APScript

general purpose script language

Axel Plinge Software

29th July 2006

Contents

1 Introduction **3**

2 Language **4**
2.1 Types . . . 4
2.1.1 Basic Types . . . 4
2.1.2 Object Types . . . 4
2.2 Variables . . . 4
2.3 Expressions . . . 5
2.3.1 Addressing . . . 5
2.4 Control statements . . . 6
2.5 Function Definitions . . . 7
2.6 Built-in Functions . . . 8

3 Examples **10**

A Compiler **13**
A.1 optimizations . . . 13
A.1.1 expressions . . . 13
A.2 memory types . . . 13
A.3 machine types . . . 14

```
printf("Hello world!\n");
```

Code 1: a small program

1 Introduction

APScript is a versatile script language used in a number of our products. There are three basic types of script

- plain apscript files
- APScript for specific applications, using application-specific functions
- inline-scripting embedded e.g. in HTML

2 Language

The APScript language can be characterized by its elements:

2.1 Types

2.1.1 Basic Types

The basic types form the base of the language typing. A true constant is of one of the basic types.

null No type, an unassigned value of `NULL`

bool Boolean, either `false (==0)` or `true (!=0)`

integer Integer constants can be written as `12` or with different radix as `0b0010101`, `0h3f` etc.

float Floating point numbers. Constant should be written as `3.14` or `1.0`.

string A string is a character sequence. Note that there is no character type. Constants are written as `"Hello"` or `"a"`.

range* A range is an interval of integers. Constants are written as `0..3`.

time Timestamp in (GMT) seconds since midnight, 1.1.2002.

2.1.2 Object Types

single value A single value that is of any of the basic types.

array A data structure consisting of consecutively numbered objects with the index range `0..array.length-1`.

map A data structure where index values (any basic type) are mapped to objects.

2.2 Variables

The scripting language supports variables, all variables are soft types, a type conversion is done at rumtime if required. Consider code 2.

```
var a=2;
printf("a is %s\n",a);
```

Code 2: example of a variable

The printf function expects a string, so ist argument is converted to a string.
Ther is a number of modifiers for variables:

global The variable exists just once throughout the code.

	math		*conditional*		*unary*
+	add	==	equal	NOT	bitwise Inversion
-	subtract	!=	not equal	-	complement
*	multiply	<	less		
/	divide	<=	less or equal	not	logical inversion
	integer math	>	greater		
>>	shift right	>=	greater or equal		*adresssing*
<<	shift left			[ ]	object[integer]
AND	bitwise AND	and	logical and		object[range]
OR	bitwise OR	or	logical or		object[type]

Table 1: operators

static The variable exists just once, it is assigned with the initial value at compile time.

persistent The variable keeps its value after execution and recompilation. It is saved in an external xml file.

```
persistent var iRunP=0;
static var iRunS=0;

iRunP=iRunP+1;
iRunS=iRunS+1;
printf("This is the %s. time you run this.\n" ,iRunP);
printf("And the %s. time after compilation.\n" ,iRunS);
```

Code 3: example of persistent and static variables

2.3 Expressions

An expression is made up of operators and objects or functions. There are a number of operators, cp. table 1. For readability, some operators differ form C++/java, the cryptic operators (`!a^~b`) can be used as well if the compiler options are set accordingly.
Operator precedence is set as one would expect, so mathematical expressions can be writte as is, e.g. `-1*3.4/8.1+23.123` as can conditional expressions e.g. `a!=b or a!=c`.

2.3.1 Addressing

Indirect adressing is done using square brackets. Not that a range can be used on arrays and strings as well, so that substring operations can be done as in example 4.

```
var s="This is some text";
printf("%s\n",s[3]);
printf("%s\n",s[8..s.length-1]);
```

Code 4: examples of string addressing

String type objects support `.length` as special operator, array type objects support `.length`, `.dim` and `.size`. Note that the size operator returns the acutual number of initialized objects. Arrays are a combination of values addressed by a sequence of integers, starting with zero. Maps can be addressed by any type, such as strings or floats.

```
var a[];
a[0..10]=0;
a[1]=12;
a[2]="Hello";
a[8..9]=a[1..2];
```

Code 5: examples of array addressing

```
map m[];
m["zero"]=0;
m["more"]=12;
```

Code 6: examples of map addressing

2.4 Control statements

Expressions can be used to control the program flow. Instructions are made conditional using the `if` statement, cp. 7.

```
var a=2;
var b=3;
if a == b
{
    printf("%d==%d\n",a,b);
}
else
{
    printf("%d!=%d\n",a,b);
}
```

Code 7: example of a conditional expression

For fixed loops, the `forall` statement can be used.

```
forall i in [0..9]
{
    printf("%d ",i);
}
```

Code 8: example of a fixed loop

For arbitrary repeated actions, the `while`, `do` or `repeat` statement can be used, as illustrated in code snippet 9.

```
var i=0;
while i < 10
{
    printf("%d ",i);
    i=i+i;
}
```

Code 9: example of a while-loop

```
var i=0;
do
{
    printf("%d ",i);
    i=i+i;
}
while i < 10
```

Code 10: example of a do-loop

```
var i=0;
repeat
{
    printf("%d ",i);
    i=i+i;
}
until i > 9
```

Code 11: example of a repeat-loop

For looping over indices, the `foreach` statement is avaliable.

```
foreach i in a
{
    printf("a[%s]=%s",i,a[i]);
}
```

Code 12: example of index iteration

2.5 Function Definitions

For repeated actions, functions can be declared. A function is declared by the `function` keyword if it returns a value, or the `procedure` keyword if it does not. The keyword is followed by the function name and the function arguments in round brackets.
In example 13 a procedure is defined that prints its argument. Not that the conversion to string is done in the printf function.
A function returns a value. This can be any type that an object can be of.

`sleep(x)`	waits x ms
`gettick()`	returns actual tick count in ms
`gettime()`	returns timestamp (UTC seconds since 1.1.2002)
`localtime(t)`	returns array with local date for timestamp
`mktime(Y,M,D,h,m,s)`	returns timestamp for local date
`rand()`	random number
`srand()`	set random seed
`printf(...)`	formatted output
`sprintf(...)`	formatting

Table 2: system functions

```
function max(a,b) { if a>b return a; return b; }
function min(a,b) { if a<b return a; return b; }
function med(a,b,c) { return max(min(a,b),min(b,c)); }

printf("Median 1..3 is %s\n",med(1,2,3));
```

Code 14: example of functions

2.6 Built-in Functions

Using the `sleep` and `gettick` functions, execution time can be controlled.

```
var i=0;
while i < 10
{
    printf("%d ",i);
    i=i+i;
    sleep(100);
}
```

Code 15: example using sleep

```
procedure printvar(var)
{
    printf("%s\n",var);
}

printvar("Hallo");
printvar(12);
printvar(3.1415);
```

Code 13: example of a procedure

`abs`	absolute value $\mid\mid$	`sin`	$\sin()$
`sgn`	signum (sign) $\operatorname{sgn}() \in \{-1, 0, +1\}$	`cos`	$\cos()$
`floor`	lower full number $\lfloor\ \rfloor$	`tan`	$\tan()$
`round`	rounded, $\lfloor +0.5 \rfloor$	`cot`	$\cot() = 1/\tan()$
`ceil`	upper full number $\lceil\ \rceil$	`asin`	$\operatorname{asin}()$
`ln`	logarithm base e, $\ln()$	`acos`	$\operatorname{acos}()$
`ld`	logarithm base 2, $\operatorname{ld}()$	`atan`	$\operatorname{atan}()$
`log`	logarithm base 10, $\log()$	`acot`	$\operatorname{acot}()$
`sqrt`	$\sqrt{}$	`rand`	a random integer value

Table 3: mathematical functions

```
var i=gettick();
while gettick()-i < 100
{
    printf(".");
}
printf("waited approx 100ms.\n");
```

Code 16: example using gettick

```
procedure dump_stm(m)
{
    printf("%02d.%02d.%04d %02d:%02d:%02d\n (map:",
    m['mday'],m['mon'],m['year'],m['hour'],m['min'],m['sec']);
    foreach i in m
    {
        printf(" %s = %s",i,m[i]);
    }
    printf(")<br />");
}

procedure dump_tm(t) {
    dump_stm( localtime(t) );
}
```

Code 17: example using localtime

There is a list of mathematical functions which are available, cp table 3.

`isNum()`	is a number, also true for ”12.3”
`asNum()`	as number, eiterh fixed or floating point
`asInt()`	as nonfractional number
`asFlo()`	as floating point number
`asStr()`	as string
`ASCII(x)`	returns the ASCII code for the first charater of x
`asChar(x)`	returns the a string with the character corresponding to ASCII code x

Table 4: type functions

```
function binfmt(i)
{
    var s;
    s = ceil(ld(i+1));
    s= "%"+s+"b";
    return s;
}
procedure printbin(i)
{
    printf(binfmt(i),i);
}
printbin(12);
```

Code 18: using mathematical functions

To querry or influence the types of script objects, there is a number of special functions.
The `printf` and `sprintf` functions supports a number of automated formattings, cp. table 5.

```
function asall(a)
{
    return sprintf("%s => %l %b %d %h",a,a,a,a,a);
}

printf("%s vs. %s\n",asall(1),asall(true));
printf("%s vs. %s\n",asall(12),asall(12.3));
```

Code 19: example of sprintf

3 Examples

Since having read a lot of examples usually serves best for learning a language, here are some more examples.

`%s`	as string
`%Sh`	string (HTML conversion (”<ä” → ”<ä”)
`%Sc`	string with C-constant conversion (”c:\” → ”c:\\”)
`%Sf`	string with URI conversion (”c d” → ”c%20d”)
`%d`	decimal
`%#d`	decimal with # digits
`%D`	decimal with dots (e.g. ”1.024”)
`%x`	heaxdecimal
`%#x`	heaxdecimal, first # digits (`printf("%02x",0x123)` → ”23”)
`%b`	binary
`%#b`	binary, first # digits
`%f`	floating point
`%#f`	floating point with # digits before dot
`%#1.#2f`	floating point with #1 digits before, #2 after dot
`%l`	logical, either true or false
`%t`	time, e.g. ”11.03. 13:15”
`%Td`	time, day only (e.g. ”11.03.”)
`%Ts`	timespan, e.g. ”1m13s”

Table 5: printf formatting instructions

```
function max(a,b) if a>b return a; return b;
function min(a,b) if a<b return a; return b;

procedure println() printf("\n");
```

Code 20: popular macros

```
var s;
s="Hallo";
s=s+" Welt!";
printf("%s\n",s);
s[1]="e";
printf("replacing '%s'\n",s[6..10]);
s[6..11]="World!";
printf("%s\n",s);
s[0..5]=s[6..11];
printf("%s\n",s);
```

Code 21: using strings

```
procedure printArray(a)
{
    var i=0;
    while i < a.length
    {
        if i printf(", ");
        if a[i].dim
        {
            printf("{");
            printArray(a[i]);
            printf("}");
        }
        else
        {
            printf(a[i]);
        }
        i=i+1;
    }
}
```

Code 22: using arrays

A Compiler

A.1 optimizations

A.1.1 expressions

When parsing a expression, the compiler will summarize all constant objects, that are numeric constants as '3.1415' or variables that are not assigned more than one value. Cosider example 23: Both variables are constant, as is the number, so the compiler would generate the code for **return** 14;.

```
var a=2;
var b=3;
return a+b*4;
```

Code 23: variables treated as constants

Long fomulars show the full scale of the optimizations done.
For example, '1-1*b+a/2.0+9/3' will be compiled as '-2+b+a*0.5'.
All constants within a sum or product are sumarized. Thus, '1+2+3+4' would be treated as '10', '1*2*3*4' as '24'. Inefective operations such as '+0' or '*1' are omitted. Remeber that the compiler makes no differece for once-assigned variables. Subtractions and divisions are considered more costly than addition and multiplication, so formulars are rewritten accodingly. For example, 'a-b-c-d' would be rewritten as 'a-(b+c+d)'. To further avoid divisions, a constant division is compiled as a constant multiplication, e.g. 'a/2.0' as 'a*0.5'.
When dealing with many objects combined with the same operator, minimal binary operator trees are generated (cp. fig. 1). This leads to minimal stack / register usage on the target machine. Operand order is maintained by filling the tree from left to right, starting at the bottom level.

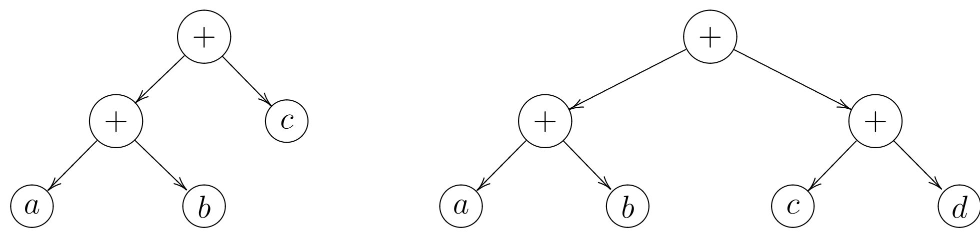

Figure 1: operator trees

A.2 memory types

calculation stack A stack for calucaltion results.

Calculations work in postfix order, e.g. '2*4+3' = '2 4 mul 3 add' which causes the sequence 'push 2; push 4; push (pop*pop); push 3; push (pop+pop);'.

localvars An address space that is increased at function entry and decreased on exit.

Thus, 'function fun() { var a=1,b;...' Will cause a sequence in the form 'lvp+=2; lv[lvp-1]=1;'.

```
      block  ::=  statement { ';' statement } .
  statement  ::=  '{' block '}' | whilestate | ifstate | assignment .
 whilestate  ::=  'while' expression statement .
    ifstate  ::=  'if' expression statement [ 'else' statement ] .
 assignment  ::=  var '=' expression .
 expression  ::=  lop [ ( '&' | '|' | '==' | '!=' | '<' | '<=' | '>' | '>=' ) lop ] .
        lop  ::=  [ ! ] shift { ( '&&' | '||' | '^' ) shift } .
      shift  ::=  sum { ( '>>' | '<<' ) sum } .
        sum  ::=  product { ( '+' | '-' ) product } .
    product  ::=  power { ( '*' | '/' ) power } .
      power  ::=  atom [ '**' atom ] .
       atom  ::=  number | var | '(' expression ')' | function '(' expression ')' .
     number  ::=  ( ['-'] ['0' ... '9'] { '0' ... '9'} '.' { '0' ... '9'} ) |
                  ( 0x ['0' ... 'F'] { '0' ... 'F'} ) .
        var  ::=  'a' ... 'z'{ '0' ... '9' | 'a' ... 'z'} .
```

Table 6: Grammar in EBNF

call stack A stack containing the reture address of a function call.

Thus, a function call causes a sequence in the form `'CS.push(PC);PC=call_address'` and a return causes `'PC=CS.pop()'` to happen on the machine.

A.3 machine types

stack There is just one stack for both computation, function arguments and local variables. That means that local variables are addressed on the stack with a changing offset.

stack, localvars There is calculation stack and a local variable "stack". Local variables have a fixed offset there within a function.

localvars There is just the local variable space, all temporary calculation results end up there.

Table 6 shows the grammar used to make up the language.

List of Figures

1 operator trees 13

List of Tables

1 operators 5
2 system functions 8
3 mathematical functions 9
4 type functions 10
5 printf formatting instructions 11
6 Grammar in EBNF 14

List of Code

1 a small program 3
2 example of a variable 4
3 example of persistent and static variables 5
4 examples of string addressing 5
5 examples of array addressing 6
6 examples of map addressing 6
7 example of a conditional expression 6
8 example of a fixed loop 6
9 example of a while-loop 7
10 example of a do-loop 7
11 example of a repeat-loop 7
12 example of index iteration 7
14 example of functions 8
15 example using sleep 8
13 example of a procedure 8
16 example using gettick 9
17 example using localtime 9
18 using mathematical functions 10
19 example of sprintf 10
20 popular macros 11
21 using strings 11
22 using arrays 12
23 variables treated as constants 13

Copyright Axel Plinge Software 2001-2006