". Next, a stimulus called "target" is inserted into the stimulus sequence at the randomly selected time. Once inserted, the stimulus remains in the sequence even after the trial is complete. Thus, we have to remove the stimulus at the end of the trial, or we would end up presenting it twice the next time the trial is run, three times the next time, and so on. We can remove the stimulus by calling the "resetstimulusframes()" function, which removes all dynamically added stimuli and restores the sequence to how it was defined in the /stimulustimes attribute - i.e., with only the prime presented at the beginning.

Text insertion macros

Another way to dynamically alter stimuli is through text insertion macros as covered above. These macros allow you to embed expressions in blocks of text, so that when the text is processed, the value of the expression is inserted into it. Consider the following example, in which a text element presents the participant's name as entered on a previous opened trial:

```
<text name>  
/ items = (“<% opened.name.response %>”)  
</text>
```

When the text is presented, the participant's response on the “name” opened trial is inserted into the string.

Working with lists

One of the most powerful, versatile, and useful elements in the IQML object model is the list element. As you might expect, the list element allows you to create lists of values, where values can be strings, numbers, properties, or even elements including other lists. The list element is more than just a list, however. It also includes powerful functions for selecting items from the list, whether in random order, sequentially, or according to custom algorithms. Lists also include functions that return descriptive statistics for numerical items, such as the mean, median, minimum, maximum, variance, standard deviation, and even the maximum number of runs (consecutive occurrences of the same value).

List Attributes

To use a list, it must be declared in IQML. The items of the list can also be declared in IQML, or they can be programmatically added and removed via IQScript. The following example shows a list that randomly selects without replacement from 3 different conditions over a course of 30 draws. The `nextvalue` property returns a newly selected value every time it is called. Each attribute is explained below.

```
<list example>  
/ items = (values.condition1, values.condition2, values.condition3)  
/ poolsize = 30  
/ replace = false  
/ selectionmode = random  
/ selectionrate = always  
</list>
```

items

The `items` attribute allows static declaration of items in the list. Items can be numbers, strings, property values, elements, or combinations of all three (though it would be rare to combine different types of items in a list). Proportions can be varied by repeating items in the list.

replace

The `replace` attribute determines whether selection is with or without replacement. If it's set to boolean `false` or `0`, selection is without replacement. If set to `true` or any non-zero value, selection is with replacement. The attribute is `false` by default.

selectionmode

The selection mode determines how items are selected from a list. The attribute supports two built-in modes, "random" and "sequence" which select items randomly and in sequential order respectively. The attribute can also be set to an arbitrary expression including formulas and property references. The default mode is random.

selectionrate

Selectionrate determines when the list's "nextvalue" property selects a new item from the pool, or returns the currently selected item. By default, a new item is selected once per trial. It can also be set to "always", in which case a new item is selected whenever the "nextvalue" property is accessed.

Using Lists for Advanced Selection

Consider the following list, which determines the order in which a set of stimulus items should be presented.

```
<list reverseorder>  
/ items = (4, 3, 2, 1)  
/ selectionmode = sequence  
</list>
```

The list includes the numbers 1 through 4 in reverse order. The selection mode is sequential, so the list effectively selects a countdown from 4 to 1. The list can control stimulus selection using the stimulus element's "select" attribute, as in the following example.

```
<text math_problems>  
/ items = ("7 - 5", "12 x 4", "23 + 9", "43 - 28")  
/ select = list.reverseorder  
</text>
```

By setting /select equal to our list, the text element will select the item at the index returned from the reverseorder list. Thus, the fourth item is selected first, then the third item, then the second, and finally the first.

Consider the following two lists, which are used to select which of two players the computerized players in Cyberball will throw the ball to next. Players 1 and 3 are computerized, and player 2 is the participant.

```
<list player1schedule>  
/ items = (3, 3, 3, 2)  
</list>
```

```
<list player3schedule>
```



```
/ items = (1, 1, 1, 2)
</list>
```

Whenever computerized player1 or player3 receive the ball, a player number is drawn randomly from the corresponding list and the ball is thrown to that player. Given these definitions, player1 will throw to player3 75% of the time, and player3 will throw to player1 75% of the time. The two computerized players thus favor each other over player2, the participant, which serves to create a sense of social exclusion.

The list is used elsewhere in the script by the trials which implement the different combinations of throwers and catchers. Consider the following trial, which runs when player3 throws the ball to player1. After the trial is run, a branch command is executed to determine who player1 will throw the ball to, and which corresponding trial to run.

```
<trial 3to1>
...
/ branch = [
  if (list.player1schedule.nextvalue == 2) {
    return trial.1to2;
  } else if (list.player1schedule.nextvalue == 3) {
    return trial.1to3;
  };
]
</trial>
```

To get the next random value from the list, the code uses the nextvalue property. By default, the nextvalue property doesn't select a new value every time it is called. Instead, one new value is selected per trial, and subsequent calls to nextvalue on that same trial will return that same value. In the above example, the trial "1to2" is run if player2 is selected, and "1to3" is run if player3 is selected.

Using Lists for Computation

Lists can also be populated dynamically, such as when they are used for performance tracking. Consider the following example from a continuous performance test. The test tracks the accuracy and response times of all responses as well as for the different types of trials. The empty lists for tracking these data are declared in IQML as follows:

```
<list accuracy_total>
</list>

<list latencies_total>
</list>
```

These lists can then be dynamically populated in IQScript. Consider the following trial, which presents a stimulus to which participants are supposed to respond by pressing a key on the left vs one on the right. At the end of the trial, the response is analyzed and added to the “latencies_total” list. Meanwhile, the “correct” property (1 if correct, 0 otherwise) is added to the “accuracy_total” list.

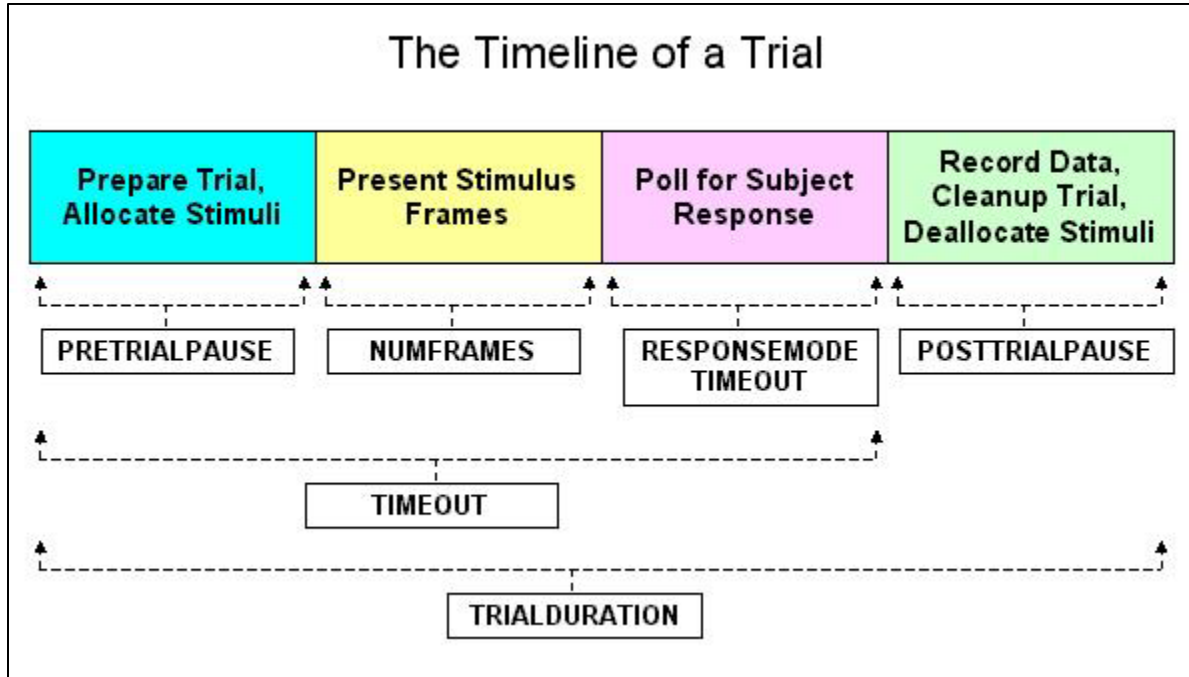
```
<trial ax>
...
/ ontrialend = [
    list.latencies_total.appenditem(trial.AX.latency, 1);
    list.accuracy_total.appenditem(trial.AX.correct, 1);
</trial>
```

The lists can then be used to compute and store performance metrics in the summarydata file. For example, the list has “mean”, “median”, and “standarddeviation” properties that return descriptive statistics for the list's content. These can be added to the summary data's columns as in the example below:

```
<summarydata >
/ columns = (computer.platform, script.startdate, script.starttime, script.subjectid,
script.groupid, script.sessionid, script.elapsedtime, script.completed,
list.latencies_total.mean,
list.latencies_total.median,
list.latencies_total.standarddeviation,
list.accuracy_total.mean)
</summarydata>
```

Trial Duration, Timeouts, and Inter-Trial Intervals

Inquisit provides a number of attributes for controlling the timing of various temporal characteristics of a trial. A trial can be thought of as a sequence of four intervals: 1) pretrialpause, 2) stimulus frames, 3) responsemode = timeout, and 4) posttrialpause. These are illustrated in the following diagram:



pretrialpause

This attribute pauses for the specified duration at the beginning of a trial, prior to stimulus presentation. In addition to providing a general means of controlling inter-trial intervals, the PretrialPause is useful for experiments that present large numbers of memory intensive stimuli on a given trial. Depending on the size of the stimuli and the speed of the hardware, stimulus preparation may add notable lengths of time to the beginning of the trial. Furthermore, stimulus preparation time may vary from trial to trial, in which case varying durations may be added to the beginning of the trials. This variance can be controlled by specifying a pretrialpause interval long enough to give Inquisit time to prepare the stimuli. Inquisit will then use this constant and predictable downtime to do all of its prep work for the trial.

stimulusframes

Specifies which stimuli should be presented on which frames for the trial. This also determines the total number of frames used by the trial. A frame corresponds to a single vertical retrace interval of the monitor.

response

By setting this attribute to a timeout procedure (e.g., `/response = timeout(1000)`), it specifies the maximum duration for Inquisit to wait for the subject to respond. If no response occurs within this duration, Inquisit finishes up the trial, waits for the `posttrialpause` to complete, and moves onto the next.

`timeout`

Specifies the maximum duration of a trial, from the very beginning of the trial to the end, not including the `posttrialpause`.

`posttrialpause`

Pauses for the specified duration at the end of each trial after the subject has responded. In addition to providing a general means of controlling inter-trial intervals, the `PosttrialPause` is useful for experiments that present large numbers of memory intensive stimuli on a given trial. Depending on the size of the stimuli and the speed of the hardware, the process of cleaning up a stimulus presentation sequence (i.e., removing stimuli from memory) may add notable lengths of time to the end of the trial. Furthermore, stimulus cleanup time may vary from trial to trial, in which case varying durations may be added to the ends of the trials. However, if a `PosttrialPause` interval is specified, Inquisit uses this time to clean up the stimulus presentation sequence. By specifying a `PosttrialPause` duration long enough for stimulus cleanup to complete, the experimenter can impose a constant and predictable duration at the end of each trial.

`trialduration`

Specifies the absolute duration of a trial, from beginning to end, including the `posttrialpause`. If the subject responds quickly, the `posttrialpause` interval is lengthened to fill out the remaining time in the duration. If the subject does not respond before the duration, the trial is terminated and the next trial begins.

Running Multiple Scripts with the Batch Element

Often a given experiment will involve running multiple scripts. For example, an experiment may administer a battery of different neuropsychological tests, or it may involve different phases, such as a memory experiment with study, interference, and test phases. Inquisit scripts can be combined and run as a sequence using the `<batch>` element, the main purpose of which is to provide an ordered list of scripts to run. The `<batch>` element is typically defined in its own standalone script called the "batch script." When a batch script is run, Inquisit evaluates the `<batch>` elements in the script and runs the resulting list of scripts.

Using batch scripts allows you to run a series of scripts without having to start each script separately. Inputs such as `subjectid`, `groupid`, and `sessionid` need only be provided once when starting the batch, and those values are passed to all the scripts.

The syntax for the `<batch>` element is shown in the following example:

```
<batch>
/ file="bart.iqx"
/ file="shortant.iqx"
/ file="ospan.iqx"
</batch>
```

In this case, the `<batch>` element specifies a simple ordered list of scripts to run. Each script file is listed using the `/file` attribute. The first script to run is the BART, then the Short ANT, and finally the OSPAN. For this batch to run correctly, all of the tests and associated materials (e.g. image files, sound files, etc.) must be saved to the same folder along with the batch script. If all files are not in the same folder, Inquisit won't be able to find and load them, and the batch script will fail.

The next example runs the same set of three tests as the previous. However, the `/selectionmode` attribute has been added:

```
<batch>
/ selectionmode = random
/ file="bart.iqx"
/ file="shortant.iqx"
/ file="ospan.iqx"
</batch>
```

The `/selectionmode` attribute determines the order in which tests are run. By default, `/selectionmode = sequence`, which indicates tests should be run in the listed order. In the example, we've set `/selectionmode = random`, which indicates the tests are run in a random order whenever the batch is launched.

Between-Group Manipulations

A batch file can also be used to run different lists of scripts for different participants. Between-subject manipulation of scripts can be used for counterbalancing order of tests or administering control and treatment conditions. Participants can be assigned a list of scripts based on the groupid, sessionid, randomly, or according to custom logic.

To run a different list of scripts for different participants, a <batch> element must be defined for each condition. Filters are then applied to the <batch> elements to determine which of them applies to the given session. For between-subject manipulations, filters are typically based on the groupid using the /groups attribute, as in the following example, which shows how to counterbalance the order of two scripts across groups:

```
<batch>  
/ groups=(1 of 2)  
/ file="script1.iqx"  
/ file="script2.iqx"  
</batch>
```

```
<batch>  
/ groups=(2 of 2)  
/ file="script2.iqx"  
/ file="script1.iqx"  
</batch>
```

In the above example, participants assigned to group id 1 will run “script1.iqx” first and “script2.iqx” second. Participants assigned to group 2 will run the scripts in the opposite order.

Between-Session Manipulations

A batch file can also be used to run different lists of scripts across different test sessions in a repeated-measures design. Between-session manipulation of scripts can be used for experiments involving training and test phases, or repeated-measure designs in which different batteries of tests are administered at different phases. With Inquisit Lab, sessions are input by the researcher at the start of the session. With Inquisit Web, sessions are automatically tracked for each participant.

To run different scripts for different sessions, a <batch> element must be defined for each type of session. Filters are then applied to the <batch> elements based on the sessionid using the /sessions attribute, as in the following example, which shows an experiment with learning and test sessions:

```
<batch>  
/ sessions=(1 of 2)  
/ file="learning.iqx"
```

```
</batch>
```

```
<batch>  
/ sessions=(2 of 2)  
/ file="test.iqx"  
</batch>
```

In the first of two sessions, "learning.iqx" is run. In the second, "test.iqx" is run. By assigning the correct session numbers, a given participant can be run through both phases over the two sessions.

The following example shows a slightly more complicated design:

```
<batch>  
/ sessions=(1, 2 of 4)  
/ file="digitspan.iqx"  
/ file="bart.iqx"  
</batch>
```

```
<batch>  
/ sessions=(3, 4 of 4)  
/ file="digitspan.iqx"  
/ file="bart.iqx"  
/ file="shortant.iqx"  
/ file="corsiblocktest.iqx"  
</batch>
```

The experiment runs a total of four sessions. In the first two sessions as indicated by the filter "/sessions=(1, 2 of 4)", the Digit Span and BART are administered. In the third and fourth sessions as indicated by the filter "/sessions=(3, 4 of 4)", the Digit Span and BART are administered along with two additional tests, the Short ANT and the Corsi Block Test.

As you can see, the syntax of the /sessions filter allows you to assign a given <batch> element to multiple sessions simply by listing the session numbers separated by commas. Similar syntax can be used to assign a given <batch> to multiple group numbers.

Resources for Programming your Own Scripts

Millisecond offers a number of resources to assist you in learning to program our platform, whether you are just starting out or already have several tests under your belt.

Millisecond Test Library

Millisecond has programmed hundreds of scripts covering a broad range of task paradigms. These scripts and source code are freely available for download. For the majority of programming projects, it can be easier to start with a similar task in our library and make modifications rather than starting from scratch. Each script is extensively commented to help you understand how various pieces of code work.

Inquisit Language Reference

This provides a complete reference for all elements and attributes in the Inquisit Language, along with examples that demonstrate usage. The reference is available through the Inquisit Lab Help menu. You can also select an element and attribute in the Inquisit Lab editor and press

F1 for help on that specific command. The reference is also available online at millisecond.com. Help for specific commands can easily be located through search engines if you include the command and the word "Inquisit" in your query.

Product Support

If you are stuck on a particular issue, Millisecond offers various channels of support where you can post questions and share relevant code snippets. Millisecond maintains a public online forum where users can post questions and search for answers. You can also get programming assistance through the support@millisecond.com email address.

Our support staff is highly knowledgeable and can quickly help you solve problems, whether they are high level questions about how to approach a given script, or issues with specific commands or procedures.