Handscript: The Poplet Programming Language

I. Overview

Introduction

A Poplet module is a Palm OS application that (1) is launched from the Palm OS command bar, and (2) executes on top of another Palm OS application. Handscript is the programming language for creating Poplet modules. It is a simple language based on Javascript, the language embedded within HTML and executed by web browsers to create dynamic web pages. One of the major application areas for Poplet modules is to customize Palm OS applications to more quickly access web information.

The Poplet Kit is the handheld device resident software for creating, configuring and executing Poplet modules. You will need the Poplet Kit installed on your Palm OS device to try the Handscript examples presented here.

The Poplet Kit makes every text field in every application capable of executing Handscript expressions. After installing the Poplet Kit, launch any application that contains a text field (for example, *Memo Pad*). Enter the expression `1+2*3` into the text field and highlight that text. Popup the command bar and you will see an equal sign button ("=") which the Poplet Kit adds to the command bar. Tap that button. The result of evaluating the expression is inserted into the field. Congratulations, you are programming in Handscript!

Your First Poplet Module

A Poplet module that is launched from the command bar must have a function called *main* which has a single argument. For your first Poplet module you will need only this one function.

Start the Poplet Kit in the Poplet Module Hierarchy form showing the list of poplets on your handheld computer. The leftmost of the three pushbuttons will be selected. Tap the "New" button. You will be prompted to enter the name of a new Poplet module. Call it *First*. After creating the *First* Poplet module, you will be in the function list form with title *First*, with the second of the three pushbuttons selected. There are no functions in the list. Again, tap the "New" button. You will be in the function editor form displaying the following text:

```
     function _new_(){
     }
```

The text *_new_* will be selected. Type in the characters *main* replacing the selection. In the parentheses, insert the name of the argument, call it *arg*. Insert a new line between the first and second lines with the text '*^"Hello World!";*'.

After you are finished, the text should look as follows:

```
function main(arg){
    ^"Hello World!";
}
```

Tap the “Check” button to syntax check the function. If there is an error, a dialog box with an error message will pop up. After dismissing the dialog box, the insertion point is set at the position of the error in the code. When there is no error, the message “OK!” is briefly flashed to the right of the “Check” button.

You have just created the prototypical first application Hello World, and it’s ready to run. Bring up the command bar, tap the Poplet button and select “First” from the Poplet menu. You should get a dialog box with the Hello World text. You now have a complete Poplet module stored in its own database named “First.pop”. It can be copied or beamed to another Palm OS device.

Your Second Poplet Module

We will build on the concepts of Poplet module *First*. Create another new Poplet module named *Second*. Enter the following main function:

```
function main(arg){
    if (#arg==0) ^"There is no text selection";
    else         ^("The selection is: " + arg);
}
```

Poplet module *Second* displays the currently selected text, or if none, a message saying there is no selection. In two lines of new code we have exposed several new concepts:

1. What is the value of *arg*, the argument to the main function.
2. The # operator.
3. The == operator.
4. String concatenation with the + operator.
5. The *if else* statement.

The argument to the *main* function, *arg*, is a string containing the currently selected text in the field that has focus in the current application. This makes it easy to create poplets that process the current selection. If there is no field with focus, or no selection, the argument is the empty (zero-length) string.

The unary # operator computes the size of a string or array. For a string, it gives the number of characters in the string. For an array, it gives the number of elements contained within the array.

The == operator compares two values for equality, giving result *true* if they are the same and *false* if they are different. In the function above, we are testing whether the number of characters in *arg* is 0. The words *true* and *false* in Handscript are simply synonyms for the numbers *1* and *0*, respectively.

The result computed by the + operator depends on the types of its operands. If both operands are numbers, it computes the sum of the numbers. If either operand is a string, the other operand will be converted to a string if necessary, and the two strings will be concatenated. For example, “Hello “ + “World!” produces “Hello World!”.

The *if* and *else* statements are control structures. They control whether other statements are executed based on values of expressions. In our example, if the length of the *arg* string is zero, the first “^” statement is executed. If the length of the *arg* string is not zero, the second “^” statement is executed.

If You Know Javascript

With the popularity of the World Wide Web and the large number of people creating web pages, there are many people who don’t consider themselves programmers but who have some experience writing Javascript code. If you understand Javascript even a little, Handscript should be easy to learn. Briefly, Handscript is Javascript minus a few features that impact performance and plus a few features that make it easier to use on a small-screen device.

How Handscript and Javascript are Similar

Handscript and Javascript have many similarities:

1. They have the same syntax. For example:

```
function big(){
    return 10e100;
}
```

2. They have the same statements. For example:

```
for (i=0; i<limit; i++){
    if (a[i] > max)
        max = a[i];
}
```

3. They have the same comments. For example:

```
//This is a comment
/*This is also …
    … a comment */
```

4. They are dynamically typed. There are no type declarations. The type of a variable is determined at runtime by the type of value it contains. Elemental types are number, string and array. Variables can contain different type at different times. For example:

```
x = 50;        //x contains a number

x = "results"; //now x contains a string
```

5. They have automatic memory management. Allocation and freeing of memory is handled automatically. It is not part of the logic of a program. For example:

```
good = "good";
bad = "bad";
answer = good + " and " + bad;
if (score<70)
     answer = good + ", " + bad + " and ugly";
return answer;
```

   The code above creates the string "good and bad", which is unused when score is less than 70. The language implementation automatically reclaims the memory (garbage collects) containing this unused value.

6. They have "associative arrays". In addition to integer indexes, arrays can be indexed by strings and non-integral numbers. These arrays are useful for "associating" related values, as follows:

```
firstName["Bush"] = "George";
firstName["Gore"] = "Al";
…
```

7. Arrays can be accessed with "dot" notation:

```
person.first = “Sam”;
person.last = “Adams”;
```

   The code above is equivalent to:

```
person[“first”] = “Sam”;
person[“last”] = “Adams”;
```

8. They have "objects", enhanced arrays that have functions associated with them.

How Handscript and Javascript are Different

Handscript has the following features not in Javascript:

1. Poplet modules introduce modularity. A Poplet module is a group of functions contained in a database. Function calls can be both within-module and cross-module. For example, the Math Poplet module defines elemental mathematical functions such as square root. The square root function is called from a module other than Math as follows:

   ```
   answer = Math.sqrt(x*x + y*y);
   ```

   The module name qualifier is optional for within-module calls. The two functions calls below are equivalent if they occur in the Brackets module:

   ```
   field = Brackets.getField(arg);

   field = getField(arg);
   ```

2. Array initializers are a concise way to create arrays. Arrays are created in Handscript using braces containing comma-separated values. For example:

   ```
   testScores = {75, 91, 86, 88};

   names = {{"Pete", "Smith"}, {"Joe", "Starky"}};

   answer = { };  //this array has no elements
   ```

   Array initializers can specify non-integer indices. The firstName example above can be simplified with the following equivalent version:

   ```
   firstName = {"Bush": "George", "Gore": "Al"};
   ```

   The term "array initializer" is from C, where they are restricted to initialization contexts. In Handscript, an array initializer can be used in any expression context.

3. Global variables are explicit. Handscript global variables begin with an uppercase letter, so there can't be an inadvertent confusion with a local variable. In Javascript, global variables are those not defined in a *var* statement. With a small screen device, it could require a lot of scrolling to scan for *var* statements.

   ```
   A = B+1;  //these are global variables
   a = b+1;  //these are local variables
   ```

4. Subscript operators also apply to strings. Handscript allows brackets to subscript strings as well as arrays. The result of subscripting a string is a number, the ASCII character value at the character position. The first

character in a string has index 0. The following 3 statements have the same results:

```
c = "abcdef"[1];

c = 0cb;

c = 0x62;
```

Substrings may be obtained with the ".." slice operator. For example:

```
s = "abcdef"[1..1]; //assigns "b" to s

s = "abcdef"[2..4]; //assigns "cde" to s
```

5. There are concise operators for frequent operations. Handscript defines two frequently used unary operators: # and ^. The # unary operator computes the number of elements in an array or string. For example:

```
s = "now is the time";
a = {"now", "is", "the", "time"};
sSize = #s;    //it's 15
aSize = #a;    //it's 4
```

The ^ unary operator displays the operand value in a dialog box. For example:

```
^"Cant take square root of negative number";
```

6. The ability to call outside the language. Handscript contains provisions for Palm OS and C library calls. See Section III, "OS and C Calls".

Javascript has the following features not in Handscript:

1. The *with* statement. The Javascript *with* statement allows expressions contained within it to be simplified, by defining the default object to use in unqualified function calls and variable accesses.
2. Prototype chains for defining object inheritance. Handscript uses a simpler "module hierarchy" based inheritance scheme.

If You Know C

C is the language used to specify interfaces to the Palm OS, and to many other platforms. Most Palm OS applications are coded in C. C is a professional programmer's language. It is both low level and sophisticated. If you know C, Handscript will be very easy to learn. The languages are syntactically similar and semantically different.

How Handscript and C are Similar

Handscript and C have many similarities in syntax:

1. Expression operators and precedence are the same. For example:
```
    i = j<i && j!=0  ? j : i;
    results = ((mask^0x100)>>3)&0xFFE;
```

2. Many statements are the same. For example:
```
    sort(a, i, j); //a function call
    if ( a[i]<a[j]){
        for (k=i; k<j; k++){
            if (a[i]==0) break;
        }
    } else {
        do {
            if (a[i]>limit) continue;
            a[i] += 3;
        } while (i++<j);
    }
    return i+1;
```
3. Literals have the same syntax. For example:
```
    "One question remains.\nWill it work?"
    "\xFF"
    1.5e100
    0xFFFFFFFE
    02347
```

4. Comments are the same. For example:
```
    //This is a comment
    /*This is also …
        … a comment */
```

How Handscript and C are Different

Handscript and C have very different semantics. Handscript is a dynamicly-typed language, where the type of value contained in a variable can change at runtime. There are four types: number, string, array and undefined. The same variable can hold different types at different times. Therefore, variables need not be declared before they are used, since there is no information to specify in their declaration.

The dynamic types are smart. There is a single type of number: 8-byte floating point, compared to the many numeric types of C. Internally, 32-bit integer operations are used to optimize performance when they are certain to yield the correct results.

Arrays automatically grow when assigned with an index that hasn't been used before, rather than raise an exception or crash. So a typical way to create an array is to assign to an element that doesn't exist. For example:

```
     squares = { }; //an empty array
     for (i=0; i<10; i++) squares[i] = i*i;
```

Likewise, when an array is accessed and an element at the specified index does not exist, the result is the value `undefined`, rather than an error or crash. For example:

```
     choices = {"stop", "go"};
     return choices[4];  //returns undefined
```

In contrast, C is statically typed, with compile-time type checking. There are numerous types of signed and unsigned integers, two floating point types, structure types and union types. The emphasis is on maximizing performance, minimizing footprint, and explicit control of storage layouts and memory management. Variables must have their type declared before they are used.

C also contains macro facilities in *#define* and *#include* mechanisms that are powerful, but can make it more difficult to comprehend a program. These features are not present in Handscript.

II. Language Elements

The Big Picture

Handscript is a language for creating Poplet modules, applications which "pop up" and execute concurrently with another application. A Poplet module consists of one or more functions. A Poplet module can both serve as a pop up application and serve as a function library for other modules.

A Poplet module function can also be called from Web Clipping Applications using the Palmcall URL. The syntax of such an URL is:

```
palmcall:PPLT.appl?moduleName.functionName?argumentString
```

The function is called with argumentString as its single argument.

A Poplet module is realized as a Palm OS database (.pdb file) containing module documentation and function source code. There is one record containing all the documentation and one record per function.

The Poplet Kit is a Palm OS device resident interactive development and execution environment for Poplets modules. It presents an organized view of all Poplet modules and the elements of each module. It includes editors for the module documentation and function source code, the ability to syntax check functions, and the ability to configure the Poplet menu.

There is a set of "base modules" that are always present and cannot be changed. These modules are not contained in separate databases, but are maintained within the Poplet Kit prc. Their source code can be browsed, but not changed. The base modules correspond to many of the built-in objects of Javascript. They are: Object, Application, Array, C, Clip, Date, Math, Number, OS, String, Undefined, URL and Window.

A function contains a series of statements. A statement includes one or more expressions. An expressions consists of variables and literal values combined with operators.

The Lexical Structure of Handscript

Handscript functions are written using ASCII characters.

Case Sensitivity

Handscript is case sensitive. Language keywords, variable names and function names are distinguished based on the case of their constituent letters. Keywords, for example `if`, must be written with all lower-case letters. The names `A1` and `a1` identify different variables because of the different cases used.

Whitespace
Space, tab, newline and carriage return are “whitespace” characters. Except in string literals, multiple whitespace characters are equivalent to a single whitespace character in a Handscript function. Therefore you are free to use whitespace to format your code to improve its readability.

Comments
Handscript supports both C style and C++ style comments. Text between `//` and the end of line is ignored. Text between the characters `/*` and `*/` is ignored. For example:

```
     // this is a comment
     /* this is …
        … a comment */
```

Literals
A literal is a direct representation of a data value in a program. Handscript has number, boolean, character, string and undefined literals.

1. Integer Literals (Base 10)
   These are represented as an optional minus sign followed by a sequence of digits that does not begin with the digit zero. For example:
   ```
   5
   12000000
   ```

2. Octal Integer Literals (Base 8)
   These are represented as an optional minus sign, followed by the digit zero, followed by a sequence of digits, each between 0 and 7. Examples:
   ```
   0377
   -0100
   ```

3. Hexadecimal Integer Literals (Base 16)
   These are represented as an optional minus sign, followed by 0x or 0X, followed by a series of hexadecimal digits. A hexadecimal digit is one of the digits 0 through 9, or the letters a through f (or A through F) which represent values 10 through 15. Examples:
   ```
   0x7FFF
   0x1a000000
   ```

4. Floating Point Literals
   Floating point literals differ from integer base 10 literals in that that have either a decimal point or an exponent, or both. An exponent is an e followed by an optional plus or minus sign, followed by a one to three digit integer exponent. The number preceding the exponent is multiplied by 10 to the power of the exponent. Examples:
   ```
   .5
   1e1
   .6666666666666
   ```

5. Boolean Literals
   Boolean literals are `true` and `false`. `true` represents the integer value 1 and `false` represents the integer value `0`.

6. Character Literals
   Character literals are a single ASCII character preceded by `0c` or `0C`. They represent a number whose value is the numeric value of their ASCII character. Examples:
   ```
   0cA
   0C©
   ```

7. String Literals
   String literals are any sequence of zero or more characters enclosed within single or double quotes. Examples"
   ```
   "Can't open database"
   "12.3"
   'He said "Hello"!'
   ```

   The backslash character \ is a special "escape" character in string and character literals. Combined with the character that follows it, it represents a character that is not otherwise representable in the string. See the Table II-1 below.

Sequence	**Character**
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\'`	Single quote
`\"`	Double quote
`\\`	Back slash
`\xxx`	3 octal digit character code

Table II-1 Character Escape Sequences

8. Undefined Literal
   The literal `undefined` represents the single value undefined, which is distinguished from number, string and array values.

Identifiers

An identifier is a name for a variable or function. The first character must be a letter or an underscore "_". Subsequent characters may be a letter, digit or underscore. Examples:

```
     i
     nameLength
     _DmNextDatabaseByTypeCreator
     x1
```

Keywords

Keywords have special meaning in the language and are not available for variable and function names. The Handscript keywords are:

```
     break     continue  div       do        else
     false     for       if        undefined return
     true      var       while     typeof    new
```

Variables

A variable is a named holder or "slot" for a value. Variables allow you to manipulate values by name. A variable can hold different values at different times – hence the name variable. All variables are implicitly initialized to undefined. The example below assigns the number 5 to variable n, then uses the variable in a later computation:

```
     n = 5;
     newN = n+1;
```

Local Variables

Local variable names begin with a lower case letter or underscore. Local variables are accessible only in the single function in which they are used. The variables n and `newN` in the example above are local variables. Local variables may be declared before they are used, in a `var` statement. For example:

```
     var x, y;
     x = 5;
     y = 0;
     paintPixel(x, y);
```

A local variable persists only for the duration of execution of the function in which it appears. When a function concludes, its local variables no longer exist.

Global Variables

Global variable names begin with an uppercase letter. A global variable persists for the life of the application that contains it. Global variables are a means of sharing values among different functions or different invocations of the same function. For example:

```
     CustomerName = "Acme Co.";
     GetCustomerAddress();
     GetCustomerOrders();
```

In the example above the functions `GetCustomerAddress` and `GetCustomerOrders` can share the global variable `CustomerName`.

Types

HandScript values are one of four primitive types: number, string, array and undefined, or a higher-level type: object. Numbers use 64-bit floating-point representation. Strings are sequences of 8-bit Ascii values. Arrays are containers for values of any type. The keyword *undefined* identifies a unique value whose type is not number, string or array.

Each Poplet module defines a type of object. Modules are arranged in a hierarchy, with Object at the top of the hierarchy. A module inherits the functions of its supermodule.

HandScript is a "dynamically typed" language. That means that variables do not have a fixed type associated with them, but instead are capable of containing values of any type. The same variable can contain values of different types at different times.

Numbers

Handscript numbers are represented internally using 8-byte IEEE floating-point format. Handscript programs compute with numbers using the arithmetic operators + for addition, - for subtraction, * for multiplication and / for division. These and other numeric operators are explained later under Expressions and Operators.

In addition to these operations, there is the Math poplet which defines many basic mathematical functions. For example, the sine function can be called as `Math.sin(angle)`. And, since Handscript is extensible through the creation of new Poplet modules, you can create new libraries of numeric functions.

Strings

A string is a series of ASCII characters. Earlier we saw that a literal string can be specified by enclosing characters in double quotes. Strings are constants, they cannot be changed. But, new strings can be created through concatenation and slicing of existing strings. For example:

```
     message = "Hello "+"World!";  //it’s: "Hello World!"
     warning = "Can’t find database "+name;
     string = "Where angels fear"[6..11]; //it’s "angels"
```

You can obtain the numeric value of a string character through subscripting. You can obtain the size of a string with the # operator. For example:

```
     str = "ABCDEF";
     char = str[2]; //it’s 'C', or 67
     length = #str; //it’s 6
```

Arrays – With Integer Indexes
The simple view of arrays is that an array is an ordered list of values, where the values are indexed by integers beginning at 0. Arrays are created with initializers and accessed via subscripts.

1. Array Initializers
   An array initializer consists of a series of comma separated values enclosed in curly braces. Examples are:
   ```
   {}                         //an empty array
   {3, 4, 5, 2, 5}            //an array of numbers
   {"sam", "joe"}             //an array of strings
   {{i, j}, {i+1, j-1}}       //an array of arrays
   {0, "contents", {2, 3}}    //an array of mixed types
   ```

2. Array Access
   Array contents are accessed via subscripting. The first element of an array is at subscript position 0. For example:
   ```
   names = {"sam", "joe"};
   first = names[0];  //it’s "sam"
   second = names[1]; //it’s "joe"
   third = names[2];  //it’s undefined
   names[2] = "pete";
   third = names[2];  //now it’s "pete"
   ```

   Arrays are flexible containers. They will grow to contain new elements. In the example below, an empty array is assigned 10 elements. The array automatically grows to contain the 10 elements.
   ```
   a = {};
   for (i=0; i<10; i++) a[i] = i*i;
   // now, a == {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
   ```

   The # operator obtains the number of elements in an array. After the example above completes, `#a == 10`.

Arrays – With Non-Integer Indexes
The more complete view of arrays is that an array is a collection of pairs of indexes and values. An index is a number or a string. A value is a number, string or array. Arrays are created with initializers and accessed via subscripts.

1. Array Initializers
   An array initializer consists of a series of comma separated values or index-colon-value-pairs enclosed in curly braces. Examples are:
   ```
   {} //an empty array

   {"sam", "joe"}     //equivalent to the array below
   ```

```
{0:"sam", 1:"joe"} //value “sam” at index 0

{i+j, i-j, 20:i*j, i/j) //these two are equivalent
{0:i+j, 1:i-j, 20:i*j, 2:i/j}

{"first":"al", "last":"dugan",
      "tels": {"668-2000","776-0123"}}
```

Where a value appears without an associated index, the index is implied as follows. From left to right, the first value without an index has implied index 0. Each subsequent value without an index has implied index one greater than the index of the preceding value without an index.

2. Array Access
   In addition to numbers, array subscripts also can be strings. For example:

```
if (person["last"]=="dugan")
    telephones = person["tels"];
```

The dot notation can be used to give an object-like array access. The example above can be re-written:

```
if (person.last=="dugan")
    telephones = person.tels;
```

Array entries can be removed by assigning undefined. For example:

```
a = {"pete", "mary", "tom"};
a[1] = undefined; //a now contains {0:"pete", 2"tom"}
```

The `for in` statement iterates over the indices of an array, so it handles the results of deleted entries, as in the preceding example.

3. Array Databases
   Array notation can be used to concisely create and access persistent databases. The unary @ operator applied to a string creates an array accessor to a database with name defined by the string. This database can then be referred to as a subscripted array using the array accessor to create and access its records. For example, the code below creates the database named "Friends" containing 2 records.

```
db = @"Friends";
db["sam"] =
    {"name": "sam smith", "kids": {"bill", "sally"}};
db["joe"] = {"name": "joe hill", "kids":{}};
```

After the database is created with this code, it can be queried as follows:

```
f = @"Friends";
f["sam"]["kids"]    //this returns {"bill", "sally"}
```

A copy of an array database can be made simply as follows:

```
original = @"db1"; //get accessor to old database
copy = @"db2";     //create new database and accessor
for (i in original)
   copy[i] = original[i]; //copy record
```

Undefined
Handscript *undefined* value indicates that it is not a number, string or array. It is a convenient way for a function to indicate that it could not compute its intended result. Rather than issuing an error message or returning a “special” number, the function simply returns undefined.

As we have above with arrays, undefined avoids error messages for illegal subscripts. When an array is subscripted with an index that doesn’t have an associated value, the result is undefined.

Objects

Objects help reduce complexity in software systems by organizing the software according to the different kinds of entities defined and processed. An “object” is the result of bundling a data value and the associated code for processing that data value. In Handscript, code is organized into a collection of Poplet modules each of which defines a collection of functions. Therefore, a Handscript object is a data value associated with a Poplet module. The object is said to be “an instance” of the module. Every Handscript value is such an object.

Primitive Objects
Handscript values of type number, string, array and undefined are primitive objects which are instances of Poplet modules Number, String, Array and Undefined, respectively. These are base modules which are always present and cannot be changed.

Higher-Level Objects
Higher-level Handscript objects are created with the unary new operator applied to a string containing the name of the associated Poplet module. For example,

```
cust = new "Customer";
cust.name = "Acme Co.";
cust.address = "30 Center St.";
```

The code above creates an object that is an instance of Poplet module Customer, assigns it to the variable `cust`, and assigns values to the `name` and `address` fields of the object. Objects created with the new operator have their state accessed just like an array, either with the dot notation as illustrated above or by subscripting.

Function Calls and the Target Object
Every function call has a *target* object which determines which code is invoked by the call. The target can be either explicit or implied. An explicit target uses the dot notation consisting of the target object, a period and the function name. For example:

```
     cust.hasOrdered(productID)
```

The Poplet module of which the target object is an instance is examined to find a function definition with the same name as the function in the call. When a function call does not explicitly identify a target object, the target object is implied to be the target object used by the function containing the function call. For example,

```
     function processOrderProduct(productID){
          if (hasOrdered(productID))
               reorder(productID);
          else
               firstOrder(productID);
     }
```

The call to `hasOrdered` in the example above does not explicitly identify a target object. In this case, the target object is the same as the target object of the `processOrderProduct` function in which the call appears.

The Pseudo-Variable *this*
The pseudo-variable `this` used within a function definition identifies the target object used in calling the function. The pseudo-variable `this` may be used as a variable within a function, but it cannot be assigned to (appear on the left side of an assignment statement). The following two expressions compute the same value:

```
     hasOrdered(productID)
     this.hasOrdered(productID)
```

Global Variables
Global variables are automatically initialized at the beginning of an execution session. The initial value assigned to a global variable depends on whether or not the variable name is the same as a Poplet module name. If the global variable name is the same as some Poplet module name, then the variable is initialized to an object that is an instance of that Poplet module. This allows calls to the functions in a Poplet module to be called from another module simply be specifying the module name (as a global variable) as the target object. For example,

```
     x = Math.sqrt(2);
```

If a global variable name is not the same as any Poplet module name, then the global variable is initialized to the value `undefined`.

Inheritance and the Module Hierarchy
Handscript organizes Poplet modules into a hierarchy, where every module except for Object has a single supermodule. Module Object is at the top of the hierarchy and has no supermodule. This hierarchy is displayed in the Poplet Kit using indentation to show the submodule relationship.

When any new Poplet module is created using the Poplet Kit, the new module has module Object as its supermodule. Once a module is created, its supermodule can be changed using the Poplet Kit.

The module hierarchy defines a function inheritance hierarchy, which is used to resolve function calls. Inheritance works as follows. When a function *f* is called, the target object of the call identifies the initial Poplet module to be searched for a function definition with the name *f*. If that module contains a function *f*, then the call is resolved and the matching function is used. If there is no matching function *f*, then the supermodule of the initially searched module is considered. Successive supermodules are searched until a matching function definition is found or until the end of the supermodule chain is reached. In the latter case, the function call cannot be satisfied and execution terminates with an error dialog.

For example, below is a partial module hierarchy and a list of functions implemented in each module.

Module Name	Functions Implemented
`Object`	`eval, parseFloat, parseInt, toString, unescape, valueOf`
`   Loan`	`formatResults, main, payment, payments`
`      PQALoan`	`formatResults`

The code for function main in module Loan is as follows:

```
function main(arg){
//Display loan payment table
//Prompt for principal amount
     s = Window.prompt(
          "Enter principal amount", "");
     p = parseFloat(s);
     if (typeof p != "Number"){//not #
          ^"Enter a number";
          return;
     }
     rates = {6.0, 6.5, 7.0, 7.5, 8.0};
     times = {20, 30, 40};
```

```
          a = payments(p, rates, times);
          formatResults(a);
     }
```

The function above is called in two different ways in the following code:

```
     Loan.main("");
     PQALoan.main("");
```

In the first function call above the target object is an instance of module Loan. In the second function call above the target object is an instance of module PQALoan. Both of these calls cause invocation of function main defined in module Loan.

Where the results of these two function calls differ is in the call to function formatResults. The first `main` call has an instance of module Loan as target, so the `formatResults` call invokes the function defined in module Loan. The second call has an instance of module PQALoan as target, so the `formatResults` call invokes the function defined in module PQALoan.

This example illustrates the major benefit of inheritance. Module PQALoan reuses most of the functionality of module Loan, and supplies a different implementation for a small portion. It reimplements `formatResults` to generate a PQA to display results, rather than generate text field contents as does module Loan.

Expressions and Operators

Expressions
An *expression* is a piece of Handscript code that is evaluated to produce a value. If you are familiar with Javascript, Java or C, Handscript expressions are very similar. The simplest expressions consist of a single constant or variable. For example:

```
     2.5e4                //a numeric literal
     "I can't do that!"   //a string literal
     undefined            //the undefined literal
     true                 //a boolean literal
     x                    //a local variable
     Form                 //a global variable
```

The value of a constant expression is the value of the constant itself. The value of a variable expression is the value contained within the variable.

More complex expressions involve operators combining simpler expressions. For example, we saw that `x` is an expression and `2.5e4` is an expression, so the following is also an expression:

```
      x + 2.5e4
```

The value of this expression is determined by adding the values of the two simpler expressions. The plus sign is an *operator* used to combine two *operand* expressions into a result value. Another operator is `*` which is used to combine expressions by multiplication. For example:

```
      (x + 2.5e4) * x
```

This expression uses the `*` operator to multiply the value of the `x` variable by the value of the previous expression `x + 2.5e4`.

Operator Overview

Operators compute a result using their operands. Table II-2 below summarizes the characteristics of all Handscript operators. Operators precedence determines the order in which operators are perfomed. Higher precedence operators are performed before lower precedence operators. For example:

```
      a = b + c * d;
```

The multiplication operator * has higher precedence than the addition operator +, so the multiplication is performed before the addition. The assignment operator = has the lowest precedence, and so the assignment is done after the expression on the right is computed.

The associativity of an operator is either *right* or *left*. It determines the order in which operators of the same precedence are performed. For example:

```
      a = b + c – d;
```

is the same as:

```
      a = ((b + c) – d);
```

because the addition/subtraction operators have left associativity. On the other hand, the following expression:

```
      a = b = c = 20;
```

is equivalent to:

```
      a = (b = (c = 20));
```

because the assignment operator has right associativity.

Prece-dence	Operator	Operand Types	Operation Performed	Associ-ativity
15	[ ]	array/string, number/string	subscript	left
15	( )	function, args	function call	left
14	++	number	pre or post increment	right
14	--	number	pre or post decrement	right
14	-	number	unary minus	right
14	~	number	bitwise complement	right
14	!	number (boolean)	logical complement	right
14	^	any	alert box	right
14	#	array/string	number of elements	right
13	*	number, number	multiplication	left
13	/	number, number	division	left
13	div	number, number	integer division	left
13	%	number, number	integer remainder	left
12	+	number, number	addition	left
12	-	number, number	subtraction	left
12	+	string, any or any, string	string concatenate	left
11	<<	number, number	left shift	left
11	>>	number, number	right shift	left
10	<	numbers or strings	less than	left
10	<=	numbers or strings	less than or equal	left
10	>	numbers or strings	greater than	left
10	>=	numbers or strings	greater than or equal	left
9	==	numbers or strings or undefined, any	equal	left
9	!=	numbers or strings or undefined, any	not equal	left
8	&	number, number	bitwise and	left
7	^	number, number	bitwise exclusive or	left
6	|	number, number	bitwise or	left
5	&&	boolean, boolean	logical and	left
4	||	boolean, boolean	logical or	left
3	? :	boolean, any, any	conditional	right
2	=	variable, any	assignment	right
2	*=, /=, div=, +=, -=, <<=, >>=, &=, ^=, |=	variable, number (except += is variable, any)	assignment with operation	right
1	,	any	multiple evaluation	left

Table II-2 Operator Characteristics

Arithmetic Operators

1. Add: +
   If both operands to the + operator are numbers, the result is the sum of the operands. If either operand is a string, the other operand is converted to a string and the result is a new string that is the concatenation of the first string and the second string.
2. Subtract: -

The – operator gives the result of subtracting the second operand from the first operand. Both operands must be numbers.

3. Multiply: *
   The * operator gives the result of multiplying its two operands. Both operands must be numbers.
4. Divide: /
   The / operator gives the result of dividing the first operand by the second operand. Note that results are not forced to be an integer. Both operands must be numbers.
5. Integer Divide: `div`
   The `div` operator converts both operands to integers , then gives the result of dividing the first operand by the second operand. The result is an integer. Both operands must be numbers.
6. Modulo: %
   The % operator converts both operands to integers, then gives the result of the remainder of dividing the first operand by the second operand. The result is an integer. Both operands must be numbers.
7. Unary Minus: -
   The – operator, used as a unary operator, gives the result of negating the single operand. The operand must be a number.
8. Increment: ++
   The ++ operator does two things: it adds 1 to its single operand and it gives a result. The operand must be a variable or an array element which contains a number. The result is either the incremented operand value or the original operand value depending on whether the ++ operator precedes or follows its operand. The example below shows the two cases:
   ```
   j = 5;    //j contains 5
   m = j++;  //m contains 5, j contains 6
   n = ++j;  //n contains 7, j contains 7
   ```
9. Decrement: --
   The -- operator does two things: it subtracts 1 from its single operand and it gives a result. The operand must be a variable or an array element which contains a number. The result is either the decremented operand value or the original operand value depending on whether the -- operator precedes or follows its operand. The example below shows the two cases:
   ```
   j = 5;    //j contains 5
   m = j--;  //m contains 5, j contains 4
   n = --j;  //n contains 3, j contains 3
   ```

Comparison Operators

1. Equals: ==
   The result of the equal operator is true (1) if the two operands are equal and false (0) if they are not equal. Strings can be compared to strings, numbers to numbers and undefined to any type. Arrays only can be

compared to undefined. Undefined compares equal to undefined, and not equal to anything else. Two strings are equal if they have the same length and they have identical characters at every position. Two numbers are equal if they have exactly the same value.

2. Not Equals: !=
   The != operator computes the exact opposite of the == operator. If the result of comparing two operands for == is true, the result for comparing the same operands with != is false, and vice-versa.
3. Less Than: <
   The result of the < operator is true if the first operand is less than the second operand, otherwise the result is false. The operands must be either both numbers or both strings. String are ordered alphabetically, i.e., it is a caseless compare.
4. Greater Than: >
   The result of the > operator is true if the first operand is greater than the second operand, otherwise the result is false. The operands must be either both numbers or both strings. String are ordered alphabetically, i.e., it is a caseless compare.
5. Less Than or Equal: <=
   The result of the <= operator is true if the first operand is less than or equal to the second operand, otherwise the result is false. The operands must be either both numbers or both strings. String are ordered alphabetically, i.e., it is a caseless compare.
6. Greater Than or Equal: >=
   The result of the >= operator is true if the first operand is greater than or equal to the second operand, otherwise the result is false. The operands must be either both numbers or both strings. String are ordered alphabetically, i.e., it is a caseless compare.

Logical Operators

The logical operators require numeric operands and they perform boolean algebra on them. For the purpose of these operations, any non-zero value is considered to be true, and zero is false.

1. Logical And: &&
   The result of && is true if both operands are true, otherwise the result is false. If the first operand evaluates to false, the second operand is guaranteed not to be evaluated, because the result is known. This is useful in certain situations. For example:
   ```
           if (n!=0 && total/n>average) m++;
   ```
   The example prevents a divide by zero with the first test.
2. Logical Or: ||
   The result of || is true if at least one of the operands are true, otherwise the result is false. If the first operand evaluates to true, the second operand is guaranteed not to be evaluated, because the result is known.
3. Logical Not: !
   The ! operator is a unary operator. The result of the ! operator is true if its operand is false, and false if its operand is true.

Bitwise Operators
The bitwise operators require numeric operands and return a numeric result. Both operands are converted to 32-bit integers before the operation is done. The result of the bitwise operators is always an integer.

1. Bitwise And: &
   The result of the & operator is the boolean *and* of the two operands. A bit is set in the result value only if the corresponding bit is set in both operands.
2. Bitwise Or: |
   The result of the | operator is the boolean *or* of the two operands. A bit is set in the result value if the corresponding bit is set in one or both of the operands.
3. Bitwise Exclusive Or: ^
   The result of the ^ operator is the boolean *exclusive-or* of the two operands. A bit is set in the result value only if the corresponding bit is different in the two operands.
4. Bitwise Not: ~
   The ~ operator is a unary operator. The result of the ^ operator is to reverse all the bits of the operand. A bit is set in the result value only if it is not set in the operand.
5. Shift Left: <<
   Let n be the value of the second operand. The result of the << operator is the rightmost 32 bits of the first operand multiplied by (2 to the power n).
6. Shift Right: >>
   Let n be the value of the second operand. The result of the >> operator is the first operand divided by (2 to the power n).

Assignment Operators

There have been numerous examples of the = assignment operator. The first operand must be either a variable or a subscripted array element. The second operand is the value to be assigned to the variable or array element. Assignment is also an expression whose result is the value of the right operand. For example;

```
     x = y = z = 0; //multiple assign of single value
     if ((a=b+c)>limit)  //assign within comparison
         return a-1;
```

The operators = and == are easy to confuse, so be careful to distinguish them. The = operator is for assignment, the == operator for comparison.

There are several additional assignment operators that perform a computation of the value to be assigned. Here are examples of them:

```
     a += 2;   //equivalent to a = a + 2;
     a -= 2;   //equivalent to a = a - 2;
     a *= 2;   //equivalent to a = a * 2;
     a /= 2;   //equivalent to a = a / 2;
```

```
a div= 2; //equivalent to a = a div 2;
a %= 2;   //equivalent to a = a % 2;
a <<= 2;  //equivalent to a = a << 2;
a >>= 2;  //equivalent to a = a >> 2;
a &= 2;   //equivalent to a = a & 2;
a |= 2;   //equivalent to a = a | 2;
a ^= 2;   //equivalent to a = a ^ 2;
```

Miscellaneous Operators

1. Conditional: ? :
   The ? operator is the only three operand operator. The first operand is a boolean value used to choose the second or third operand as the result. If the first operand is true, the second operand is the result; if it is false the third operand is the result. For example:
   ```
   max = n>m ? n : m;  //pick the greater of n and m
   ```

2. Number Elements: #
   The # operator is a unary operator. The operand must be a string or an array. The result of the # operator is the number of elements in the string or array.

3. Display Alert: ^
   Earlier we saw that ^ used as a binary operator computes the exclusive-or of its operands. When used as a unary operator ^ displays the operand in an alert box. The operand can be any type. This is useful for inserting debugging code to concisely dislplay a string. But, remember that unary operators have high precedence, so that if you want to display the value of a computation, you will most likely need parentheses. For example:
   ```
   j = 5;
   ^"j="+j;        //displays "j=" in alert box
   ^("j="+j);      //displays "j=5" in alert box
   ```

4. Comma: ,
   The comma operator computes the left operand, then returns the result of computing the right operand. Its purpose is to combine what would be multiple statements into a single statement. This is sometimes useful in the *for* statement (see below). For example:
   ```
   i=10, j=5, k=3; //this is a single statement
   i=10; j=5; k=3; //these are 3 statements
   ```

5. Subscript: [ ]
   Square brackets are used for subscripting. The subscripted operand appears before the brackets and the subscript operand appears within the brackets. When the subscript expression is not being assigned to, the subscripted operand must be an array or a string. When the subscript

operator is the left operand of an assignment, the subscripted operand must be an array (i.e., you cannot assign to the elements of a string). Examples:

```
     a = {4, 3, 2, 1};
     b = {{3, 5}, {2, 4}};
     s = "here we go";
     c = s[2];           //it's 'r'
     a[1] = b[0][1];     //it's 5
```

6. Array Initializer: { }
   Curly braces are used for array initializers. The result of an array initializer is a newly created array. The elements of the array are specified by a series of comma separated values inside the braces. In the subscript example above, the variable a contains a 4-element array of numbers, indexed with integers 0, 1, 2 and 3. Array initializers can specify non-consecutive integer indexes and non-integer indexes. For example:

```
     a = {10:4, 20:3,    //non-consecutive indexes
          30:2, 40:1};
     name = {"first":"joe", //non-integer indexes
          "last":"smith"};
```

7. Function Call: ( )
   A name followed by a left parenthesis indicates a function call. The arguments to the call, if any, are a comma separated series of expression within the parentheses. For example:

```
     sine = Math.sin(angle);
     menu = Window.menu({"inches", "feet", "yards"},
          "From ");
     date = dateAndTimeNow();
```

   It is required that the function being called is actually defined and that the number of aruments in the call is the same as the number of arguments in the function definition; otherwise a runtime error will occur. In the example above, the sin function call is qualified with the name of the poplet in which the sin function is defined. This is necessary when the function call and the function definition are in different poplets.

   Handscript allows a call to use a variable function name string. In this case, the name must not be qualified with the Poplet module name. For example:

```
     functions = {"sin", "cos", "tan"};
     trig = functions[i];
     return Math[trig](x);
```

In the example above, the variable containing the function name is enclosed in parentheses to indicate that the name is computed. Otherwise, it looks like a call of a function called “trig”.

8. Value Type: typeof
   This is a unary operator that returns a string describing a value type. For the primitive values typeof returns: “Number”, “String”, “Array” and “Undefined”. For object values typeof returns a string containing the name of the module defining the object type.

9. Object Creation: new
   This is a unary operator that creates a new object. The argument is a string identifying the type of object to create. This is the same as the name of the Poplet module which defines the object’s code.

Statements

Let’s look at the hierarchy of language concepts we have in Handscript. There are: poplets, functions, statements and expressions. Generally, a Poplet module consists of one or more functions, a function consists of one or more statements, and a statement consists of one or more expressions.

Expression Statements
An expression followed by a semicolon is a statement. For example:

```
j+k;
n = m*5;
initGlobals();
```

Notice that an expression statement must change something (a variable, an array element, an external database) to do something useful. The first statement above doesn’t accomplish anything useful; it computes a sum which is then discarded.

Compound Statements
A compound statement turns a series of statements into a single statement by enclosing them in curly braces. For example:

```
//convert byte to hex
{
     digits = "0123456789ABCDEF";
     d1 = byte>>4;
     d2 = byte&0xF;
     answer = digits[d1..d1]+digits[d2..d2];
}
```

Many of the statements described below require single statements as part of their syntax, so compound statements are frequently used to turn multiple statements into a single statement in these cases. Note that a compound statement does not end with a semicolon.

Compound statements and array initializers both begin with a left brace. How can you tell which is which? If the left brace begins a statement, it begins a compound statement. If the left brace is within an expression, it begins an array initializer. For example:

```
     {x = x1+x2; y = y1+y2}        //a compound statement
     a = {x = x1+x2, y = y1+y2};   //an array initializer
```

Statements - *if*

The `if` statement is a control statement that allows you to conditionally execute code. There are two forms of the `if` statement. They are:

```
     if (expression)
         statement
```

and:

```
     if (expression)
         statement1
     else
         statement2
```

In the first form, *expression* is evaluated, and if the result is `true` *statement* is executed. If the result is `false`, *statement* is not executed. In the second form, *expression* is evaluated, then if the result is `true` *statement1* is executed, whereas if the result is `false` *statement2* is executed. Now you can see why compound statements are needed: if you want to conditionally execute a series of statements, you put them inside a single compound statement. For example:

```
     if (v[i] < v[lo]){  //exchange entries
         temp = v[i];
         v[i] = v[lo];
         v[lo] = temp;
     }
```

The indentation above is optional. It is done to improve the readability of the code. Often if statements are nested within if statements and the indentation is a big help in making the code understandable. Here is an example of nested `if` statements:

```
     if (direction=="down"){  //scroll down
         if (numberLines!=0){
             gotoLine(CurrentLine+numberLines);
         }
     } else {                 //scroll up
         if (numberLines!=0){
             gotoLine(CurrentLine-numberLines);
         }
     }
```

Statements - *while*

The while statement is a looping statement that allows you to execute code repeatedly. The while statement has the following form:

```
        while (expression)
            statement
```

The while statement works as follows. The *expression* is evaluated. If it is false, the while statement is finished, and control proceeds to the next statement in the program. If the expression evaluates to true, then *statement* (making up the body of the loop) is executed. At this point the cycle repeats, until evaluation of the *expression* is false. Here is an example while loop:

```
        sum = i = 0;
        while (i < #a)
            sum += a[i++];
```

The while loop above sums the elements of an array. The array index i starts at zero and is incremented by 1 every time through the loop. The loop terminates when i >= the number of elements in the array.

Statements - *do while*

The do while statement is another looping statement. It has the following form:

```
        do statement
        while (expression)
```

The do while statement works similarly to the while statement. First the loop body *statement* is executed. Then the while *expression* is evaluated, and, if true the loop is repeated. The difference between do while and while statements is this. In the do while statement, the loop body *statement* is always executed at least once, whereas in the while statement, the loop body *statement* may be executed zero times. Another way to look at is that the while statement has the test at the top (before the *statement*), whereas the do while statement has the test at the bottom (after the *statement*). Here is an example of a do while loop:

```
        do {
            cmd = getCommand();
            processCommand(cmd);
        while (cmd!="stop");
```

Statements - *for*

The for statement is another looping statement. It has the following form:

```
     for (initialize; test; increment)
         statement
```

The meaning of the `for` statement can be illustrated with this equivalent `while` loop:

```
     initialize;
     while (test){
         statement
         increment;
     }
```

The `for` statement is so often used because it captures the key elements of a typical loop in a single place, within the parentheses. Here is an earlier example rewritten to use a `for` loop:

```
     sum = 0;
     for (i=0; i<#a; i++)
         sum += a[i];
```

Statements - *for in*
The for in statement is a loop specifically for iterating over an array. The form of the for in statement is as follows:

```
     for (variable in array)
         statement
```

The *variable* above must be a variable. The *array* must evaluate to an array. Recall that an array in its most general form is a collection of pairs of indexes and values. This statement iterates through the pairs, assigning the index of a pair to *variable* for each iteration of the loop. For example:

```
     names = {"jim":"carey", "ben":"afleck",
             "julia":"roberts"};
     for (first in names){
         last = names[first];
         ^(first + " " + last);
     }
```

The example above assigns the strings `"jim"`, `"ben"` and `"julia"` successively to the variable `first`. Then the concatenated first and last name are displayed in an alert. Note that the order of assignment of the indexes is not necessarily the order that the indexes are declared in the initializer. In general the order is not predictable.

Statements - *break*
The break statement must appear in a loop. It causes an exit from the innermost loop in which it appears. For example:

```
     for (i=0; i<#a; i++){
        if (a[i]==searchKey) break;
    }
```

The example above searches an array for a matching search key. If it finds one, the `break` statement causes a loop exit, with the variable `i` containing the index of the matching entry. If there is no matching entry, the loop is exited with `i==#a`, which is greater than all valid array indexes.

Statements - *continue*
The `continue` statement must appear within a loop. It is similar to the `break` statement, but rather than causing a loop exit, it causes the next iteration of the loop to begin. In a `while` loop, the specified test expression is evaluated, and if `true` the loop body executed. In a `for` loop, the increment expression is evaluated, then the test expression is evaluated to see if another iteration should be done. In a `for in` loop, the loop is started at the beginning of its loop body statement with the next index being assigned to the loop variable.

The continue statement in effect aborts the current loop iteration and proceeds to the next iteration. For example:

```
    oddCount = 0;
    for (i=0; i<#a; i++){
        if ((a[i]&1)==0) continue; //it's even
        oddCount++;
    }
```

Statements - *var*
The `var` statement provides a way to explicitly declare and initialize local variables. The form of the `var` statement is the keyword `var` followed by a comma separated list of variable names and optional initialization expressions. For example:

```
    var i, j, k;
    var start = 3, end = start+5;
```

Since variables need not be declared before they are used, the `var` statement is optional. The effect of the second `var` statement above could also be accomplished as follows:

```
    start = 3; end = start+5;
```

Statements - *function*
In order for a function to be called, the function must be defined. The function statement defines a new function. The form of a function statement is as follows:

```
    function name(arguments){
        statements
    }
```

The function *name* is a single identifier, it is not qualified with the function's poplet name. *arguments* is a list of comma separated argument names. These names are variables within the statements of the function. When the function is called, the arguments are assigned the values specified in the function call expression. The number of arguments in the function definition and in a function call must match. Here are examples of function definitions:

```
//Display "Hello World!" in alert box
function main(arg){
    ^"Hello World!";
}

//Answer the mean (average) of the elements of array
function mean(array){
    answer = 0;
    for (i=0; i<#array; i++)
        answer += array[i];
    return #array==0 ? 0 : answer/#array;
}

//Answer n factorial – a recursive function
function factorial(n){
    return n<2 ? 1 : n*factorial(n-1);
}
```

Statements - *return*
The *return* statement is used to exit a function and to specify the value returned by the function. The form of a return statement is the `return` keyword, optionally followed by a return value expression and concluded with a semicoln. If the return statement does not specify a value, the value returned by the function is 0.

A function does not have to contain a return statement. For example, see the function main shown above. Such a function exits and returns a value of 0 after executing its final statement.

The *Empty* Statement
The empty statement is simply a semicolon. It performs no action, but can still be useful. For example, here is a search loop with an empty statement body.

```
//search for "joe"
for (i=0; i<#names && names[i]!="joe"; i++) ;
```

Base Modules

Base modules are physically bundled with the Poplet Kit. They are always present and cannot be changed. Therefore base module functions can be called from any module. The base modules are Object, Application, Array, C, Clip, Date, Math, Number, OS, String, Undefined, URL and Window. Many of these correspond to the Javascript core and client builtin objects.

Object
Object is the supermodule of all other modules. All of its functions are available to other object, unless redefined in a submodule. Object functions are:

`eval(script)`
: Evaluate `script` as a Handscript expression, e.g., eval("1+2");

`parseFloat(string)`
: Convert string to a number.

`parseInt(string)`
: Convert string to an integer

`toString()`
: Convert this to a string

`unescape(string)`
: Answer new string which is string with all %XX escape sequences replaced by single-character equivalent.

`valueOf()`
: Answer primitive value for numbers, strings and arrays

Application
Application is the supermodule of all Ami modules. Ami modules have two purposes: (1) to get control when the Poplet menu is requested, to perform application-specific processing (e.g., display an application menu instead of global Poplet menu), and (2) to allow application-specific implementations of getting text selections and returning text results (see the WordSmith Ami module Ami_WrdS).

An Ami module has a name of the form "Ami_XXXX". When the Poplet button is tapped, if the currently running application has a creator ID that is identical to the 'XXXX' of some Ami module, then there is said to be a matching Ami module. When there is a matching Ami module, the target object for following operations is an instance of the matching Ami module. When there is no matching Ami module, the target object for following operations is an instance of Application. Application functions are:

`doMenu(launchCode)`
: Popup standard poplet menu with no application extensions. Argument launchCode is used by those Ami modules that use two-phase clipboard copy to get text selection, value = 0 for first phase (start copy to clipboard), value = 1 for second phase (text is

available in clipboard). This function is only called by the Poplet Kit.

`enqueueRelaunch(launchCode)`
Sets launchCode and implements getting control after clipboard copy.

`getClipboardText()`
Answer clipboard text contents.

`getTextSelection()`
Answer text selection string, or “” if none.

`insertAfterSelection(string)`
Insert string after selection. Answer false if application field read-only.

`invokePoplet(name)`
Invoke poplet module *name* on selection. Display result after selection, or if application is read-only, display result in dialog.

`isFormID(n)`
Answer true if current form ID == n. Allows Ami modules to be form specific. See AppDiscover, which displays current application creator ID and form ID.

`popletMenu()`
Display standard poplet menu.

`replaceSelection(string)`
Replace selection with string. Answer false if application read-only.

`setInsertionPoint(n)`
Set insertion point to offset n within current selection. Used by compiler to identify position of syntax errors.

`startSelectionCopy()`
Strart selection copy to clipboard (if no direct copy selection possible) and answer true if two-phase selection.

Array

Array defines functions usable by array objects. Array functions are:

`concat(array2)`
Answer an array consisting of the concatenation of `this` and `array2`.

`join(separator)`
Answer a string representing the concatenation of all elements of `this`, separated by optional `separator` string.

`reverse()`
Reverse elements of `this` in place.

`sort()`
Sort elements of `this` in ascending order.

`sortFromTo(lo, hi)`
Sort elements of `this` in ascending order from index `lo` to `hi`.

`toString()`
Answer `this` converted to a string

C
C contains support for low-level C language operations, used mainly in order to perform OS calls. C functions are:

`free(pointer)`
: Free memory at integer address `pointer`.

`getFloat64(pointer)`
: Answer 64-bit float at integer address `pointer`.

`getInt16(pointer)`
: Answer signed 16-bit integer at integer address `pointer`.

`getInt32(pointer)`
: Answer signed 32-bit integer at integer address `pointer`.

`getInt8(pointer)`
: Answer signed 8-bit integer at integer address `pointer`.

`getUInt16(pointer)`
: Answer unsigned 16-bit integer at integer address `pointer`.

`getUInt32(pointer)`
: Answer unsigned 32-bit integer at integer address `pointer`.

`getUInt8(pointer)`
: Answer unsigned 8-bit integer at integer address `pointer`.

`malloc(size)`
: Allocate memory of size bytes and answer integer pointer. Abort if memory not available

`setFloat64(pointer, value)`
: Set 64-bit float at integer address `pointer` to `value`.

`setInt16(pointer, value)`
: Set 16-bit signed integer at integer address `pointer` to `value`.

`setInt32(pointer, value)`
: Set 32-bit signed integer at integer address `pointer` to `value`.

`setInt8(pointer, value)`
: Set 8-bit signed integer at integer address `pointer` to `value`.

`setUInt16(pointer, value)`
: Set 16-bit unsigned integer at integer address `pointer` to `value`.

`setUInt32(pointer, value)`
: Set 32-bit unsigned integer at integer address `pointer` to `value`.

`setUInt8(pointer, value)`
: Set 8-bit unsigned integer at integer address `pointer` to `value`.

Clip
Clip generates Palm Query Applications (PQAs). Call Clip function to generate related html entries, e.g., `Clip.b(contents)` for bold contents. See related Handwave document “Poplet Kit and Web Clipping” for descriptions of Clip functions.

Date
Date supports operations on dates. Date functions are:

`selectDay()`
: Answer a date object from date selector, or undefined if cancel.

Math
Math supports all the Javascript Math object functions and most of the MathLib functions. It requires MathLib to be installed on device. Math functions are:

`abs(x)`
: Answer absolute value of x.

`acos(x)`
: Answer the arccosine of x.

`acosh(x)`
: Answer the hyperbolic arccosine of x.

`asin(x)`
: Answer the arcsine of x.

`asinh(x)`
: Answer the hyperbolic arcsine of x.

`atan(x)`
: Answer the arctangent of x.

`atanh(x)`
: Answer the hyperbolic arctangent of x.

`cbrt(x)`
: Answer the cube root of x.

`ceil(x)`
: Answer x if x is an integer, otherwise answer the next integer greater than x.

`cos(x)`
: Answer the cosine of x.

`cosh(x)`
: Answer the hyperbolic cosine of x.

`exp(x)`
: Answer *e* raised to the power of x.

`floor(x)`
: Answer x if x is an integer, otherwise answer the next integer less than x.

`log(x)`
: Answer the natural logarithm of x.

`log10(x)`
: Answer the base 10 logarithm of x.

`log2(x)`
: Answer the base 2 logarithm of x

`main(string)`
: Popup a menu of math functions for the user to choose one to apply to `string` converted to a number.

`max(a, b)`

Answer the greater of a and b.

`min(a, b)`

Answer the lesser of a and b.

`random()`

Answer a pseudo-random number between 0 and 1.

`round(x)`

Answer x rounded to nearest integer value away from 0.

`sin(x)`

Answer the sine of x.

`sinh(x)`

Answer the hyperbolic sine of x.

`sqrt(x)`

Answer the square root of x.

`tan(x)`

Answer the tangent of x.

`tanh(x)`

Answer the hyperbolic tangent of x.

`trunc(x)`

Answer x truncated to nearest integer value not greater than x.

Number

Number defines functions applicable to numeric values. Number functions are:

`toString()`

Answer `this` converted to a string.

OS

OS contains many functions providing interfaces to Palm OS trap calls. See Palm OS documentation for definitions of these trap calls. Functions are:

```
DmCloseDatabase(dbR)

DmCreateDatabase(card, name, creator, type, resDB)

DmDatabaseInfo(card, dbID, name, attr, vers, create,
    mod, back, modno, appi, sorti, type, creator)

DmDeleteDatabase(card, id)

DmFindDatabase(card, name)

DmGetNextDatabaseByTypeCreator(newSearch, stateP,
    type, creator, latest, cardP, dbIDP)

DmGetRecord(dbR, index)

DmNewHandle(dbR, size)
```

```
DmNewRecord(dbR, indexP, size)

DmOpenDatabase(card, id, mode)

DmOpenDatabaseInfo(dbR, idP, openCtP, modeP, cardP,
    isResP)

DmQueryRecord(dbR, index)

DmReleaseRecord(dbR, index, dirty)

DmRemovRecord(dbR, index)

DmResizeRecord(dbR, index, size)

DmSetDatabaseInfo(card, dbID, name, attrs, vers,
    crDate, modDate, bckUpDate, modNum, appInfoID,
    sortInfoID, type, creator)

DmWrite(to, offset, from, size)

FldGetAttributes(field, attrP)

FldGetSelection(field, startP, endP)

FldGetTextPtr(field)

FldInsert(field, text, length)

FldSetInsertionPoint(field, position)

FldSetSelection(field, start, end)

FrmGetActiveForm()

FrmGetActiveFormID()

FrmGetFocus(form)

FrmGetNumberOfObjects(form)

FrmGetObjectPtr(form, index)

FrmGetObjectType(form, index)

FtrGet(creator, featureNum, valueP)
```

`FtrSet(creator, featureNum, value)`

`MemHandleLock(handle)`

`MemHandleSize(handle)`

`MemHandleToLocalID(handle)`

`MemLocalIDToLockedPtr(id, card)`

`MemMove(to, from, byteCount)`

`MemPtrSetOwner(ptr, owner)`

`MemPtrUnlock(ptr)`

`MemSet(ptr, num, value)`

`StrCopy(to, from)`

`StrLen(string)`

`StrNCopy(to, from, length)`

`SysAppLaunch(card, id, flags,cmd, param, resultP)`

`SysUIAppSwitch(card, id, code, param)`

`TblGetCurrentFld(table)`

`TimGetSeconds()`

`TimGetTicks()`

String
String defines functions for string values. Functions are:

`fromCharCode(char)`
Answer one character string containing character with value `char`.

`indexOf(substring, start)`
Answer index of first occurrence of `substring` in `this`, beginning at optional `start` index. If `start` is omitted, search begins at index 0.

`isVowel()`
Answer true if first character of `this` is a vowel.

`lastIndexOf(substring, start)`

Answer index of last occurrence of `substring` in `this`, beginning at `start` index. If `start` is omitted, search begins at index 0.

`newString(length)`

Answer new empty string with capacity for `length` characters. (Used for calling C code).

`split(delimitor)`

Answer an array of substrings of this delimited by string `delimitor`.

`substring(from, to)`

Answer substring of this where `from` is index of first character, `to` is optional index+1 of last character.

`toLowerCase()`

Answer a new string which is `this` with all letters lowercase.

`toString()`

Answer `this`.

`trim()`

Answer new string equal to `this` with no leading and trailing white space.

Undefined

Undefined defines functions for undefined values. Functions are:

`toString()`

Answer `this` converted to a string.

URL

URL defines functions for URL strings. Functions are:

`goto(url)`

Launch Clipper on `url` string.

`objectToQuery(object)`

Converts `object` name/value pairs to a query string of form:

```
"name1=value1&name2=value2& … "
```

`queryToObject(query)`

Converts `query` string of form:

```
"name1=value1&name2=value2& … "
```

to an object o where:

```
o.name1="value1", o.name2="value2", …
```

`script(query)`

Evaluate script in `query` string, the form submit part after "?".

1. Convert 'name=value' pairs into an object
2. Get script string.
3. Unescape script – replace "%XX" s with single characters
4. Evaluate script with object as target

`submit(object)`

Submit object of fields to the URL string value in url field. Convert `object` name/value's into query part of url string.

Window
Window defines user interface functions. Functions are:

`abort(message)`
: Display `message` string in dialog, then abort execution.

`alert(message)`
: Display `message` string in alert dialog.

`confirm(message)`
: Display `message` string in dialog. Answer true if OK button tapped, else false.

`menu(choices, title)`
: Popup a menu of `choices` (an array of strings). Place optional `title` in menu bar. Answer index of choice, or –1 if none.

`popletMenu(choices, title)`
: Popup a combined menu of local `choices` (an array of strings) followed by poplet menu. Place optional `title` in menu bar. Answer index of local choice, or –1 if none or poplet entry.

`prompt(message, default)`
: Popup prompter dialog with `message` and `default` answer. Answer response string, (a zero length string if user cancels).

OS and C Calls

Handscript is a high-level language with a breadth of features to create many useful applications. There are some applications which by their nature need to call the Palm OS or other C code. For example, an application which lists all databases on the device needs to call the OS to identify the databases. This section describes the Handscript facilities for calling C code. To use these facilities requires some understanding of the underlying C and Handscript implementations.

The OS base module contains functions which encapsulate many of the Palm OS calls. Browse the OS source code to see if the functions you need are already included.

What Does a Handscript to C Call Do

A Handscript to C call needs to satisfy the C interface. There is a Handscript call stack and a C call stack. The Handscript values must be converted to the C argument types and pushed on the C stack. Then a "trap call" instruction is done to invoke the C code. Upon return from C, the C stack arguments are discarded, the C result type is converted to a Handscript type and the result is pushed on the Handscript stack. Fortunately, most of these details are handled automatically by the C call mechanism.

C Argument and Return Types Supported

The following C argument types are supported:

Int8	Int16	Int32	Float64
UInt8	UInt16	UInt32	MemPtr
Int8 *	Int16 *	Int32 *	Float64 *
UInt8 *	UInt16 *	UInt32 *	

The following C return types are supported:

Int8	Int16	Int32	MemPtr
UInt8	UInt16	UInt32	Void

The Trap Call Statement

HandScript code calls C code using the Palm OS supported trap call mechanism. This enables calling the operating system as well as calling C code that has been packaged as a dynamic library. C interfaces are defined in terms of typed arguments. The Handscript trap call syntax is as follows:

```
[returnType trapSelector cArguments]
```

The returnType is one of the eight return types listed above. See the section below on trap selectors. The cArguments is a (possibly empty) list of arguments. Each argument has the form:

```
, argumentType argumentValue
```

The argumentType is one of the 15 argument types listed above. The argumentValue is any Handscript expression. Here is an example of the trap call statement:

```
//Call OS TimGetSeconds()
seconds = [UInt32 0xA0F5];
```

This example calls TimGetSeconds, which has no argument. The result is a 32-bit unsigned integer. A more comprehensive example is the following:

```
//Call DmNextDatabaseByTypeCreator
err = [UInt16 0xA078, UInt8 newSearch, MemPtr state,
        UInt32 type, UInt32, creator, UInt8 onlyLatest,
        UInt16Ptr &card, UInt32Ptr &dbID];
```

In this example, the return type is *UInt16*, the trap selector for *DmGetNextDatabaseByTypeCreator* is 0xA078, and the remaining elements are pairs of argument types and values. Arguments are pushed C-style, right-to-left. Each argument gets converted to the specified type before the call is done.

C Pointer Arguments

Passing pointer arguments from a language that doesn't have pointers requires some understanding of C. There are two separate cases, both illustrated in the example above: (1) pointers to Strings and structs, and (2) pointers to numbers. There are nine C types ending in "Ptr" (see above for a the complete list of types). One type, MemPtr, is used for passing strings and structs. The other 8 C pointer types are used for passing pointers to various kinds of integer and floating point numbers.

For *MemPtr*, valid HandScript argument types are string and integer. For MemPtr with a HandScript integer argument, the integer value is passed. Use an integer to pass a null (zero) pointer. If an integer is non-zero, it must be a valid C address - most likely obtained by an earlier C call. For MemPtr with a HandScript string argument, the address of the string is passed.

For the C pointer to number types, for example UInt16Ptr, pass the address of a HandScript variable that contains the number or will receive the number. This is done using the "&" operator (see example above and &card argument value). Note that a pointer can be used to pass a value in to a C call, receive a value out from a C call, or both. HandScript cannot know which is the case, so it assumes both. One implication of this is that if a pointer to a variable is passed, HandScript requires that the variable contains a number, even if the argument is output only.

Trap Selectors
The trap selector is the 16-bit argument to the hardware trap instruction that identifies which C routine to call. The trap selector values for the Palm OS are defined in include file CoreTraps.h.

HandScript supports a variant of the selector: when the selector value is greater than 0xFFFF, it is a selector pair. In this case, the higher-order 16 bits is the trap selector, and the lower order 16 bits is the sub-selector, which is pushed on the stack before the call. This form is used to call the Handspring API.

A Final Caution
Calling C from HandScript should be done carefully. A HandScript-only environment is crash free (theoretically at least), in that catastrophic user errors normally are detected and reported. On the other hand, calling C code from HandScript and passing bad arguments will very often cause a crash that requires a system reset.

III. HandScript Syntax

Lexical Grammar

Tokens

token:

  keyword
  identifier
  constant
  string-literal
  operator

Keywords

keyword: one of

break	**continue**	**div**	**do**
else	**false**	**for**	**if**
undefined	**return**	**true**	**var**
while			

Identifiers

identifier: one of

  nondigit
  identifier nondigit
  identfier digit

nondigit: one of

  _
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z

digit: one of

  0 1 2 3 4 5 6 7 8 9

Constants

constant:

  floating-constant
  integer-constant
  character-constant
  true
  false
  undefined

floating-constant:
    fractional-constant exponent-part
    digit-sequence exponent-part

fractional-constant:
    digit-seqence$_{opt}$ **.** *digit-sequence*
    digit-sequence **.**

exponent-part:
    e *sign*$_{opt}$ *digit-sequence*
    E *sign*$_{opt}$ *digit-sequence*

sign: one of
    + **-**

integer-constant:
    decimal-constant
    octal-constant
    hexadecimal-constant
    character-constant

decimal-constant:
    nonzero-digit
    decimal-constant digit

nonzero-digit: one of
    1 2 3 4 5 6 7 8 9

octal-constant:
    0
    octal-constant octal-digit

octal-digit: one of
    0 1 2 3 4 5 6 7

hexadecimal-constant:
    0x *hexadecimal-digit*
    0X *hexadecimal-digit*
    hexadecimal-constant hexadecimal-digit

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

character-constant:

0c*single-quote-char*
0C*single-quote-char*

String Literals

string-literal:

"*double-quote-sequence*$_{opt}$**"**
'*single-quote-sequence*$_{opt}$**'**

double-quote-sequence:

double-quote-char
double-quote-sequence double-quote-char

single-quote-sequence:

single-quote-char
single-quote-sequence single-quote-char

double-quote-char:

escape sequence
any member of the source character set except the double-quote **"** backslash \ or new-line character

single-quote-char:

escape sequence
any member of the source character set except the single-quote **'** backslash \ or new-line character

escape-sequence:

simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of

**\' \" \? **

\a \b \f \n \r \t \v

octal-escape-sequence:

\octal-digit

\octal-digit octal-digit

\octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:

\x *hexadecimal-digit*

hexadecimal-escape-sequence hexadecimal-digit

unicode-escape-sequence:

\u *hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit*

Operators

operator: one of

```
[ ] ( ) { }
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
= *= /= %= += -= <<= >>= &= ^= |=
, # ## @ ? : ->
```

Phrase Structure Grammar

Expressions

primary:

constant

string-literal

(*expression* **)**

array-initializer

trap-call

identifier-or-call

array-initializer:

{ *element-list*$_{opt}$ **}**

element-list:

element

element-list **,** *element*

element:
  expression
  expression **:** *expression*

trap-call:
  [*type trap-selector trap-arguments*$_{opt}$ **]**

type: one of

Int8	**Int16**	**Int32**	**UInt8**	**UInt16**	**UInt32**	**Float64**
Int8Ptr	**Int16Ptr**	**Int32Ptr**	**UInt8Ptr**	**UInt16Ptr**	**UInt32Ptr**	**Float64Ptr**
MemPtr	**Void**					

trap-selector:
  expression

trap-arguments:
  trap-argument
  trap-arguments **,** *trap-argument*

trap-argument:
  type expression

identifier-or-call:
  identifier arguments$_{opt}$

arguments:
  (*argument-list*$_{opt}$ **)**

argument-list:
  expression
  *argument-list***,** *expression*

member-expression:
  primary
  member-expression **[** *subscript-expression* **]**
  member-expression **.** *identifier-or-call*

subscript-expression:
  expression
  expression **..** *expression*

postfix-expression:
member-expression
member-expression ++
member-expression **--**

unary-expression:
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator unary-expression
& *name*

unary-operator: one of
+ - * ~ ! # ## @ ^ typeof

multiplicative-expression:
unary-expression
multiplicative-expression ***** *unary-expression*
multiplicative-expression / *unary-expression*
multiplicative-expression **div** *unary-expression*
multiplicative-expression **%** *unary-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

equality-expression:
relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

and-expression:
equality-expression
and-expression **&** *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression **^** *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression **|** *exclusive-or-expression*

conditional-and-expression:
inclusive-or-expression
conditional-and-expression **&&** *inclusive-or-expression*

conditional-or-expression:
conditional-and-expression
conditional-or-expression **||** *conditional-and-expression*

conditional-expression:
conditional-or-expression
conditional-or-expression **?** *expression* **:** *conditional-expression*

assignment-expression:
conditional-expression
member-expression assignment-operator assignment-expression

assignment-operator: one of
= *= /= div= %= += -= <<= >>= &= ^= |=

expression:
assignment-expression
expression **,** *assignment-expression*

Statements

statement:

expression-statement
selection-statement
iteration-statement
jump-statement
var-statement

block:

{ *statement-list*$_{opt}$ }
statement

statement-list:

statement
statement-list statement

expression-statement:

expression$_{opt}$ **;**

selection-statement:

if (*expression* **)** *block*
if (*expression* **)** *block* **else** *block*

selection-statement:

while (*expression* **)** *block*
do *block* **while (** *expression* **)**
for (*expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *block*
for (*name* **in** *expression* **)** *block*

jump-statement:

continue **;**
break **;**
return *expression*$_{opt}$ **;**

var-statement:

var *variable-declaration-list*$_{opt}$ **;**

variable-declaration-list:

variable-declaration
variable-declaration-list **,** *variable-declaration*

variable-declaration:
    name variable-initializer$_{opt}$

variable-initializer:
    = *conditional-expression*

Functions

function:
    function *name* **(** *name-list*$_{opt}$ **)** *block*

name-list:
    name
    name-list **,** *name*