

GNU

HP Compiler Language Description

Željko Vrba

Zagreb, 01.01.2000.

1	Introduction	2
2	Types	3
2.1	Real numbers	3
2.2	Integer numbers	4
2.3	Strings	4
2.4	Binary numbers	5
2.5	Character	5
2.6	Complex numbers	5
2.7	Arrays	6
2.8	Lists	6
2.9	Supported conversions	7
3	Expressions	7
3.1	Using stack in expressions	8
3.2	Other stack operations	9
4	Control structures	9
4.1	Conditionals	9
4.2	Definite loops	10
4.3	Indefinite loops	11
5	Procedures, functions and programs	11
6	Important notes	12

## 1 Introduction

First of all, excuse me for poor English because it is not my native language.

I came to an idea of writing a high-level language for HP because I was very unsatisfied with the existing languages and development tools:

- SystemRPL has over a thousand names, many of which are very similar, but only a small subset of them is used in actual programs. Because of their similarity I often had to look up in the manual which one I really needed.
- HP and GNU Tools are very user un-friendly in the sense that for every compiled program you need several steps to get an executable (RPLCOMP, SASM, SLOAD).
- Port of GCC which generates machine code for the Saturn architecture is almost useless: the generated code is excessively big, two additional libraries are needed on the HP for the code to run and many things can't be done easily from within that restricted C language (like stack manipulations etc.)
- RPL++ is very buggy (I have had to make several modifications to the sources before I got it to run).
- Pascal developed by one of the people in HP team is distributed for DOS only, without sources (and sincerely, I haven't tried it because of that).

In designing this language I have had several concepts in mind:

- The language should support procedural paradigm with “normal” prefix/infix syntax.
- The language should be strictly typed since SystemRPL is strictly typed (for example, there are different operators for adding binary integers, reals or hexadecimal strings). Using an operator on wrong types results, at best, in wrong results (if not in calculator warmstarts or memory clears).

- The language should support direct stack manipulations, since they are much more efficient (based on my present experience with UserRPL) than “traditional” local variables. However, there are pitfalls.
- There should be no (or very few) things that compiler does automatically: these things are not clearly visible in the source, but they can result in significant performance losses. As in SystemRPL, *every operation must be explicitly coded*. However, there are a few exceptions and they will be documented later in this manual.
- Automatic compilation: the user should not be burdened with manual compilation of resulting SystemRPL program and writing control files for SLOAD.

I think that I’ve accomplished all the goals and the generated code is reasonably good. However, it could be made smaller (hence faster) if someone would write a **code optimizer** (this work is in progress).

One final note: PLEASE send bug reports every time you get unexpected behaviour from compiled program (it may be a feature, not a bug, but report it anyway). A program source and a short description of what it should do (and, of course, description of *what it does do instead*) would be very helpful in reporting bugs. *Be sure to read the last section to find about not-yet implemented features.*

If you get a crash<sup>1</sup>, please mail the following to me:

- the source file you were compiling
- a complete error message
- system type and version you are running (`uname -a` on UN\*X)
- compiler version and used switches

The email you can reach me at is:

`mordor@fly.srk.fer.hr`

This is intended for bug reports, wishes, suggestions and help to further develop thins program. See the last section about things to do.

Throughout the text all keywords will be underlined. The language is *case-insensitive* to both keywords and identifiers. The following sections describe elements of the language. *Familiarity with RPLCOMP syntax (either from HP or GNU Tools), and RPL programming in general, is assumed.*

## 2 Types

This section describes all supported types and operations on them and discusses some performance issues. **Table 2** summarizes operators by groups and explains their semantic. **Table 1** summarizes possible conversions between types and explains conversion semantic.

### 2.1 Real numbers

This is ordinary real number used in UserRPL calculations. For a number in the program to be recognized as real, it *must be followed by a decimal point*. Here are some examples:

---

<sup>1</sup> A crash is when the message says something about “internal compiler error”; other messages are normal error/warning messages

```
12.  -11.3  11.e-6  -17.48e4  0.
12  -11   -3   0
```

All numbers in the first line are all valid reals (note that 0 also must be written with a point), and no number in the second line will be interpreted as real (because there is no decimal point). Real numbers are declared as **real**.

There is also an extended real number type: **long real**. This is analogous to the System RPL object. There is no way to write a long real constant in source, except with explicit cast like:

```
long real 12.
```

A code optimizer should optimize this to a single extended real object (the code that is originally generated is real object and a cast to extended real.) Also, the optimizer should optimize away negative real constants: they are assembled as a positive constant and a **CHS** operation.

The following operators work on real and long real numbers: **++\*/%**, and unary minus **-**. They have the same meaning and precedence as in C.

## 2.2 Integer numbers

This is system binary integer (5 nybbles in size), and is declared as **natural**. Natural constants can be written either in decimal or hexadecimal notation, see this example:

```
12   24   36   7   19   256
0xC  0x18 0x24 0x7 0x13 0x100
```

You should note that division (and also taking the remainder) of naturals takes *two* operations, not one. That is because **#/** operation returns two objects on the stack: remainder and quotient, and one of them must be dropped (depending on operation). When doing division, this is done with one supported operation (**SWAPDROP**) instead of two separate operations.

All operators are supported as with real numbers, *except unary minus*. Unary minus doesn't make sense (and doesn't exist in System RPL, either), since binary integers are *defined* to be positive.

Beware when comparing naturals: comparison like **a < 0** (where **a** is natural variable or expression) *will never return true* since naturals are always positive. Compiler will *not* issue a warning in this case.

## 2.3 Strings

This is user-level string (" object delimiters). It is declared as **string**. In a program source, everything between double quotes (") is considered a string literal. A string literal *cannot contain newlines or other double quotes*. Here is an example of string literal:

```
"this is A string\20with escape"
```

**\20** is an **escape sequence**: the two character following the backslash will be interpreted as hexadecimal ASCII code of the character. That way it is possible to insert characters like newline, double quotes etc.

There is also a special string constant: the null string, which is entered as

```
null string
```

There are no built-in operators for strings. You should use the facility for defining **external** procedures and functions that work on strings (take the needed addresses from some System RPL reference).

## 2.4 Binary numbers

This is user binary integer type (**#** delimiter). It is declared as **binary**. Here are a few examples of binary numbers: (everything after **#** is interpreted as *binary* number):

```
#12AC3 #1884 #1999
```

All operations are supported, as with real numbers. Using the unary minus on a hex string will produce two's complement. Additionally, there are three logical operators: **&**, **|**, **~**, with their usual C semantic: bitwise and, or and not (1st complement). When comparing with zero, the same warning applies as for natural numbers.

There are also five bit-shift and rotate *prefix (unary)* operators: **shl**, **shr**, **asr**, **rol**, **ror**.

## 2.5 Character

This object represents 8 bit quantities; it is the System RPL object, and is declared as **char**. It can be entered in either of two ways:

```
'A'  
'\41'
```

The second form includes a backslash and a *hexadecimal code* of character (in this case, 41 in hexadecimal is 65 in decimal, which is ASCII code of the letter A). There are no operations defined for this type.

## 2.6 Complex numbers

This is user-level complex number. It is declared as **complex**. There are two constructs for construction of complex numbers. Complex numbers can be constructed *from one or two real numbers*.

```
complex 2.  
complex(2., 3.)
```

The first form converts a single real number into a complex number with imaginary part 0. The second form makes complex number  $2 + 3i$ . Complex numbers need not be constructed only from real constants; in place of 2. and 3. you can use arbitrary expressions with real results. There is also a **long real** type, which is a system object.

Writing complex constants is, in effect, casting real to complex numbers, thus producing more operations than necessary. Code optimizer should fix this.

All operators are defined as for real numbers, *except* for modulus (**%**) which isn't defined for complex numbers. Additionally, there are two more unary operators: **re** and **im** which take the real (imaginary) part of the following complex expression. They are at the same precedence level as unary minus, and nonassociative, so you will *have to* use parentheses in any combination of those operators and unary minus.

## 2.7 Arrays

There are three ways to declare an array:

```
var a : array of real;  
var b : array [3] of real;  
var c : array [3, 2] of complex;
```

*Only complex and real types are allowed as array elements.* Long real and complex elements are not allowed! The first form is used to declare an array of unknown dimensions. It is mostly useful to declare external procedures/functions which take arrays as arguments. *This form can't be used to declare actual variable* (the one that is not a formal parameter within a procedure or function declaration).

The second and third form are used to declare one- and two-dimensional arrays. Upon entry to subroutine, array will be filled with zeros. Note that *array indexing is 1-based* (the first element has index 1), and arrays are stored *by rows* (just as in System and User RPL).

The access to array elements is same when dealing with both 1- and 2-dimensional arrays (indexing can be done *only with natural expressions*):

```
b[2] = 2.  
c[5] = complex 4.
```

The first line references 2nd element in vector b; the second line references (3,1) element of array. This is the calculation from 2-dimensional position (row, column) to 1-dimensional index:

$$index = (row - 1) \cdot nc + column$$

where *nc* is number of columns in array. Note that there is *no range checking* done on indexes.

Storing elements in an array is currently very inefficient, and I don't see a (simple) way to make it more efficient. So try not to build your arrays element by element; it is much better to put all elements on the stack, and then create an array from them.

I deliberately didn't add two-dimensional indexing because the innocent-looking statement like

```
a[2, 2] = 4
```

would produce much SystemRPL code (this way it's much clearer that arbitrary element access is pretty expensive operation). No operators are defined for arrays. Use ROM routines.

## 2.8 Lists

List is a heterogeneous collection of elements. So, when declaring a list, we don't have to specify a type of elements (like we had to with arrays):

```
var a : list;
```

There is also a special list constant: the empty list, which is written as

```
null list
```

No operators are defined for lists. Use ROM routines.

## 2.9 Supported conversions

Expressions of one type are converted to expressions of another type with a **cast operator**. Casting is done by writing the name of type in front of an expression, like this:

```
real 2
natural (a+b-c)
natural a+b-c
```

The first line shows another way of writing a real constant in programs. *Don't use this way of writing real numbers in programs, since it is slower and larger: this sequence is translated into following SystemRPL commands:*

```
# 2
UNCOERCE
```

instead into one if the `real 2` were replaced with `2.`. Code optimizer should make this issue irrelevant.

Cast operator is at the *same precedence level as unary minus*. This means that, in line 2, the result of whole expression inside parentheses will be converted to natural. However, in line 3, *only a will be converted to natural*. **Table 1** summarizes possible conversions. These are the only conversion pairs that exist.

Source type	Possible destination types
natural	real, string, binary, char
real	natural <sup>1</sup> , string <sup>2</sup> , binary <sup>1</sup> , complex <sup>3</sup> , long real
binary	natural, real, string <sup>4</sup> ,
char	natural, string
long real	real
long complex	complex

**Table 1** Supported conversions

1. When converting from real to natural (or binary), non-integral values will be rounded, and negative values will be replaced with zero
2. Real will be converted to string using current display mode
3. There are two possible conversions to complex type, see description of complex
4. Base character will not be appended

## 3 Expressions

In expressions, *all operands must be of the same type*. If they are not, cast operator must be used (see **table 1** for available casts). **Table 2** lists all operators and their precedence; operators in the same row have the same precedence.

Operator	Meaning	Precedence
- re im	unary minus <sup>1</sup> ; real and imaginary part	highest <sup>2</sup>
* / %	multiplication, division, modulus <sup>3</sup>	
+ -	addition, subtraction	
shl shr rol ror asr &   ~	bit operators	
< > <= >= == !=	relational operators <sup>4</sup>	
not	logical negation	
and and-then	logical <sup>5,6</sup> and	
or or-else	logical <sup>5,6</sup> or	lowest

**Table 2** Operators

- Unary minus doesn't work with natural type.
- All three operators are *non-associative*, so parentheses must be used if they are combined in expressions like `- re a`.
- % works only on naturals and reals. Executed on two reals it executes the MOD command (see the HP48 manual).
- Relational operators work on all types except booleans. Greater, greater-than, less and less-than operators aren't defined for complex operands. Both compared operands must be of the same type. != is the "not equal" relation.
- Logical operators work only on boolean types. The only way of producing a boolean-typed expression is by using one of the relational operators.
- Short circuit operators work like in C. With **and**, if the left side evaluates to false, the right side is not evaluated. Similar with **or**, if the left side evaluates to true, the right side is not evaluated. Be careful: in combinations like **a and then b and c**, *c will be evaluated no matter what*.

Expressions are constructed as in C using the listed operators. Parentheses (()) may be used freely to alter the normal operator precedence. Operators in the same row have the same precedence. All operators are *left-associative* unless stated otherwise. Assignment is done with = operator, like

```
a = 2;
```

### 3.1 Using stack in expressions

Direct stack manipulations are more efficient than using local variables, so I've decided to put them in the language. The most important thing to realize is that all expressions are evaluated on the stack, and that results are leaved on the stack: there is always something on top of the stack. This top of stack can be accessed with the "top of stack" operator/expression: (). This expression *has no associated type* and a cast operator must always be applied to get a valid expression. So, you must be *very certain* about the type of the item on top of the stack. If you are wrong, all sorts of bad things can happen (wrong results at best, but resets and memory clears are also possible).

This breaks the type system of the language, but there is no other way around: the compiler can't predict the flow of the program or the user input. So you, the programmer, has to tell what type is on top of the stack.

You can best understand the working of the () operator by considering that, essentially, it *compiles into a no-operation*. So, the statement

```
() = 13.;
```



is effectively a *push* operation. First, the simple expression `13.` is executed and its result is left on the stack. Since the `()` operator compiles into nothing, the real 13 is pushed on the stack. This is the *only* case where the `()` can be used as an “lvalue” (in C terminology).

Here are some other examples. Suppose the following two statements:

```
() = 12; a = natural () / 4;
() = 4; a = 12 / natural ();
```

The first statement will store the result  $12/4 = 3$  into variable `a`. But the second statement will store the result  $4/12 = 0$  into `a` (integer division). First the 4 is put onto the stack, then 12 and then division is performed (recall that `()` is a no-op). All expressions are evaluated in this manner (by conversion to reverse polish notation, RPN). With commutative operators (`+``*`) both expressions would give the same result.

## 3.2 Other stack operations

**Table 3** lists other stack operations. All of them can be used in expressions, as well, as stand-alone statements. If used in expressions, cast operator must be prepended and `()` suffixed. This will use the item that is on top of the stack after the operation is performed. Operations that can work on variable number of elements are suffixed with `:n`.  $n$  is any natural-typed expression.

Operation	Explanation
<code>over</code>	copies level2 to top
<code>dup</code>	duplicates top of stack
<code>swap</code>	swaps the top two elements
<code>rot</code>	like <code>roll:3</code>
<code>drop</code>	drops the top of stack
<code>roll:n</code>	rotates up $n$ items
<code>rolld:n</code>	rotates down $n$ items
<code>:n</code>	picks the $n$ -th element

**Table 3** Stack operations

Some examples:

```
a = natural depth();
a = natural roll:4();
a = natural :3() + natural :5();
:3;
rolld:5;
```

## 4 Control structures

This is a summary of control structures. Keywords are underlined.

### 4.1 Conditionals

```
if expr then
    statements
```

```

else
    statements
endif

```

This is the traditional if-then construct. The else part can be omitted. **expr** must be an expression of boolean type.

```

case
    expr1 then
        statements1
    end ;
    .
    .
    .
    expr_n then
        statements_n
    end ;
end

```

This is a counterpart of **switch** in C or **case** in Pascal. However, executing one block of statements exits the whole **case** statement (no need for break, like in C). **expr\_i** must be an expression of boolean type. If you need some kind of default behaviour, just put default for an expression. There can be as many test/statement pairs as you wish. The semicolons after end that ends the then action inside **case** statement are obligatory!

## 4.2 Definite loops

```

start expr1 , expr2 do
    statements
next

start expr1 , expr2 do
    statements
step expr3

```

All expressions must be of natural type. The first form iterates a loop from **expr1** to **expr2** with counter increments of 1. The second form increments counter by **expr3**.

The loop counter value can be accessed by the @0 expression. In nested loops, the inner loop's counter is accessed with @0, the outer's is accessed with @1. If there are more than two start loops nested, then only the counters of the last two can be accessed (there are no operators like @2 etc.). @0 and @1 have natural type.

The break can be used to break from the loop. However, the loop will be exited only at its step or next, and no sooner.

Don't rely on break! I'm thinking of replacing it with an **exit-if** type of statement altogether. The **exit-if** statement would exit the current nesting level (current procedure, if statement, etc). The difficulty is that break could be buried in many levels of if statements. Every if statement adds a level on the return stack. The compiler would have to track how many items on the return stack should be removed; after removing them from the return stack, other complicated manipulations of the return stack would be required to actually exit the loop. **exit-if** statement would just leave the current grouping, be it the current procedure, some loop or if statement.

### 4.3 Indefinite loops

```
do
    statements
until expr

while expr repeat
    statements
end
```

These are the familiar do-while and while loops from C. `expr` must be of boolean type. `break` is currently unimplemented for indefinite loops.

## 5 Procedures, functions and programs

Procedure and function declaration follows that of Pascal. Here are some examples without further explanation:

```
function sln$(a : Binray; n : Real) : Binary;
function random : Real;
procedure Test;
```

The last two lines illustrate declaring a function/procedure that take no parameters. Note that `$` is allowed in function and variable names. Here is a complete example of a function that takes a binary and a real and shifts the binary number appropriate number of bits to the left.

```
function sln(a : Binary; n : Real) : Binary;
var Result : Binary;    -- local variable
begin
    Result = a;
    start 1, Natural n do
        Result = shl Result;
    next;
    () = Result;        -- returning value from a function
end;
```

This example illustrates several things: how comments are introduced in the source, local variables and returning values from a functions. Leaving the value on top of the stack is the *only* mechanism for returning values from functions (remember that you are working with a stack-based language). That way you can make a function that returns *multiple* values. In that case, declare the return type of the function as the type of the last value pushed on the stack.

The compiler *does not* check that you actually leave a value on the stack, nor that it is of the same type as the return type of the function. Here is the same example without using local variables, only stack:

```
function sln(a : Binary; n : Real) : Binary;
begin
    () = a;
    start 1, Natural n do
        () = shl Binary ();
```

```

        next;
    end;

```

Procedures and functions do not check that they receive the proper number and type of parameters. This is unacceptable for user programs. This makes a user version of the `sln` program:

```

program usersln(2);          -- expect two arguments
begin
    binary, real : sln      -- dispatch on argument types
end;

```

This program, when executed, will check that there are two arguments on the stack and that they are of types `binary` and `real`. `real` argument is on top of the stack. After `:` comes the name of the procedure/function to be called if the arguments match. The compiler checks whether the dispatch type list and the specified procedure argument lists have matching types. *You can't put any statements or variable declarations within program; only dispatch lists are allowed.*

You can also declare *global* variables: just declare them outside any function or procedure. However, you'll have to ensure their existence some other way (if the program doesn't create them, but relies on their existence).

ROM routines can also be used: here is an example of declaring a ROM routine for adding two `real` arrays:

```

external function ArrayAdd(
    a : array of real;
    b : array of real) : array of real at 0x1234;

```

0x1234 is the (fictious) address of ROM routine. Now you can add two arrays with this function. When this function is called, an internal ROM routine will be executed. Get System RPL reference to find out real addresses of useful ROM routines.

## 6 Important notes

This is the syntax for invoking the compiler:

```
hpc [OPTIONS] source
```

`source` is the source file to compile. I use the extension `.p`, but you can use any other you want to. Option description:

**-e entries** specifies the full path and file name of the alternate entry point list. By default, this list is loaded from path determined at build time. If `HPCLIB` environment variable is set, then `$HPCLIB/entries.lib` is loaded. **-e** option overrides both, the compiled-in path, and the environment variable setting. The compiler will exit with an error if the entry point list can't be found or doesn't contain required entry points (displaying the required entry) (and **-t** option is not specified, see below).

**-i include** specifies the full path to the alternate compiler library. The search rules are the same as for **-e** switch, only that the file is `hpc.lib`. It is not an error if the specified file can't be found. This is a way to avoid loading of library file: just specify a nonexistant file, or `/dev/null`.

**-o** flag tells the compiler to perform the optimization step. This is intended for tracking bugs in the optimizer. Once the optimizer is stable, this flag will be ignored (for compatibility reasons; optimization will always be performed, regardless whether the switch was specified).

`-t` flag tells the compiler not to generate the binary output, but the resulting System RPL code. This is useful for debugging the compiler, or the System RPL to binary compilation. The generated file will have `.s` extension (the usual convention for System RPL sources). If this flag is specified, then the entry point list is not required.

`-v` tells the compiler to describe its actions; this is useful for tracking bugs related to path searching.

**Note:** The `HPCLIB` variable must specify a directory, and it must be exported; in Bourne-like shells this is done like this:

```
$ HPCLIB = /home/zax/hpc
$ export HPCLIB
```

This distribution comes with the entries file `entries.lib`, which is located in the source tree. So the simplest command to compile a program into executable form (assuming that `HPCLIB` is set) is

```
hpc sln.p
```

The compiler will output *one file per function/program/subroutine*. The name of this file will be the same as the name of the subroutine, but converted to uppercase, and optionally appended with `.s` suffix when the `-t` flag is given (see below). So the `sln` function from the above example will be output to the file `SLN` or `SLN.s`. If you haven't specified the `-t` flag, the resulting file(s) can be downloaded to HP and tried out.

Before reading in the actual source, the compiler will look for the file `hpc.lib` (see the description of `-i` flag). If it exists, it will be automatically read in and compiled before the actual source. This feature is intended for declaring frequently used ROM routines, so that you don't have to retype them at the beginning of each program you make. The sample `hpc.lib` comes with the compiler distribution. The addresses are taken from various System RPL references for the HP48.

**Note for HP49 users:** The `entries.lib` and `hpc.lib` that come with this distribution are ROM addresses for the HP48. This compiler can be used to compile the programs for HP49 without change, but you will need to get the HP49 entry points. For further reference about meaning of those entries, see the "System RPL Manual" from HP, or Edouardo Kalinowski's "Programming in System RPL" (better, with more entry points).

Things to do and things to watch out for (in random order):

- A better manual and more examples. I have some ideas about examples, but no time to write them. The same about the manual.
- Add more optimizations (see `optimize.c`).
- More functions in `hpc.lib`. There are currently only a few real functions and some procedures.
- Better error messages. In the case of syntax error, all you get is the mysterious "parse error" message and the line number, with no other explanation. Other error messages could also be more explanative.
- Escape sequences in strings and characters should be implemented (see `pascal.1.`)
- A mechanism for storing elements in an array that is on the stack, and not stored in some variable (the former is more efficient): allow indexing of any expression, and then checking whether this expression really is of array type.

- Document the use of `typeof` operator (see `pascal.y`) and maybe alter its semantic (see `codegen.c`).
- Implement `break` for indefinite loops.
- Change so that the `break` in `start` loops is effective immediately.
- Check the correctness of the generated code, especially for short-circuit boolean operators, and `break` statements.
- Implement switches for the desired type-cast convention (e.g. whether to append base character etc.), and issue a warning only when relying on the default behaviour, without any switch.

So this is the list of the things you can help with. If you have any ideas, and want to help, please send me a mail. I'll be happy to answer all your questions.