-- RPilot: Rob's PILOT Version 1.4.1 --

-- WWW: http://rpilot.sourceforge.net/ --

Intro::
RPilot is an interpreter for the PILOT programming language. PILOT, the Programmed Inquiry, Learning, Or Teaching language, was originally designed to be used by teachers as an aid in instruction. PILOT is easy to learn, and most people should be able to in a very short amount of time. PILOT was first developed in 1962, and was standardized by the IEEE in 1991. RPilot probably isn't completely standard, but it is very close. It also adds a few extensions not found in standard PILOT interpretors, such as rudimentry debugging and the ability to call other programs. RPilot was written in 100% ANSI standard C, and should be easily portable to any platform with a standard C compiler.

Programming::
PILOT programs consist of a series of statements, which are either labels or function calls. PILOT statements all end with a newline, and the lines themselves may be no longer than 127 characters long. PILOT is case insensitive when it comes to label and variable names, $name, $NaMe, and $NAME all refer to the same variable. Speaking of variables, PILOT has only two different types: string, and numeric.

Variables::
String variables consist of a list of characters which can be up to 127 bytes long. Numeric variables are equivilent to C's int. They can be any number in the range of -2,147,483,648 to 2,147,483,648 (with 32 bit rpilot) or -32,768 to 32,768 (with 16 bit DOS rpilot). Variables are referenced by their names, which can be up to 10 characters in length, and consist of a type designator followed by any non-whitespace characters. When I say "type designator", I mean that all string variables have a "$" as their first character, and all numerics use "#". Thus, $count refers to a string, and #count to a number. This leads to another point: a string variable and a numeric one can have the same name, so both $count and #count could exist at the same time. They are independent of each other, however.

PILOT also implements labels, which are used just like in any other language, and are declared in the following way:

*labelname

This would define a label called "labelname". All label definitions start with an asterisk (the "*"), and follow that with the label name. Label names, like variable names, are case insensitive.

PILOT statements that do not declare labels have the following syntax:

<command>[conditional] : [arguments]

<command> is any of RPilot's internal commands, which I will cover shortly. [conditional] is an optional conditional expression which determines whether the statement will be executed. This is different from more contemporary languages which use "if" clauses and the like, in that the "if" clause is implemented in every function, in a way. [arguments] is an optional list of arguments to be passed to the function. Some functions require arguments; others don't.

```
Commands::
    Commands in RPilot are exactly one character long.  This may seem
    a little wierd, but that's the way things are.  Even if you are a
    PILOT expert, you will find it beneficial to read ALL of the command
    descriptions, to learn the nuances of RPilot.  On to the commands:

    R ::
        The "R" (Remark) command flags the rest of the line as a comment,
        and therefore the interpreter ignores it.  This is a good way to
        add reminders to yourself about what you have typed. Example:

        R: This is a comment

    A ::
        The "A" (Accept) command takes input from the user.  It can
        take the name of a string or numeric variable as an argument, in
        which case it will store the input into the given variable.  If
        no arguments are given, it stores input into the string variable
        "$answer".  Examples:

        R: The next line gets a string from the user and puts it in
         : "$answer"
        A:

        R: This gets a number and stores it in "#group"
        A: #group

        Note that you can only name one variable for each A command

    T ::
        The "T" (Type) command is used to display information.  If any
        arguments are given, it displays the string, substituting
        variables as needed.  With no arguments, it prints a blank line.
        Examples:

        R: The next line prints the contents of the variables "$name",
         : "$rank", and "#serialnum"
        T: Name: $name Rank: $rank Serial Number: #serialnum

        T: Note that instead of using the same command over and over
        T: again, as I have just done, you can skip the command name
        T: and just use the colon, as I have done in a few examples,
         : like this.
         : It saves time, and makes things look nicer.

    J ::
        The "J" (Jump) command is like GOTO in BASIC and other languages.
        It causes the interpreter to jump to the label given to it as
        an argument.  When giving a label name, do not add on the
        initial "*".  Example:

        R: The next line causes a jump to the label "done"
        J: done

    U ::
        The "U" (Use) command is used to implement a sort of primitive
        subroutine.  It causes the interpreter to jump to the given
        label, just like J, but first it pushes to current file offset
        onto a stack.  When the E command is reached, that value is
        popped off and jumped to.  This is like GOSUB in BASIC.
        Example:

        U: sub
        T: Back from the sub!
        J: done

        *sub
        T: Now we're in the sub.
        R: We'll look at the "E" command next
        E:

        *done
```

E ::
    The "E" (End) command is used with U to return from subroutines.
    If no subroutines were called, E ends the program. Example:

    T: The next line ends this program
    E:
    T: This is never executed

    Also note the example for the U command.

M ::
    The "M" (Match) command is used to handle user input. It
    compares a given list of strings to the last string which was
    used to hold input from the A command. This also is very wierd,
    and it would probably help to check some of the example programs.
    If RPilot finds a match, it sets the variable "#matched" to 1,
    and the variable "#which" to the position of the matched string
    in the argument list. Example:

    T: What is your favorite flavor of ice cream?
    A: $icecream

    R: The next line checks to see if "vanilla" or "mint" were
     : entered during the last A command
    M: vanilla mint

    If $icecream = "vanilla" then #which would equal 1. If the answer
    was "mint", then #which would equal 2. In either case, #matched
    would be set to 1. If neither matched what the user typed, then
    #matched would be set to 0, and so would #which

C ::
    The "C" (Compute) command sets variables. The argument string
    contains a variable name followed by an equal sign ("="),
    followed by a value to assign tot he variable. For numerics,
    this is a mathematical expression which can contain one or more
    terms, and variables as well as constants. See the section on
    math for more info on expressions. For strings, it is a list
    of strings and variables which will be copied into the given
    variable. For example:

    R: This causes "#number" to be incremented by 5
    C: #number = 5 + #number

    R: This copies the contents of "$firstname" and "$lastname"
     : to the variable "$fullname"
    C: $fullname = $firstname $lastname

Y ::
    The "Y" (Yes) command works like T, except that it only prints
    if the variable "#matched" equals 1. This is usually used in
    conjunction with M. Example:

    A:
    M: Herbert Floyd
    Y: You typed "Herbert" or "Floyd"
    N: You entered some other name.

N ::
    The "N" (No) command is just the opposite of Y, it only prints
    if "#matched" equals 0. For an example, see above.

That's it for the standard PILOT. Next we take a look at RPilot's
extensions.

X ::
    The "X" (eXecute) command runs a line of PILOT code which you
    pass to it. The argument(s) are either a constant string,
    (ie, "T: Hello!"), or a string variable. Try the following:

    A:
    X: $answer

    It will wait for you to input a string, and then will try to run
    it. If you were to type "T: Boo!", that command would take
    place, and "Boo!" would be displayed

S ::
    The "S" (Shell) command allows you to execute other programs.
    This gives you access to the operating system and all other
    programs on the user's machine. For example:

    S: dir
    OR
    S: /bin/ls

    will display a list of files, depending on what operating system
    you are running.

      The return code of the program which you ran is stored in the
      variable `#retcode'. If a program runs succesfully, this should
      be zero. If not, it usually means that something went wrong.

D ::
    The "D" (Debug) command is used as a way to quickly and easily
    get a list of all variables and labels. It takes a string as an
    argument, and checks for two characters. If it sees an "l" or
    an "L" (case is unimportant), it will dump a list of all labels
    and their offsets. If it sees a "v" or a "V", it will do a
    variable dump, listing all variables and their values. This is
    useful when something goes wrong, and you need to check
    everything at once. Example:

    D: Lv

    Lists all labels, followed by all variables

G ::
    The "G" (Generate) command is used to generate random numbers,
    and place them in a variable. It takes three arguments, first
    a numeric variable, then two numbers. It randomly creates a
    number between the second and third arguments, and places it
    in the variable specified by the first. Example:

    G: #rand 23 56

    This generates a number between 23 and 56, and stores it in #rand

    This is especially useful when making things that require random
    values, such as games. (See the example programs for a few games
    that use G)

```
Math::
    RPilot supports the following mathematical operators (math ops):

    Standard:
    + :: Adds two things together
    - :: Subtracts one number from another
    * :: Multiplies two numbers
    / :: Divides one number by another

    RPilot Extensions:
    % :: Gets the modulo (remainder after division) of two numbers
    & :: Gets the bitwise AND of two numbers
    | :: Bitwise OR of two numbers
    ^ :: Bitwise XOR of two numbers

    I'm not sure how useful all the bitwise operators are, but they
    were easy enough to add, so why not?  RPilot works all expressions
    from left to right, and currently ignores operator precedence.


Conditionals::
    RPilot statements can contain "conditional expressions" (condexs),
    which are evaluated and checked to see if they are true.  If so,
    the rest of the statement is run.  They are placed after the command,
    such as in this example:

    J(#answer > 45): menu1

    In this case, the condex is "(#answer > 45)"  The J (Jump) will only
    take place if the value of "#answer" is more than 45.  If not, RPilot
    goes on to the next line.  RPilot understands the following
    "relational operators" (relat ops):

    =  :: True if two numbers are equal
    <  :: True if the first number is less than the second
    >  :: True if the first is greater than the second
    <> :: True if the two numbers are not equal
    <= :: True if the first is less than or equal to the second
    >= :: True if the first is greater than or equal to the second

    Of course, you are free to use all the math ops in a condex, such as:

    T(#score + 10 >= 50): You made it by at least 10 points!

    Note, however, that you can have only one relational operator per
      condex.

    In addition to standard condexs, there are also two other methods
    of testing a condition, Y and N.  If the condex of a statement is
    a captial "Y", then RPilot checks to see if "#matched" is equal to
    1.  If so, the condition is true, and is then executed.  "N" is the
    opposite, and checks whether "#matched" equals 0.  If so, the
    statement is executed.  These are ususally used after an "M" command,
    for example:

    A: $answer
    M: 1776 1812 1968 1998
    R: The following line jumps to "correct" if "#matched" equals 1.
     : That would be true if "M" matched any dates listed
    JY: correct
    R: If "#matched" equals 0, then "N" statements are true, and are
     : then executed
    TN: You did not answer correctly
```

Special Variables::
    RPilot has two special variables which you can use. They are special
      because they are set by the interpreter itself, so you can use them
      without assigning a value first. The two special variables are:

      $prompt - This is the prompt which is printed out for an A: (Accept)
                command. The default value is ">". If you run the following
                  program:

                  T: Please enter a number
                  C: $prompt = rpilot>
                  A: #number

                  You would see that when you are asked for a number, the
                  prompt is "rpilot>" rather than ">".

      #rpilot - This is the version number of RPilot. For version 1.4,
                it's value is 14. RPilot 1.5 would be 15, and 2.0 would
                  be 20.

Interactive Mode::
    If RPilot is invoked with the "-i" command line switch, it will
      attempt to enter an interactive mode, where you type commands in, and
      have them executed immediately.

      Interactive mode is not a part of the `rpilot' program, but rather
      it should be written as an external PILOT program. Therefore, in
      order for RPilot to use interactive mode, it needs a program to run.
      It will find one in one of two ways:

      1) Look for a program called "interact.p" in the current directory.
      2) Run the program specified by the RPILOT_INTERACT environment
         variable.

      The `inter.p' file located in the examples directory is a good
      program to use. Feel free to customize it by editing the source code.

Note::
    I think that the best way to learn RPilot is to look through all the
    example programs after reading this, and hopefully it will make more
    sense. They are all fully commented, and are meant to serve as
    learning aids.