

Towards a Pascal compiler for HP RPN calculators

YUAN, Feng
(Hewlett Packard Singapore)
email: yuanfeng@hpsgrt1.sgp.hp.com

Paper submitted to PROMPT HP-GC Anniversary Conference 1994

ABSTRACT

This paper describes an ongoing experiment to compile Pascal program into system RPL source code, which is a high level language used to build the HP RPN calculators.

1. Introduction

Despite the popularity of HP RPN calculators like HP 48SX and HP 48 GX, the ability to program HP calculators in user RPL or system RPL is still regarded as something only the super users could do.

System RPL (reverse polish lisp) and user RPL are hard to programming for several reasons:

- 1) Major differences to other common programming languages like Pascal, C or Basic, although user RPL has done a much better job.
- 2) Lack of books, manuals, training courses, source code reservoir and knowledgeable friends on RPL.
- 3) Difficult to debug user RPL programs, not to mention about system RPL programs, for the end users.
- 4) Hard to modify program because of the close interdependency of RPL codes.
- 5) Lack of compile time syntax checking such as variable definedness, type compatibility, number of operands or parameters, which leaves more errors to the debugging phase.

Technically speaking, RPL is an intermediate code that works on a software emulated stack machine. RPL has lots of cousins: for example, Forth which is one of the predecessor of RPL, P_code which is used by several Pascal compilers, HPcode which is used by HP in many of its compilers. For this reason, the author has chosen to work on porting the P4 Pascal compiler to generate system RPL source code.

The P4 package is a public domain portable Pascal compiler and P_code interpreter written by Urs Ammann and others from ETH Zurich and fully documented in S. Pemberton's book titled "Pascal Implementation". The P4 compiler has been substantially modified to generate system RPL code.

The following sections will describe the details of compiling Pascal source code into system RPL code, familiarity with both languages are assumed.

2. Local Variables, Procedures and Functions

One of the design objective is that we must produce system RPL code that can merge with hand written system RPL code as smoothly as possible. That is to say we should be able to call hand written code and being called by hand written code. One big obstacle to this is variable storage and access.

P_code machine stores local variables on the stack. Each procedure call creates a standard stack frame, a chunk of parameters and local variables. The stack frame contains 5 elements for return address, function return value, base address, last base address and maximum stack extent. To access a variable, you must specify the static block level and offset within the block. During runtime, there is a MP register which points to the first variable of the current block. To access a local variable, adding offset to MP will get the address. To access variables defined in outer block, you get its base address from the stack frame chain.

For example, the following example shows a simple Pascal procedure and the corresponding P_code:

```

procedure p(i,j: integer);
  var k: integer;
  begin
    k:=i+j
  end;

ent1    8      p needs 5+2+1 for stack frame, parameters and locals
ent2    7      p will need another 7 stack levels for computation,
               not so accurate
lodi    5      copy parameter i to top_of_stack
lodi    6      copy parameter j to top_of_stack
adi     7      integer addition
stri    7      store into local variable k
retp                    procedure return

```

In system RPL, there is no MP register and stack frame chain (except a return address that is pushed on the return stack), so we can't generate 5 element stack frame and use (block level, offset) based addressing.

Nevertheless, because Pascal is a strong typed language, for local variable access, we know for sure during compile time how many elements are there on the (local) stack. We can then use stack operations to fetch and store variables, similar to what system RPL programmers are doing. For the example shown above, we know for sure when entering routine p, two elements would be on the stack: i on stack level 2 and j on stack level 1. So it can be compiled into the following system RPL code:

```

NAMELESS proc_3
::      (2) standard RPL routine entering
      ZERO      (3) reserve space for local variable k
      3PICK     (4) i is three levels away
      3PICK     (5) j is now three levels away too
      #+       (4) binary integer addition ( integer is implemented as
               unsigned integer now, beware )
      SWAPDROP  (3) store into location for k
      3DROP     (0) remove i,j,k from stack
;         standard RPL routine quitting

```

The numbers enclosed in parenthesis indicate the current (local) stack levels. It starts with 2 for 2 parameters, incremented as you reserving space for k, loading i and j, decremented as you adding or storing and finally cleared before you quit. While the original P4 compiler does maintain a stack level during compilation, it's not updated in some cases.

Stack level is very critical for compiling into system RPL, the P4 code is modified to maintain exact stack level. As a side note, because system RPL code uses lots of relative fetching and storing words like 3PICK, ROT, UNROT and 4UNROLL3DROP, system RPL code is hard to write, understand, debug and modify. A Pascal or C programmer seldom has to worry about the runtime address of a variable.

Compiling a Pascal function would be a little more complicated in handling return value. To following example illustrates this:

```

function p(a,b,c: real): real;
begin
  p:=sqrt(b*b-4.0*a*c)
end;

NAMELESS proc_3
::
  ZERO          (3)
  3PICK          (4) reserve space for return function value
  4PICK          (5) b is three levels away
  %*             (6) b is four levels away now ( DUP would be better )
  % 4.0          (5) b*b
  6PICK          (6) direct constant ( not smart enough to use %4 )
  %*             (7) a
  4PICK          (6) 4.0*a
  %*             (7) c
  %-             (6) 4.0*a*c
  %SQRT          (5) b*b-4.0*a.c
  XY>Y           (5) real square root
  4UNROLL3DROP  (4) store as function return value
  (1) retain top of stack, drop 3 levels under it
;

```

3. Global Variables and Interface with Outside World

For global variables, while we can store them on the stack, we have no way to locate them during runtime without using some form of base pointer. So, we decided to use temporary environment and unnamed LAM variables for global variables. The compilation is quite straight forward. Here is an example for globals:

```

program test(input,output);
var a,b,p: real;
begin
  p:=a+b
end.

NAMELESS proc_3
::
  ZERO THREE NDUPN DROP      (0)
  ' NULLLAM THREE NDUPN DOBIND (3) three dummy values
  3GETLAM                    (0) environment for 3 variables
  2GETLAM                    (1) a is at fixed location now
  %+                         (2) b
  1PUTLAM                    (1) a+b
  ABND                       (0) p is the first variable
  (0) discard environment
;

```

Program like shown above can only be useful if it can read value from somewhere outside and return result to somewhere. Generally speaking, we need the power to bypass the Pascal compiler, and to be able to insert system RPL words into the code generated. Inline procedures as defined Turbo Pascal would be a solution.

If we define two inline procedures 'tos' to fetch the top of the non local stack element and 'push' to push something on the non local stack, we can rewrite the above program as:

```

program test(input,output);
var a,b: real;

function tos: real; inline ' ( tos ) ';
procedure push(val: real); inline ' ( push ) ';

begin
  a:=tos;
  b:=tos;

```

```

    push(a+b)
end.

NAMELESS proc_3
::
    ZEROZERO                (0)
    ' NULLLAM TWO NDUPN DOBIND (2) two dummy values
    ( tos )                 (0) environment for 2 variables
    2PUTLAM                  (1) comment only
    ( tos )                 (0) store tos into a
    1PUTLAM                  (1) comment only
    2GETLAM                  (0) store tos into b
    1GETLAM                  (1)
    %+                       (2)
    ( push )                 (1) a+b
    ABND                     (0) leaving result on stack
;                             (0) discard environment

```

By non local stack, we mean everything on the stack before the first parameter of the current procedure/function is pushed. Beware that the tos and push defined here only works for empty local stack. They should be defined by the compiler so that they can work even when the local stack is not empty. For example, the expression 'a+tos' should be compiled into:

```

2GETLAM (1) local stack contains one element
SWAP    (2) get one element from nonlocal stack, remove old copy
%+

```

With tos, we can even accept variable number of parameters. Here is an example which does list summation:

```

function decompose(l: list): integer; inline 'INNERCOMP';

function sumlist(listlen: integer): integer;
var sum: integer;
begin
    sum:=0;
    while listlen<>0 do
        begin sum:=sum+tos;
              listlen:=listlen-1
        end;
    sumlist:=sum
end;

begin
    sumlist(decompose(tos));
end.

```

Inline procedures/functions can be very useful in extending Pascal's limited standard procedures/functions and access HP calculator's rich reservoir of math functions and user interface functions, while enjoying the benefits of Pascal compiler at the same time. Here are some examples:

```

function max(a,b: integer): integer; inline '#MAX';
function min(a,b: integer): integer; inline '#MIN';

type string=integer;
(* Define string as occupying one element space *)
(* so that it can be a function return type *)
function concat(s1,s2: string): string; inline '&S';
(* Char is implemented as binary integer internally, so *)
(* it needs to be converted to CHR before calling >T$ *)
function conchr(s1: string; c: char): string; inline '#>CHR >T$';

(* graphics *)
procedure lineon(x1,y1,x2,y2: integer); inline 'LINEON3';

```

```
(* complex, list, matrix, etc add your own *)
```

4. Record and Array

Although user defined record type in Pascal looks like a major feature, its implementation is quite straight forward. The compiler has to remember the offset address of every field relative to the start of the record. When generating code for simple variable access, adding field offset to the variable address will get the field address. Here is an example with both local and global variables:

```
type complex = record rel: real; img: real end;
var a: complex;

procedure negate;
var b: complex;
begin
  b.rel:=-a.rel; b.img:=-a.img
end;

NAMELESS proc_3
::
  ZEROZERO      (0)
  1GETLAM        (2) ( b.rel b.img )
  %CHS           (3) ( b.rel b.img a.rel )
  XYZ>ZY         (3) ( b.rel b.img -a.rel )
  2GETLAM        (2) ( -a.rel b.img )
  %CHS           (3) ( -a.rel b.img a.img )
  XY>Y           (3) ( -a.rel b.img -a.img )
  2DROP          (2) ( -a.rel -b.img )
;                (0) ( )
```

Up to now, the compiler still manages to find the exact address (offset) of a variable, local or global. Array as defined in Pascal would be totally different. Array in Pascal is a composite data type, with each element being a valid data object. While system RPL and user RPL defines array as a simple data type (atomic). Which means that each element is an object body. Object prologue should be added to it to form a valid object. When we compile Pascal into system RPL here, array is implemented as a composite data type. Separate mechanism should be provided to handle one dimensional and two dimensional arrays in RPL, and to explore HP calculator's rich set of array functions.

The basic problem of compiling array access is to generate code to compute the address of 'a[i]', where a is defined as:

```
a: array [index_min..index_max] of typet;
```

Any book on compiler will give a formula like:

```
address(a[i]) = address(a) + (i-index_min)*sizeof(typet);
               = (address(a)-index_min*sizeof(typet)) + i*sizeof(typet);
```

For local variable access, another version of the formula is used, because we are using offset that is relative to the current top of stack:

```
offset(a[i]) = offset(a) - (i-index_min)*sizeof(typet);
              = (offset(a)+index_min*sizeof(typet)) - i*sizeof(typet);
```

It follows that, we can't use DUP, OVER, 3PICK, 4PICK ... to fetch an array element, we can't use XY>Y, XYZ>ZY to store, because address is only available at runtime. We should instead use PICK and STOI which both accept an offset on the stack.

STOI means indexed store and can be defined as:

```
( Indexed store:      obn obn-1 ... obl index val )
(                      ==> val obn-1 ... obl          )
NAMELESS STOI
::
    SWAPDUP #2+ ROLLDROP UNROLL
;
```

An example will help to illustrate:

```
procedure p;
  var a: array [5..7] of record rel: integer; img: integer end;
      i: integer;
begin
  i:=6;
  a[i].rel:=1;
  a[i+1].rel:=a[i].rel+1
end;

NAMELESS proc_3
::
    ZERO SEVEN NDUPN DROP      ( 0)
    SIX                         ( 7) 6 for array, one for integer
    XY>Y                        ( 8) 6
    SEVENTEEN                   ( 7) store into i
    OVER                        ( 8) 7+5*2, 7 is offset(a)
    #2* #-                      ( 9) read i
    ONE                         ( 8) 7+(5-i)*2
    STOI                        ( 9) 1
    SEVENTEEN                   ( 7) a[i].rel:=1
    OVER                        ( 8) 7+5*2, 7 is offset(a)
    ONE                         ( 9) i
    #+                          (10) 1
    #2* #-                      ( 9) i+1
    EIGHTEEN                    ( 8) 7+(5-i-1)*2, offset for a[i+1]
    3PICK                       ( 9) 8+5*2, 8 is offset(a) now
    #2* #-                      (10) i
    PICK                        ( 9) 8+(5-i)*2
    ONE                         ( 9) a[i]
    #+                          (10) 1
    STOI                        ( 9) a[i]+1
    7DROP                      ( 7) a[i+1]:=a[i]+1
                                ( 0) clean up
;
```

An experienced system RPL programmer would say, this is only the code generated by a compiler, not me.

5. Control structures

Control structures in system RPL is actually very rich, we have IT, ITE, DO_LOOP, BEGIN_WHILE_REPEAT, BEGIN_AGAIN, BEGIN_UNTIL, ', SEMI, COLA, SKIP, case and even GOTO. The excessive usage of SKIP (explicitly or implicitly) make system RPL control structures not as efficient as GOTO based implementation. For example, skipping one single object or object pointers in run stream takes about 120 clock cycles. Skipping a block of 33 objects would take 1ms on GX machines.

We base our implementation here on relative jumps. Two new routines are defined to support IF_THEN_ELSE, WHILE_DO, REPEAT_UNTIL and GOTO in Pascal: JMP and FJMP.

JMP expects a binary integer offset on the stack, adding this offset to the next RPL instruction pointer will get the target instruction pointer. That integer should be a direct embedded constant. The original system RPL word GOTO must be followed by an object pointer. Which is faster, shorter but more restrictive.

FJMP expects a True/False flag and a binary integer offset on the stack. If the flag is False, control is passed to that target address, otherwise, the next instruction in run stream is executed.

GOTO statement is just JMP: goto 00; ... 00: ... can be compiled as:

```

        DOBINT
ASSEMBLE
    REL(5) lab_1
RPL
    JMP
    ...
LOCALLABEL lab_1
    ...

```

To simplify the example codes, we assume we could define a macro for system RPL compiler:

```

DEFINE OFFSET(x)      DOBINT      \
                      ASSEMBLE    \
                      REL(5) x    \
                      RPL

```

IF statement: if ... then ... else ... can be compiled as:

```

        ..            conditional evaluation
        OFFSET(lab_1)
FJMP
        ...            then part
        OFFSET(lab_2)
        JMP
LOCALLABEL lab_1
        ...            else part
LOCALLABEL lab_2

```

WHILE statement: while ... do ... can be compiled as:

```

LOCALLABEL lab_1
    ...            conditional part
    OFFSET(lab_2)
FJMP
    ...            loop body
    OFFSET(lab_1)
    JMP
LOCALLABEL lab_2

```

REPEAT statement: repeat ... until ... can be compiled as:

```

LOCALLABEL lab_1
    ...            loop body
    ...            conditional part
    OFFSET(lab_1)
FJMP

```

Implementation of FOR statement needs a special location for storing the end of loop variable index. Our P4 to system RPL compiler is designed to be a single pass compiler, so we can't allocate space for the end of loop index together with parameters and other variables. We choose to put them dynamically on the data stack. FOR statement: for i:=start to end do can be compiled to:

```

...      (1)  evaluate 'start'
1PUTLAM  (0)  i:=start
...      (1)  evaluate 'end', leave it on stack

```

```

LOCALLABEL lab_1
  1GETLAM    (2)  read current i
  OVER      (3)  end of loop value
  #> NOT    (2)  i<=end
  OFFSET(lab_2)  jump to end if index passed end_of_loop
  FJMP      (1)
  ...
  1GETLAM    (2)  read i
  #1+       (2)  increment by 1
  1PUTLAM    (1)  store i back
  OFFSET(lab_1)  jump to loop test
  JMP
LOCALLABEL lab_2
  DROP      (0)  remove end_of_loop index from data stack

```

System RPL programmer would notice that this implementation is less efficient than the system RPL built-in DO_LOOP which is supported by a runtime Do_Loop_Environment. DO_LOOP is not used to implement Pascal's FOR statement because DO_LOOP is executed at least once and access of the loop index within and outside the loop body is not well defined (especially if there are more than two levels of FOR loop). Another reason is DO_LOOP does not allow simple break_out_of_loop mechanism (especially for nested loops).

Breaking out of WHILE and REPEAT loop is simply a normal GOTO statement. Breaking out of a FOR loop using GOTO statement is possible too except that compiler would not automatically drop the end_of_loop index(es) for you. You will have to place the exact number of 'drop's before GOTO, where 'drop' is defined as:

```
procedure drop; inline 'DROP';
```

For example, the following code fragment will be compiled correctly:

```

label 00;
...
for i:=1 to 10 do
for j:=1 to 10 do
for k:=1 to 10 do
  if solution_found(i,j,k) then
    begin drop; drop; drop; goto 00 end;
00:
...

```

We will finish this section by implementing the CASE statement. The concept of CASE in Pascal or switch in C is implemented by a series of conditional statements in user RPL. System RPL provides several methods to perform similar tasks: OVER#=case, EQLookup, CK<n>&Dispatch, etc.

To implement CASE statement, we define a new basic control word here: XJP (case jump). XJP accepts a binary integer on the data stack and a binary integer array of relative offsets that follows the XJP word. XJP just finds the relative jump offset for that index, and jumps to that location. Here is an example:

```

procedure testcase(c: char);
begin
  case c of
    '0': ...
    '1': ...
    '2': ...
    '3': ...
  end;
  ...
end;

NAMELESS proc_3

```



```

::
    DUP                fetch c
    OFFSET(lab_1)      jump forward for XJP
    JMP                multiple pass compiler can do a better job
LOCALLABEL lab_3      '0' goes here
    ...
    OFFSET(lab_2)      jump pass case statement
    JMP
LOCALLABEL lab_4      '1' goes here
    ...
    OFFSET(lab_2)      jump pass case statement
    JMP
LOCALLABEL lab_5      '2' goes here
    ...
    OFFSET(lab_2)      jump pass case statement
    JMP
LOCALLABEL lab_6      '3' goes here
    ...
    OFFSET(lab_2)      jump pass case statement
    JMP
LOCALLABEL lab_1      actual dispatch without check
    FORTYEIGHT         subtract internal value of '0'
    #-
    XJP                case jump
ASSEMBLE
    CON(5) =DOARRAY    an array object here
    REL(5) lab_2        indicating size of array
    CON(5) =DOBINT     element type
    CON(5) 1           one dimensional array
    CON(5) 4           four elements
    REL(5) lab_3        offset for '0'
    REL(5) lab_4        offset for '1'
    REL(5) lab_5        offset for '2'
    REL(5) lab_6        offset for '3'
RPL
LOCALLABEL lab_2      end of case code
    ...
    DROP
;

```

6. Limitations and possible future developments

The preceding sections describe how Pascal features can be compiled into system RPL code, we have touched:

- 1) local and global variable declaration and access.
- 2) type declaration and related variable handling: integer, real, character, Boolean, array and record.
- 3) label declaration
- 4) procedure and function declaration, including inline procedure/function. Recursive procedure/function is supported naturally.
- 5) goto, assignment, procedure call and compound statements
- 6) if, while, repeat, case statements

Some features are already supported, but not explicitly mentioned:

- 7) constant declaration
- 8) enumeration type implemented as binary integer and set type implemented as hexadecimal string
- 9) most standard functions and procedures

Still, there are lots of standard or commonly accepted extended Pascal features not supported or not implemented quite well in this implementation, which provides lots of opportunities for future developments:

- 1) Integer type is temporarily implemented as system binary integer in system RPL which is actually unsigned integer type. Runtime routines for true signed integer arithmetics should be added, especially comparisons, conversions to and from real values.
- 2) variable parameter for procedures/functions is not supported yet. Use of relative stack level as a form of address and different storage scheme for local and global variables make it quite hard. The need to support system RPL's variable number of parameters and return values makes it worse.
- 3) nested procedure/function, with statement
- 4) pointer and string type. New and dispose routines
- 5) write, writeln, read, readln, text and typed file operations
- 6) procedure/function as parameters
- 7) break and return statements
- 8) conditional compilation, modular structure of program
- 9) constant expression
- 10) peephole optimizations

Another direction of possible developments is to expand Pascal's limited data types and standard procedures/functions using system RPL's rich resources and good concepts:

- 1) high precision integer (user integer type)
- 2) complex, double precision real
- 3) vector, matrix
- 4) list, symbolic, program, units
- 5) character string
- 6) graphic object (grob or bitmap)
- 7) text display routines
- 8) graphic display routines
- 9) keyboard input
- 10) menu, input form, choose box, parameterized outerloop
- 11) communication, timer, alarm system

Still another direction of possible development is to make the experiment reported here truly accessible and useful to end users. Things to be done includes:

- 1) Implement a Pascal language runtime ROM.
- 2) Generate code for external ROM.
- 3) Simplify steps to build a final ROM image.
- 4) Develop a scheme such that the user rely more on PC to write and debug program, and only turn into system RPL when everything seems OK.
- 5) more complete documentation and example programs for this Pascal compiler.

7. Acknowledgment

The author would like to acknowledge the HP calculator software team members for introducing me to the wonderful land of HP calculator. Jim Donnelly gave the first lesson on user RPL programming. Charlie Patton's lectures and lots of emails are very valuable to the understand of RPL internals.

8. References

- [1] W.C. Wickes, "An Evolutionary RPN Calculators for Technical professionals", HP Journal, Vol. 38, No. 8, August 1987, pp.11-17
- [2] D.K. Byrne et al, "An Advanced Scientific Graphing Calculators", HP Journal, Vol. 45, No. 4, August 1994, pp.6-22
- [3] S. Pemberton & M. Daniels, "Pascal Implementation", Ellis Horwood, Chichester, UK.
- [4] B. Kinnersley, "The Language List", Ver. 2.3, Sep 1994, Internet newsgroup comp.lang.misc
- [5] James Donnelly, "The HP48 Handbook", Armstrong Publishing Company, 2nd Edition, 1993
- [6] W.C. Wickes, "HP 48 Insights", Part I and II, Larken Publications, 1992
- [7] RPLMAN.DOC, RPLCOMP.DOC, ftp from hpcvbbs.cv.hp.com

9. Appendix

```
Basic Pascal runtime support library

( data stack: # -> )
( run stream: DOARRAY len*5+5 DOBINT off0 off1 off2 .... offn-1 )
NAMELESS XJP
CODE
    GOSBVL  =POP#           A=offset
    C=A      A
    A=A+A    A
    A=A+A    A
    A=C+A    A           A*=5

    D0=D0+   10
    D0=D0+   15           D0: point to first offset
    CD0EX
    C=C+A    A           C: point to THE offset
    CD0EX

    A=DAT0   A           A: value of offset

    CD0EX
    C=C+A    A           C=D0+offset
    CD0EX
    LOOP
ENDCODE

NAMELESS JMP  ( #offset -> )
CODE
    GOSBVL  =POP#           A=offset
    CD0EX           C=Instruction pointer
    C=C+A    A           C+=offset
    CD0EX           D0+=offset
    D0=D0-   10           adjustment ( 16 for external ROM )
    LOOP
ENDCODE

NAMELESS FJMP ( T/F #offset -> )
CODE
    GOSBVL  =POP#           A=offset
    R0=A.F    A           R0=offset
    GOSBVL  =popflag
    A=R0.F    A
    GOC      over
    CD0EX           C=Instruction pointer
    C=C+A    A           C+=offset
    CD0EX           D0+=offset
    D0=D0-   10           adjustment ( 16 for external ROM )
over LOOP
ENDCODE

* Indexed store:      stack[n] stack[n-1] .... stack[1] index value
```

```
*          ==> value      stack[n-1] .... stack[1]
NAMELESS STOI
::
  SWAPDUP #2+ ROLLDROP UNROLL
;
```

•