Quinta

Introduction

This document describes the language Quinta, an object oriented, stack based programming language. Quinta has been implemented on the Macintosh.

Basic Definitions

The relevant terms in Quinta are object, class, message, and response. These terms will be explained below with examples given later.

Objects represent data. Each object is an *instance* of a *class*. Different forms of data are represented by different classes. An object my be pushed on the *stack*. An object may also be stored into a variable. An object may be destroyed.

Messages manipulate data. A message is sent to the object(s) on top of the stack. The actual manipulations which take place depend entirely on the *response* of the receiving object(s) to the message which was sent. It is possible that the object(s) will not respond at all.

For a given message and object on top of the stack, the response of the object to the given message is determined by the *class* of the object. If the class of the object is defined to respond to the message, then the object will respond in the defined manner.

Classes have a relationship to one another which allows them to share responses. A class may have one or more *subclasses*. A subclass *inherits* all of the responses of its parent. Any object of the subclass may respond to any message to which objects of its parent class may respond as well. Subclasses may have subclasses themselves. Any inherited response has an original definition class. This class, the class in which the response is defined and not inherited, is referred to as the *home* of the response.

Generally, the definition of a response to a given message for a given class is another stream of messages to be sent. In other words, an object responds to a message by sending more messages.

A response to a message for a given class may be *public* or *private*. A private message may not be sent from outside of the class in which it is defined. In other words, a response may only send private messages successfully to objects of the same class as its home. A public response may be sent by the response of any object. If a private message is sent to an object by an entity outside of the class of that object, the object will not respond.

Lexical Conventions

Quinta source code is stored in text files. The code consists of tokens, delimited by white space. Delimiter characters are defined so that a token may contain white space. Quinta source code is a stream of tokens. A token can be either a message (a command) or the textual representation of an object (a data constant). A token is passed to the interpreter for processing by typing it in the Worksheet window and pressing ENTER. Quinta messages will appear in this text underlined. Capitalization is important -- the Quinta interpreter is case sensitive. Also, TOS is the abbreviation for Top Of Stack.

Quinta Constructs

The most important data structure of Quinta is the main stack, simply referred to as the **stack**. The stack is global and can hold objects, and only objects. The fundamental mechanism in execution of a Quinta program is the passing of messages to the stack. The object(s) on the stack respond appropriately to the messages, perhaps generating more messages and modifying the stack further.

The stack is used as the intermediate storage for all calculations and manipulations of data. Therefore, all operations are postfix. This "Reverse-Polish" system gives Quinta the flavor of Forth, or an RPN calculator. Any token received by the interpreter which is not a message will be processed as the textual representation of an object. If the token does not match any standard pattern for conversion to an object, it is an error.

If the token is converted to an object, it is pushed onto the stack. For example, by typing

```
487.9962
```

into the Quinta interpreter, a floating point object would be pushed onto the stack. Then by typing

```
sqrt
```

the floating point object on TOS would change to 22.0906360252. sqrt is a message. "487.9962" is the textual representation of an object. The class of this object is **float**. Objects of class float have a response defined for the message sqrt. The effect of this response is to push a new object containing the square root of the value of the receiver.

The interpreter continually obtains tokens from the user and processes them. Any token is assumed to be a message and is passed to the object on top of the stack. If the class of that object or any of its super classes have a response defined for the message specified by the token, that response is interpreted. Otherwise, the interpreter attempts to convert the token to an object and push it onto the stack.

Messages are always passed to the object or set of objects on TOS. The number of receivers of the message is determined by the **order** of the message. A message of order n requires n receivers. For example, sqrt is a message of order 1. Exactly 1 object is needed to respond to this message. An example of such an object is a **float**, which must be the object on TOS.

However, swap is a message of order 2. For this message to be interpreted, there must be a response defined for the 2 objects in level 1 and 2 of the stack. This particular message is a trivial example, because swap is defined for ANY two objects on TOS. swap has a response defined for the set of classes "generic:generic". All object classes are subclasses of generic, so any two objects can be swapped. The effect of swap is to exchange the objects in level 1 and 2 of the stack. Note that swap is not defined for a single object; i.e. the depth of the stack must be at least 2.

A more complicated example is the message *. This message is obviously used for multiplication. There are many different responses possible for this message. For example, there is a response to * defined for the classes "integer:float". In other words, if the object in level 1 of the stack is an integer and the object in level 2 is a float, then that response will be executed. The effect of this message is to multiply the two numbers a push the result onto the stack. (Incidentally, in Quinta, multiplication of an integer and a float produces a float.) There is also a **separate** response defined for the classes "float:integer". Clearly, * is a message of order 2.

This mechanism, known in some other object oriented languages as a **multimethod**, is quite flexible. For example, there is a response to * defined for "matrix:integer" which implements the multiplication of a matrix by an integer constant, pushing the resulting matrix. However, there is no response to + defined for "matrix:integer". The addition of a matrix and an integer does not produce a defined result.

Consider the following line of Quinta source:

```
45 98 + dup print drop
```

45 is an Integer, it is pushed onto the stack. 98 is handled the same way.

The next token is +. This is a message. The message is passed to the top of the stack. + is a message of order 2. Both of the top two items on the stack are of class integer. The classes integer:integer have a response defined for message +. The effect of this response is to add the two integers and return the result to the top of the stack.

dup is a message. At this time, the object on top of the stack is an integer (the sum of 45 and 98). The message dup has no response defined in class Integer. However, the class integer is indirectly a subclass of the class generic. Class generic has a response defined which duplicates the object on top of the stack. Because integer is a subclass of generic, it **inherits** all of the responses defined for generic. Consequently, objects of class integer may respond to dup, even though a special response to this message is not defined for that class. Notice, that since all classes are subclasses of generic, any object may be duplicated.

print and drop are messages which output an object's textual representation and drop an object from the top of the stack, respectively.

Note that Quinta messages are untyped. In other words, a given message does not always modify the stack in an identical known manner for all its responses. Multiplication of two integers yields an integer. Multiplication of two floats yields a float. The interpreter places no restrictions on user defined responses, except that all responses to a given message must be of the same order. However, a user is free to define a response to * for classes generic:string which exits the interpreter. It is good practice to keep the essential effect and meaning consistent in all responses to a given message. For example, drop should always drop the object from the top of stack. Quinta will happily allow you to define a response to drop which duplicates the receiver, but to do so would be poor practice.

Response definitions

The essence of Quinta programming is in defining new classes of objects and in defining new responses for the various classes.

To define a response to a message, the following four items are needed on the stack, in order (the last item is the top of the stack):
4:	A Block of tokens which will become the response
3:	A String containing the name of the message
2:	A Logical which should be TRUE if the response is to be Private.
1:	A List object, containing the set of classes for which the message is to be defined. The number of elements in this list is the order of the message.

If an attempt is made to define a response to a nonexistent message, the message is newly created.

For convenience, the messages priv and pub have been predefined as true and false, respectively.

An example response definition: Suppose we wished to define a public message to drop two objects from the stack and multiply the third by 4. We will call this message TRYME.

```
[ drop drop 4 * ] "TRYME" pub generic 1 >list
```

This set of four tokens prepares the stack to create the new response. To actually perform the operation, we must send the message respond. This will be received by the class object on top of the stack, and the response will be created.

Therefore, consider the following fragment of source:

```
[ drop drop 4 * ] "TRYME" pub generic 1 >list respond
7 4 9 TRYME print
```

The output of this fragment is

```
28
```

Class Definitions

To define a new class, the following objects are needed on top of the stack, in order:

4:	A List of strings giving the names of the Public data members of the class
3:	A List of strings giving the names of the Private data members of the class
2:	A String containing the name of the new class
1:	A class object which will be the parent class.

The message which must be sent to perform the actual operation is subclass.

For example, let us say we want to define a class to represent a window, such as is found in a Window, Icon, Mouse and Pointer operating system interface. The attributes of the window will be the location of the cursor, the size of the window, and whether or not it needs to be redrawn. These will correspond to 4 integers and a logical. The cursor coordinates will be public and the other data members will be private.

```
"Height" "Width" "Dirty" 3 >list
"X_cursor" "Y_cursor" 2 >list
"Window" generic subclass
```

There is now a class Window which has 5 member variables. In most cases, member variables may be treated exactly as "regular" variables which are described in the next section.

Variables

A variable is a place to store an object. To store an object into a variable, push the object, then push the variable and send the message sto. To recall the value stored, push the variable and send the message rcl. Once a value has been stored in a variable, simply sending the variable name as a message results in a rcl as well.

A variable object may be removed by placing it (not its value) on the stack and sending the message purge.

To clarify, a variable's contents may be pushed onto the stack by sending the name of the variable as a message. You may think of this as a message to CONTROL telling it to push the contents of a variable onto the stack. To place the variable itself on the stack, it must be surrounded by single quotes.

For example,

```
48 'simple' sto
```

this creates a variable called **simple**. After this, the message

```
simple
```

will push 48 onto the stack. However the message

'simple' will push the variable itself onto the stack. Then, sending the message

```
rcl
```

will push 48 onto the stack.

The fragment

```
33 'plok' sto plok 21 'plok' sto plok * 'plok' purge
```

will leave the value 693 on the stack. Also, after this fragment is executed, the variable 'plok' will not exist.

A user defined class in Quinta may have a number of member variables. Member variables operate as other variables, generally. For a given member variable within an object, to retrieve its contents or push it onto the stack, the object in which it resides must be on top of the stack. After a <u>sto</u> into a member variable, the newly modified object is left on the stack. For example, consider the class Window described above and presuppose that an object of class Window is in variable 'pocket'. By sending the message:
pocket
a copy of the window object in 'pocket' will be pushed onto the stack. Then, by sending the message

```
'x_cursor'
```

the x_cursor member variable of the window object will be pushed onto the stack (the actual window object will not be on the stack). Now send the message

```
78 sto
```

After this operation, the window object will again be on the stack, including its new value for x_cursor. The reason for this, is that the contents of 'pocket' have not changed! Remember, whenever a variable contents are recalled to the stack, only a copy of the contents are ever obtained. To complete this sequence, and store the resulting changes back into pocket, we must do just that:

```
'pocket' sto
```

Finally, observe that Quinta variables are untyped.

Local variables

A local variable is one which exists only for the lifetime of its enclosing and defining response. The variable may be initialized at any time during a response definition, and is destroyed upon exiting that response. Statically allocated local variables are not allowed.

The message <u>local</u> is of order 2. In level one, a string containing the name of the local variable. And, in level two, the initial contents of the local variable, which may be **any** object.

```
[ "x" local x x * ] "sq" pub integer 1 >list respond
```

The above fragment of code is one way of implementing a message to square the receiver.

Another way of doing the same thing would be:

```
[ dup * ] "sq" pub integer 1 >list respond
```

The second example uses no local variables, and is probably faster. In general, local variables make Quinta code easier to read, and less efficient. Local variables may be created at any time in a block of code but they always expect there to be at least one object on the stack to initialize the variable.

Recursion is allowed, and local variables with the same name as other local or global variables (or messages !) previously in existence do not destroy the previous definition for that name. Try examining the response to <u>fact</u> for class integer. This is a factorial function, actually defined in Quinta, and is recursive. To do this,

```
_fact_ responses print
```