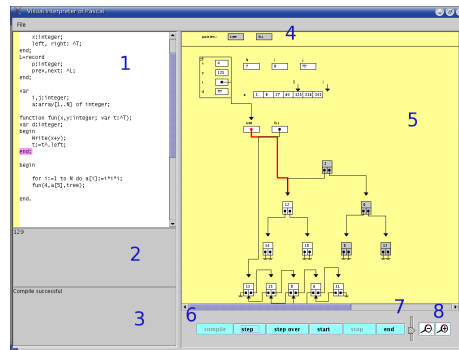


# Quick introduction to VIP

## Overview and quickstart

VIP (Visual Interpreter of Pascal) is an interpreter of the core subset of Pascal with some extensions, whose main feature is the graphical visualisation of the program execution (memory, stack, control flow). The user interface has the following elements:



- 1) — the code of the interpreted program. The current line is highlighted. Breakpoints can be set in the narrow strip on the left,
- 2),3) — the consoles where the program output and the other messages go,
- 4) — buttons that switch on/off the visibility of respective variables in the main window,
- 5) — the main window, where the contents of the program's memory is displayed,
- 6) — the control panel: start, stop, pause buttons, also a speed control slider and scaling buttons that increase/decrease the main panel contents.

VIP operates in edit, compile and run modes. The program code can only be changed in the edit mode. After the *compile* button is pressed and any compile errors are resolved the program execution starts and can be controlled with the control panel in an intuitive way. The *end* button takes you back to edit mode after the program has ended. Breakpoints can be set at any time of run mode.

## Language extensions

Here is a list of the properties of the programming language that VIP interprets:

- the following elements of Pascal work in VIP: integers, reals and booleans, arrays, records, user-defined record and array types, pointers, functions (no nested declarations however) with parameters both by value and variable, while/for/if constructions, Write/Writeln for output. Pascal syntax is used.
- extra available functions are: `change(a,b)`, `sqrt(real):real`, `max(...)`, `min(...)` with any number of parameters
- array bounds can be simple expressions (e.g. `array[1..N] of integer`), they will be computed at run-time, when the array is created
- input to the program can be specified as input parameters in the program header as in any procedure declaration, for example:

```
program binSearch(x,N: integer; a: array[1..N] of integer);
```

The user will be prompted to provide values after compilation. Some default initialization patterns are provided to choose from. The value from the previous program run can be reused if possible too.

Only variables of simple types, one- or two-dimensional arrays of such variables or pointers to special types (see below) can be used as input. User-defined types declared later in the program may be used in the header.

- special types are lists, double lists and binary trees. There is a naming convention that must be followed to allow such types to be recognized. It is based on the names of the last fields in the record description, as in the example:

```
type Tree    =    record
                    ... my fields ...;
                    left, right: ^Tree;
                end;
```

The last field for a list must be called `next`, for a double list they are `prev` and `next`, for a binary tree `left` and `right`.

You may use lists and trees as input:

```
program BSTInsert(x: integer; t: ^Tree);
```

provided the type `Tree` is defined later in the program.

- variables are uninitialized by default. Reading from such variable will cause a run-time error.
- errors are also triggered by a null pointer reference and array index out of bounds usage.

There is an attachment to this document that contains the syntax of VIP.

## Visual features

- each variable corresponds to a box in the main panel. The general rule is that there are no „nested boxes” on screen — for instance an array inside a record will not be shown. Simple variables, pointers, one- and two-dimensional arrays of these and records are displayed.
- pointers are arrows. Click on the pointer starting box to highlight the arrow and see clearly where it points. If a variable value does not fit in its box click on the box to see the value in the output console.
- special data types (trees, lists) have special layout patterns, more sophisticated than general record types. For instance a tree „looks like a tree”. They also have dedicated visual input procedures that help create the desired tree shape with a few mouse clicks.
- when an array reference to a static array occurs in the program, the cell referred to will be pointed by an arrow with the indexing expression. These indices remain visible from the first use onwards and they are updated after each instruction.
- when functions are called the stack (divided into successive frames) is displayed. Due to space limitations only simple variables and pointers are displayed on the stack. You may compress/expand any stack frame to trace the previous calls. To see the function call in the code that created some stack frame, click on that frame. A parameter passed by variable is indicated by putting a name at the actual variable that was passed.
- there is a „garbage collector”: the dynamic variables that are no longer referenced and have not been disposed are marked in gray
- the display may be rescaled and the execution speed of the program regulated