Miroslav Nemecek

Guide

to the World of Peter Rabbit

Gemtree Software

Guide to the World of Peter Rabbit

Reference book for users of the Peter application

December 2000, updated September 2012

petr.hostuju.cz
www.gemtree.com

This text was originally designated for users of licensed version of the Peter application, so some its parts may be invalid with current version (e.g. installation does not require installation CD and license diskette).

Contents:

1 Setup

Installing Peter on Your Computer

Our work with Peter begins by installing the program. You will need the setup CD and the license floppy disk, computer with a Pentium processor (or at least 486), floppy disk drive and CD-ROM drive, and with Windows 95, 98, NT or 2000. There should be about 500 MB of available hard disk space, but as few as 10 MB will be enough for minimal setup.

First, insert the CD into your CD-ROM drive. After a while, you will see the setup program window. If this window does not appear, the CD AutoPlay feature is probably disabled, and you will have to start the setup program manually: Click **Start** / **Run** / **Browse**, locate the **SETUP.EXE** program on your CD, and click **OK**.

The setup program window contains four choices. The first choice installs the Peter application, the second one adds or removes installed components, the third one uninstalls the Peter application, and the last choice quits the setup program without applying any changes.

Click the first choice, **Install**. The setup program prompts you to insert the license floppy disk. Insert your license floppy disk into the drive, and then click the **Next** button. Wait a little while the setup program reads all the necessary data from the disk. After that, the License Agreement window appears. Read this agreement carefully, and then accept its conditions by clicking the **I Agree** button.

When you accept the License Agreement, a window containing the installation choices will be displayed. Your license data are displayed in the upper right corner. Make sure that these data are correct and complete. The upper left pane contains check boxes that specify the components of the Peter application you want to install. To the right of the check boxes, the size of the components is displayed. Select the appropriate check boxes for the components you want to install.

The most important check box is the first one, which represents the Peter application main program. This check box should always be selected. The second check box installs the sample programs created in Peter. The sample programs are a good beginning for creating your own programs, and so it is recommended to keep it selected as well. The remaining check boxes install libraries containing images, sounds, sprites, and other elements. These libraries are not required for the operation of the Peter application itself. You can select them as you wish, depending for example on the amount of available hard disk space.

The amount of free space required on the target disk is displayed in the middle right part of the window. The bottom part shows the name of the target folder, into which Peter will be installed. You can change the folder by clicking the **Browse** button. When you are ready, click **Finish** to start the setup process.

Multi-user Environment

The Peter application may be used by several independent users. Multi-user environment is achieved by separating the Peter application folder from the individual users' data folders. Each user has their own working space, in which they can change both sample and their own programs and libraries, without any effect on the other users' programs and libraries.

On the Windows desktop, right-click Peter's icon, and then click **Properties**. The properties dialog shows two paths. The first path is labeled **Target**, and it represents the path to the main program of the Peter application. Typically, this will be **"C:\Program Files\Peter\Peter.exe"**. This path will be the same for all users. The second path is called **Start In**, and it represents the path to the user's working folder. This is usually **C:\My Documents\Peter** (according to Windows settings). Each user has his or her separate working folder.

If you want to create a Peter startup icon for a new user, you should first create the user's working folder by using e.g. **Windows Explorer**. Browse to the users' common working folder (typically **C:\My Documents**), and create a new folder by choosing **File** / **New** / **Folder**. Then, right-click the Peter application icon, drag it on the desktop, and click **Copy Here** to create its copy. Right-click the new icon and display its properties. Modify the **Start In** path to represent the path to the new user's working folder. You can also (using the right mouse button again) rename the new icon, and everything is ready for the new user.

Network Installation

When installing into a networking environment (e.g. in a school class), install Peter on the network **server** by the same procedure as with a standalone computer. The folder with the Peter application may be set as **Read-only** after setup, as no additional write operations will be performed in it.

When Peter is installed, create Peter's working folder for saving user files on the system administrator's workstation (e.g. **H:\My Documents\Peter**). Prepare a startup icon whose **Target** path will refer to the Peter application's main program (**Peter.exe**), and whose **Start In** path will refer to Peter's working folder. Create working folders for the remaining workstations in the network, and copy the startup icon on their desktops.

Peter, just like other 32-bit applications, supports long file names with extended characters. In Novell networks, you can enable long file names by using the following commands:

1. On the server, type "**load os2**",
2. On the server, type "**add name space os2 to** *volume1*", where *volume1* is the name of the volume where you install long file names support,
3. Add the "**load os2**" command into **STARTUP.NCF**.

In some versions of Novell networks, it is impossible to run programs with extended characters in their names (not only from Peter, but also from **Explorer**), although all the remaining file operations work fine. In such cases, you will have to avoid using extended characters in program names, or upgrade the network software.

Reinstalling

When there is not enough disk space, you may install only some of Peter's components. You can add or remove the individual components that will be installed by clicking **Add/Remove** in the setup program. After you insert the license disk and the license data are loaded, the same selection window appears as when you perform installation, but now you cannot change the target folder. Blue check boxes indicate the components already installed. You can change what components will be installed by selecting and unselecting the appropriate check boxes. Click **Finish** to begin reinstallation.

Changes to the Sample Library

You can add or remove files from the sample library as needed. When you uninstall a library, the setup program removes only the original, unchanged files. When you add files, they will overwrite files with the same name, regardless of the date, time and size of the files.

When you run Peter from the Windows **Start** menu, you can make direct changes to the sample library. In such cases, the sample folders become working folders at the same time. You may also use the **Peter with Sample Library Modification** command in the **Peter x.xx** menu.

Uninstalling

You can uninstall Peter by running the setup program from the setup CD-ROM and clicking **Uninstall**. Another possibility is launching the uninstaller through Windows **Add/Remove Programs** control panel or by clicking **Start** / **Peter x.xx** / **Uninstall**.

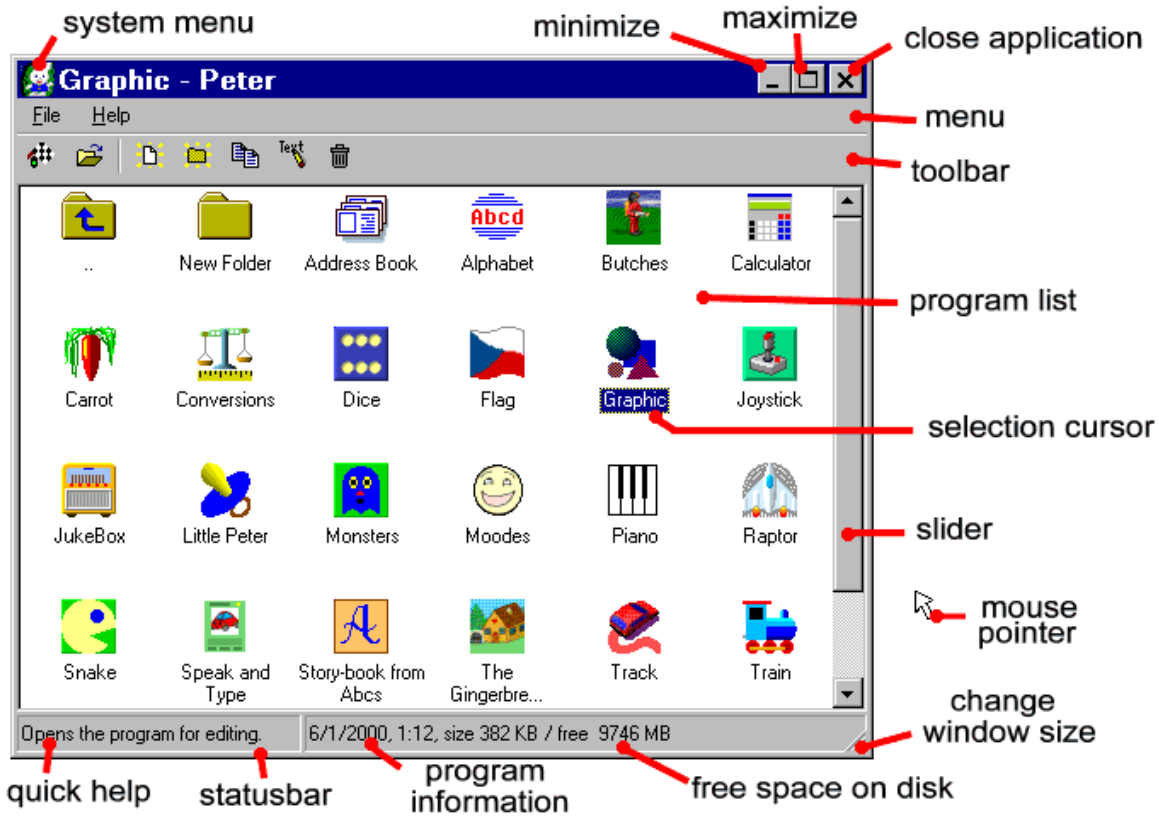
Uninstalling deletes all unchanged sample programs and libraries and Peter's program files. Uninstalling does not delete users' working folders. If needed, you can delete these folders manually, e.g. in **Windows Explorer**.

2 The Program Window

After Peter starts, you will notice the color icons displayed in the **program window**. These represent the sample programs created in Peter, and they are plentiful. The sample programs are a good starting point for creating your own programs. They can be both an inspiration and an answer to the question “How to do it?” It is very easy to take a prepared program and modify it to create a new one.

How do we begin with Peter? **By playing.** Take a good look at the sample programs, try their functions, and play with them. You will find out what to expect from Peter, and learn the rules and conventions of controlling programs. You will probably also come with many ideas about what could be done in a better way.

Don’t be afraid of playing, as any creative work is a play, in fact. Treat your work with enthusiasm and playfulness — this is the way for acquiring the best results.



Let us take a closer look at the program window. Try the way of starting programs. Double-click a program’s icon. Alternatively, click an icon once, and then press **Enter**.

There are several ways to close programs. You can close most programs by pressing **Esc**, some of them even by pressing any key. Programs are also closed when you click the little ☒ button in the upper right corner of the program window. If none of these ways suits you, press the **Alt+F4** combination on your keyboard (hold down **Alt** and press **F4**). This way you can also close full-screen programs without the ☒ button.

Let us take one more look at a running program's window. Most programs have two more buttons besides the closing button in the upper right corner. The middle button with one or two rectangles switches the window size to maximal ( ) or to adjustable size ( ). The left button enables you to collapse the program window into a small button on Windows taskbar. You can expand the program to its previous size by clicking that button again.

You may have noticed that when you move the mouse pointer over the border of the window, it changes into a double arrow ↔. This indicates that you can move the window's border. If you do so, you will see the window changing its size, and its contents change their size accordingly. This is not typical for standard Windows applications, but it is a common feature of graphical programs created in Peter. It enables you to stretch your favorite game over the whole screen, or to shrink it into a very small window.

Another interesting feature of the programs created in Peter is the possibility to pause the program by pressing **Pause**. You can then resume the program by pressing any key, for example **Pause** again.

The last, but not least interesting feature of Peter's programs can be invoked by pressing **Alt+Enter** (hold down **Alt** and then press **Enter**). This combination switches the program into full-screen mode or back into normal mode. In full-screen mode, the border of the window disappears, and your monitor switches into the mode that suits the program's requirements best. You must have a DirectX driver installed to switch into the full-screen mode. In Windows 98 and Windows 2000, the DirectX driver is included in the system.

Let us return into Peter's window. Like with Peter's programs, you can close the Peter application itself by clicking the button in the upper right corner or by pressing **Alt+F4**. You can also minimize the application's window by clicking , maximize it by clicking , or change its size by dragging the border with the ↔ pointer.

Take one more look at the **program window** picture on the previous page. In the upper left part of the window, there are several special icons. They represent folders. Folders contain programs or other folders. They organize programs into groups, so that it is not necessary to have all programs "stacked in one pile".

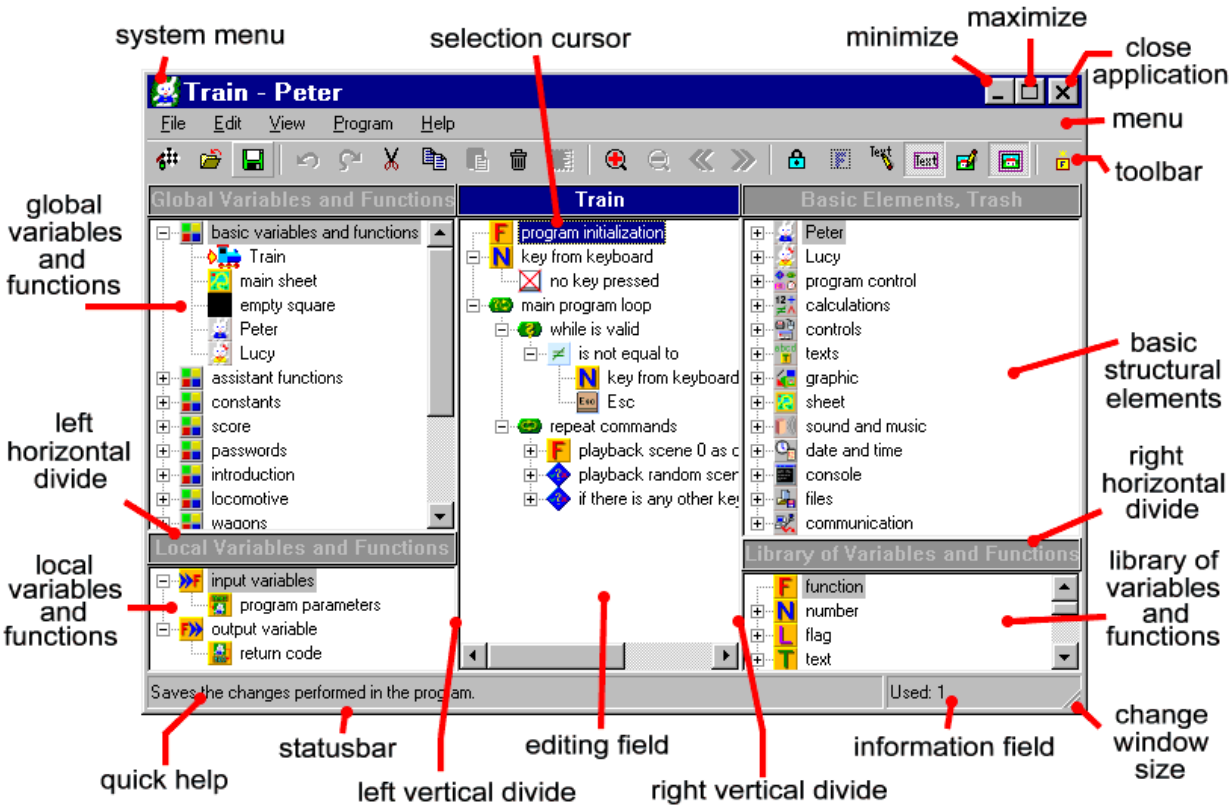
The icon labeled **New Folder** is important. You can get inside a folder by double-clicking its icon (similarly to starting programs). If you want to get out of the folder, double click the icon.

3 How Does It Work?

If you have played enough with the prepared programs, you surely wonder what it looks like inside Peter's programs.

We will look at one program. For example, we will take the **Train** game. It is located in the **Puzzles** folder. Click the program icon. It will become enclosed in a frame and highlighted. The frame is called **selection cursor**, highlighting indicates a **selected** program.

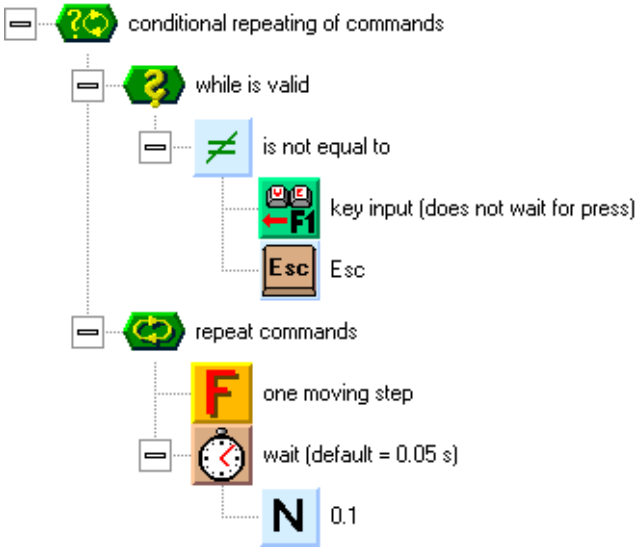
Notice the bar with color icons in the upper part of the window. This is a **toolbar**, and it contains buttons for invoking actions quickly. The second button from the left is called **Open** . This opens the program and allows its contents to be edited. Click this button.



After you click the button, the program expands into many color icons. You have entered into the program editor. This opens up the great world of Peter the rabbit. If the icons seem unfamiliar to you, there is no need to worry. You will soon be working with them with the ease of a professional.

As you can see, the editor is divided into five parts — five windows. The middle window is the one that interests us most. On the picture, you see that it is labeled as **editing field**. This window is most important. It is used for assembling programs. The word "assembling" is used on purpose, as the programs are really assembled from predefined components. Each component — an icon — represents one command, one

program element. All of the elements make up a working program, for example the **Train** game.



It is a distinguishing feature of Peter that the program elements, represented by the graphical icons, are put together using tree structures. The structures are like a tree with branches and leaves. Or do the colors and shapes of the icons look more like gems to you? (Why do we mention this? Guess what inspired the company name, **Gemtree**.)

One of the most useful features of the tree representation is the possibility to collapse its individual branches into icons. Only the part of the program that you are interested in can be left open. This feature of Peter's editor enables a radical improvement in the programs readability.

Before we start to rearrange a program, we have to learn how to control the editor. To the left of several icons, you may notice gray squares with a "**+**" or "**-**" sign:

These squares tell us that the program element contains a branch with other elements. If you click a square with the "**+**" sign, the branch expands. If you click a square with the "**-**" sign, the branch collapses.

Look at the upper left window. Its title reads **Global Variables and Functions**. Each data element used in a program, e.g. a numeric variable containing a number, or a graphical variable containing a picture, must be created first in the **Global Variables and Functions** window (or in the **Local Variables and Functions** window, but this will be discussed later). The act of creating a data element is called **declaration** in programming languages, but as we work in a purely graphical environment, we do not have to pay much attention to such terms.

Each data element contains something during the creation of a program. A numeric variable contains a number, a graphical variable contains a picture, an item variable contains an item (which is basically a picture whose size is 32x32 points). You can view and change the contents of the data elements by double-clicking their icons. The contents of the elements appear in the editing window, and the menu and the toolbar change accordingly to the type of the element.

When viewing the contents of the elements, you will come across two special types of them. The first one contains commands, and it is referred to as a **function**  . Function is not a data element, and so its contents cannot be changed or passed during the program's run-time. It is used in the program as a regular command. The second element, **group**  , does not contain anything, and is used only for grouping elements for the sake of the program's readability.

Notice the **Zoom In**  and **Zoom Out**  buttons on the toolbar. These change the size of the active (selected) window view. They are used most commonly in the graphical editor for increasing or decreasing the size of pictures. They are also used in windows with tree structures, for example the **Global Variables and Functions** window. Select the window by clicking it. The selected window is indicated by highlighting its title bar (usually dark blue color). Then you can use these button to switch the view of the icons in the window to half and normal size.

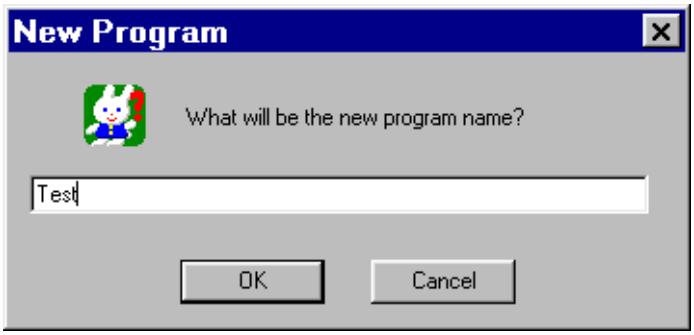
If you still want to play with the sample programs, you can modify the pictures in them. The modified program starts when you click the **Start**  button (the first button from the left on the toolbar). If you want to take the changes back, click the **Undo**  button (fourth from the left). You don't have to be afraid of damaging the sample programs. If needed, you can simply delete the program. Only the modified version will be deleted, and the original sample program appears in its place again. When you finish modifying the sample programs, close the program editor by clicking the **Close**  button (the second button from the left), and you may finally start creating.

A useful tip: If you want to learn something about an element, select it by clicking its icon, and press the **F1** key. This displays a comprehensive help for that element.

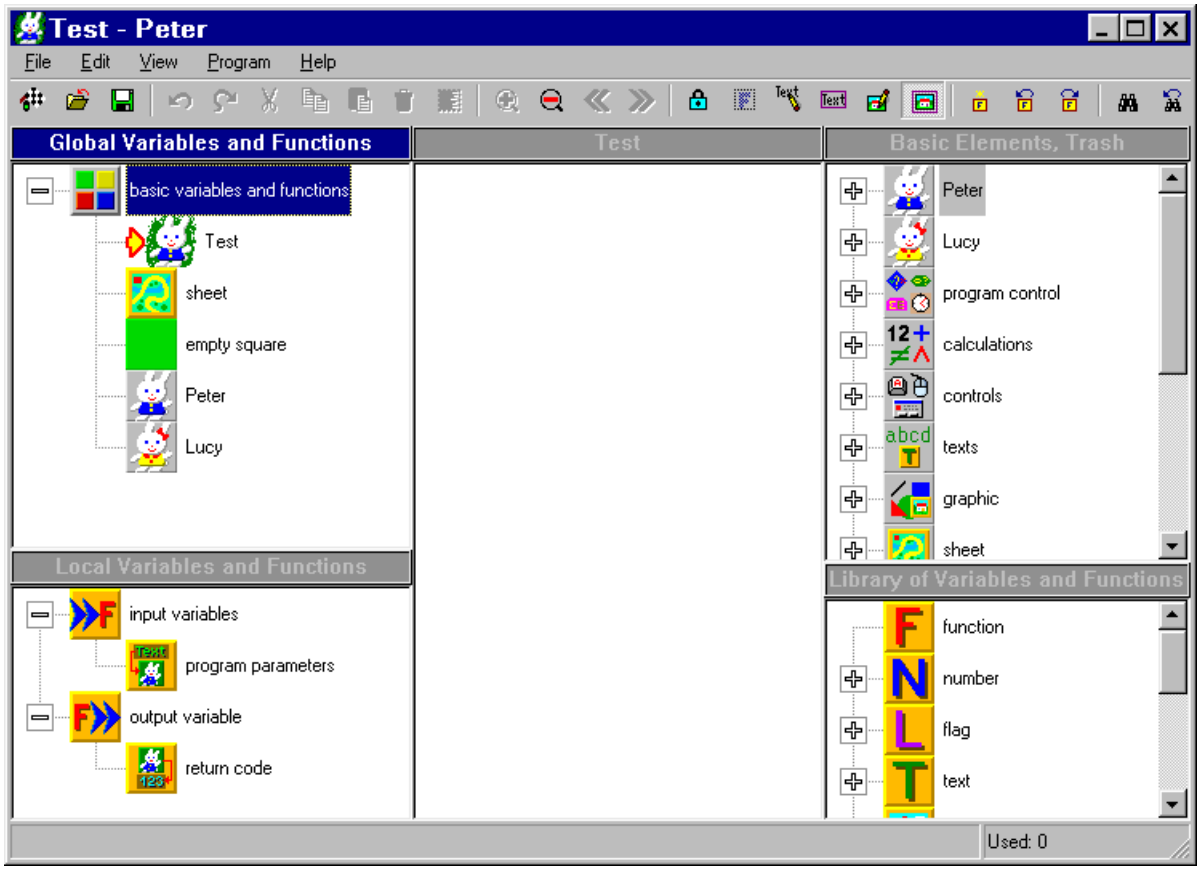
One more tip: Sometimes when you save or start a program, the editor may display an error message stating that the program is probably running. A running program is the most common cause of problems. If you don't quit a program and switch into Peter's editor, e.g. by clicking on Peter's window, the program hides under this window. It is not possible to save changes to a running program. You must close the program first. For this reason, you should always close the program first, and only then start modifying it.

4 First Steps

Let us start to create our own program. First, we have to create a new, empty program. In the **program window** (the window that appears after Peter starts), click the third button from the left  . If you rest the mouse pointer over the button before clicking it, a small window with the word **New** appears, and the quick help in the bottom part of the window says **Creates a new program**. After clicking the button, you will be prompted to enter the new program name. Type **Test** and press **Enter**.



Peter creates a new, empty program called **Test**, and opens it for editing. You will see the **program editor**, as we described it in the previous chapter, only the editing window will be empty.



In the **Global Variables and Functions** window (upper left), a group called **basic variables and functions** is prepared. This group contains several basic elements. You can change them, but cannot delete them. We will describe the elements one after another.

The first element  is called **main program function**. It contains the program as such, and is empty when new program is created. The text besides the icon is the **main function name**.

Notice the arrow to the left of the main function icon:  . The arrow identifies the element that is currently being edited, that is, the element displayed in the editing window. You can edit an element by double-clicking its icon.

The main program function has two interesting properties: (1.) Its icon is also the **main icon** of the program. You see this icon in Peter's program window, as well as in (e.g.) Windows Explorer. (2.) The main function name will be displayed in the title bar of the program. In new programs, the main function name is preset to the program name (**Test** in our example).

You can change the main function icon (or any other icon) by clicking the **Icon**  button. The name of the main function or any other element can be changed by clicking the **Description**  button. Before you edit an element, select it by clicking its icon. You can also modify the descriptive text by pressing **Alt+Enter**, or by clicking the text beside the selected element. A little frame with a blinking cursor appears. After you change the text, press **Enter**. The **Esc** key returns the text to its original state. When you are changing the text for more elements in a row, you can move the editing frame with the up and down arrow keys, and only press **Enter** on the last element.

The element under the main program function is called **sheet**  . If you double-click its icon, a green sheet separated by blue lines opens in the editing window. The grid of the blue lines serves only for your orientation on the sheet, and you can turn it off by clicking the **Raster**  button. The sheet separated into green squares remains displayed. The sheet in the **basic variables and functions** group represents (unlike other sheets) also the running program's window sheet, and it is called **main sheet**.

The contents of the squares are called **items**. Items are pictures with the size of 32x32 points. Changing the items on the main sheet squares also changes the contents displayed on the sheet. This way, you can create various animations.

Under the main sheet, there is an **empty square**  element. This item fills every new sheet. Double click the empty square to edit it. Draw something into the square, and switch to the main sheet editing. Notice that all the squares on the sheet have changed.

The last two elements are labeled **Peter**  and **Lucy**  . Double-click Peter's icon. Several pictures of Peter in different positions and directions appear. To be precise, there are four rows, each containing five pictures. Such an element is called a **sprite**. A sprite is a moving animated object, such as the rabbit, which we can move around the window sheet. Later, we will learn how to change Peter's and Lucy's appearance by changing the sprite.

Now, let us look at the upper right window. It is called **Basic Elements, Trash**, and it contains all of the Peter development environment commands and features necessary for creating programs. The trash is a supplementary feature of this window. Here you can move all the elements that you don't need.

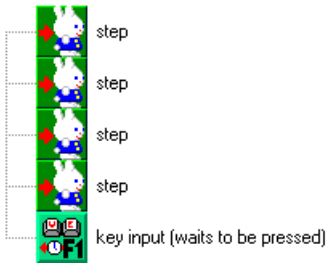
Now we can start "writing" our first program command. Start editing the main program function by double-clicking its icon (the icon labeled **Test**). In **Basic Elements**, expand the first icon from top with the Peter  picture. You will see commands used for controlling the Peter character.

Click the **step**  element, and drag it into the editing window. This means: Click the left mouse button on the element's icon, and while holding the button, move the mouse over the editing window. Notice that under the pointer, there is a transparent picture of the element you are dragging and its text. On the bottom edge of the pointer, there are two white overlaying rectangles. These indicate that the element will be copied, i.e. the original element will remain in its location, and a copy of the element will appear in the new location. Try to move the mouse behind the editing window. A black struck circle appears on the bottom edge of the pointer, indicating that you cannot drop the element here. Return to the editing window and release the mouse button. The element appears in the upper left corner of the window.

This way, we created the first command. Nevertheless, this will not be enough for us, and we will create four such commands. It is not necessary to drag them again from the **Basic Elements**; we can duplicate three times the command that is already in the window. We will drag the elements similarly to dragging from the **Basic Elements**, but this time, we will use the right mouse button, and we will drop the elements one under another. The right mouse button always copies the elements. The left mouse button moves the elements into a new place. Between the editor windows, you can only copy the elements, no matter which button you use.

When there are four **step**  elements, one under another, our first program is ready. Now we just have to run it. This is done using the **Start**  button. It is the first button from the left on the toolbar. When you click the button, a program window filled with a green sheet appears. Peter the rabbit is in the bottom left corner. He makes a few steps to the right, and the program closes. It is a great program, isn't it?

We only don't like one detail. It is the fact that the program closes immediately. For this reason, we will add a command that will wait for a key to be pressed. It can be found in the **controls** group, the **keyboard** subgroup, and it is labeled **key input (waits to be pressed)**. Add this element behind all of the commands. The whole program will look like this:

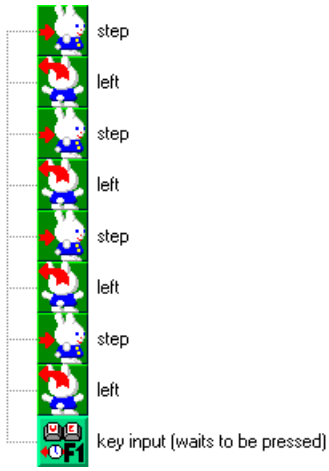


Run the program. The rabbit goes to the right again, and waits there. The program will close when you press any key.

Our first program is done. It cannot do much, but it is our own program. Moreover, what is most interesting — it is a fully **32-bit Windows multitasking application**. What does it mean? You can run the program as many times as you want to, and all of the programs will run at the same time. Advanced users can use Windows Explorer to look into Peter's programs working folder (usually **C:\My Documents\Peter\Program**). Here you can find the **Test.exe** program, which you can share with your friends, or create an icon for it on the Windows desktop. The program is **not** dependent on the Peter environment in any way, and you can use it just like any other Windows program.

After our first successful steps, in fact, Peter's steps ("*That's one small step for a rabbit, one giant leap for mankind*" — does that ring a bell?), we can enthusiastically go on experimenting.

Besides the step command, you have probably noticed other commands for controlling Peter, such as turning to the left, right, and back. Try to modify the program, so that Peter makes a small circle and returns to his original place. The result should look like this:



Does it work? Congratulations, you have just become a programmer.

5 Peter's Garden

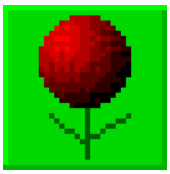
After the first steps, we will teach Peter how to plant his own garden. We will continue editing the **Test** program. We will add a picture of a flower, and tell Peter to plant the flower during his walk.

First, we will prepare the picture of the flower. We will use an empty square, into which we will draw the flower. Look at the **Global Variables and Functions** window. In the **basic variables and functions** group, there is an **empty square** icon. Using the right mouse button, drag the icon under all of the elements in the group.

This creates a copy of the element with the name of **empty square 2**. Click the icon to select the element (it will be marked by a rectangle). Click the text beside the icon. The text will be framed, and there will be a blinking cursor. Type **Flower** as the new element name, and then press **Enter**. This creates (declares) a new item called **Flower**.

Now we will draw the picture of the flower. Double-click the **Flower** icon. An enlarged picture of the empty square appears in the editing window. You can draw the flower using the following steps.

On the toolbar, there is a drop-down list of the graphical editor functions. Another drop-down list enables you to select line thickness. Choose the **Sphere** drawing function. At the bottom of the editing window, there is a color pick-list. Click the red color (the topmost one). At the top of the picture, approximately in the middle, click and hold the left mouse button, and drag the mouse towards the middle of the picture. Release the button. The red sphere will be the bloom. Pick a dark green color (the second from the bottom) with the left mouse button, and in the functions list, choose **Line**. Draw one line downwards from the sphere and then two lines on the sides. These will represent the leaves. The result could look like this (maybe a dahlia?):



When you finish drawing, return to editing the main program function by double-clicking its icon (it should still be called **Test**). If you want to get back to the elements that you edited before, you can use the **Previous Edit** « and **Next Edit** » buttons that enable you to scroll through the history of the edited elements.

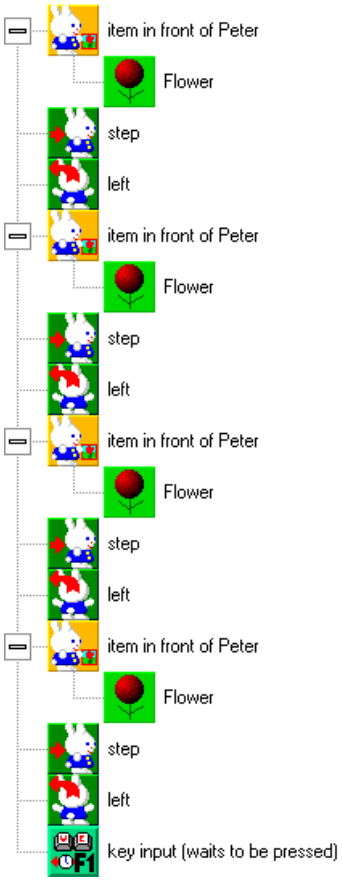
Let us assemble the new program. In the previous chapter, we have left it in a state, in which Peter took one step, turned to the left, repeated both of the commands for three more times, stopped, and waited for a key to be pressed.

Take one more look into the group with commands for Peter. You can find it in the upper right window called **Basic Elements, Trash,** and it is called **Peter**. The sixth element from the top is **item in front of Peter**. Drag this element in the uppermost place in the program, in front of the first command. When you drop this element, an

equals sign appears in front of it. By this, the editor informs you that the element requires a parameter. This parameter will be the flower we have created.

In the **Global Variables and Functions** window, drag the **Flower** item towards the **item in front of Peter** element, so that the upper left corner of the element you drag gets over the bottom right corner of the destination element. During dragging, you may notice that when you are close to the **item in front of Peter** element, a selection rectangle appears around its text. The destination element signals in this way that you can drop the element there. It is a basic feature of the elements that you can only create combinations that make sense. You cannot assemble nonsense (automatic syntax).

When you connect the flower item, the equals sign disappears. The **item in front of Peter** element has its parameter, and is satisfied now. We have created a small branch consisting of two elements, whose meaning is laying the flower in front of Peter. Copy the branch three times before the remaining step commands (by dragging the **item in front of Peter** element with the right mouse button). As a result, you should have this program:



Start the program. Peter plants four flowers and stops on his original position. One must admit that he steps on the flowers, but we should forgive him. He is just learning how to take care of his garden.

6 The Garden with Repetitions

Imagine Peter wanting to plant a garden that would be 15 squares wide and high. We could make more copies of the commands from the previous chapter, but this would make any programmer feel ashamed ☹.

For this reason, we will learn how to use a **cycle**. A cycle ensures repeated execution of commands. We will use a cycle with a set number of repetitions. It is labeled **command repeating with specified run number**, and you can find it in the **Basic Elements, Trash** window, in the **program control** group. Drag it to the very beginning of the program from the previous chapter (in front of all of the commands).

When you drop the cycle element, notice that two more elements came with it. These two elements are a natural part of the cycle, and cannot be deleted, nor moved somewhere else. The first element is labeled **for number of repetitions**, and it specifies the number of times that the cycle commands will be executed. The second element is called **repeat commands**. Here you put the cycle commands that you want to repeat. It is called a **cycle body**.

In the beginning, we specify the commands that will be executed in the cycle. From the previous chapter, we have a step command, and a command that lays a flower in front of Peter. Drag these two commands into the cycle body, that is, into the **repeat commands** element. Just to remind you — the command for laying the flower in front of Peter had been created from two elements, the **item in front of Peter** element and the connected element called **Flower**.

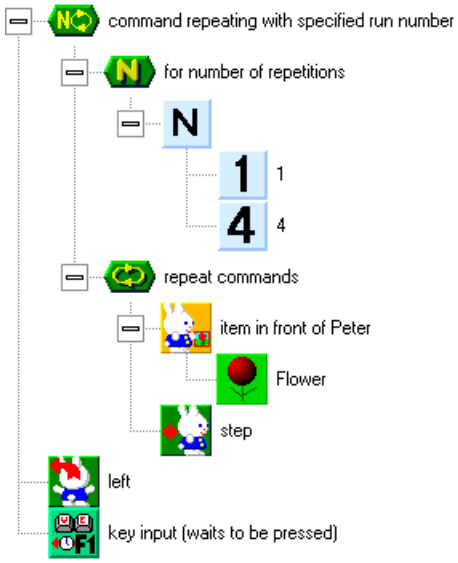
Now we specify the number of repetitions, for which the commands in the cycle will be executed. For this, we need the **numeric constant** element from the **calculations** group. Drag this element into the **for number of repetitions** element. If you expand the numeric constant element in the **Basic Elements** window, you will find elements for the **0** to **9** digits in it. Drag the digits into the numeric constant in the program to create a number of repetitions. The number that you create is read from the top. As the number of repetition, we will choose a number that is smaller by one than the garden width. That is, we will set the number to **14** — the constant will contain the digit **1** and the digit **4** under it.

There is also another way of specifying numbers. If the numeric constant does not contain any digit element, the number in the descriptive text of the numeric constant is used. There may be any notes behind the number. In our example, we could only type the text **14 — number of steps** as the element's description. Anyway, the previous way is more illustrative.

Now you can throw all of the remaining elements outside the cycle into trash (the upper right window), with the exception of one command for turning left, and the command that waits for a key to be pressed. You can move elements into trash with your mouse, but you can do it more quickly with your keyboard. Click to select the icon of the first element you want to delete, and then press the **Delete** key, until all of the unnecessary

elements are deleted. Be careful not to press the key for too many times, otherwise you would have to use the **Undo** button.

After the modifications, the program should look like this:



Notice that thanks to the elements' descriptions, the program can be read in quite a natural way: "*For number of repetitions 14, repeat commands: lay down a Flower item in front of Peter, then make a step.*"

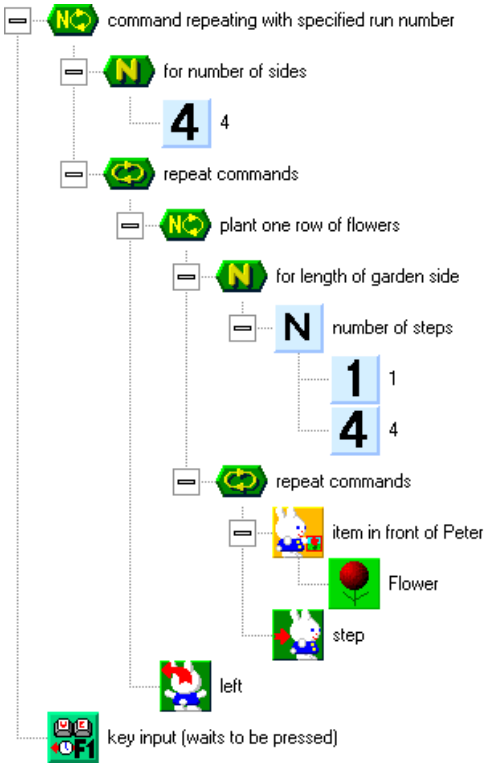
Run the program. Peter will go towards the right border, laying down flowers on his way. When he plants them along almost the entire bottom border of the window, he turns up and waits. He forgot to plant a flower at the beginning of the line (on the left), but his next attempt will surely be better.

Now we could copy the cycle for three times, as we need four sides. However, being good programmers, we will use a cycle again. Cycle is, in fact, a normal command, and so we can nest it inside another cycle.

From the **Basic Elements** window, we will drag another **command repeating with specified run number** element, and insert it in the very beginning of the program. As a number of repetitions, we will specify the number **4**. For numeric constants with one digit, we can use the digits alone. We will find them by the numeric constant element. This means that we will drag the **4** digit element into the **for number of repetitions** cycle parameter.

Into the **repeat commands** cycle body, we will drag the cycle for planting one side of the garden, which we have created before, and under it, a command for turning to the left. The command for waiting for a key to be pressed will be left in the very end of the program. A good programmer also adds comments to his work, so that the function of the program is clear to him or her even after a time; and not only to him or her, but also to everybody else who ever sees the program.

Here is the result of our hard work:



Run the program. Peter will go around his new garden, planting flowers on his way. This time he plants them all, and does not forget anything. Do you like his garden?

If you increase the number of steps in the inner cycle to e.g. **44**, you will see that Peter plants the flowers around really the whole of the garden now, and that he does not even mind that in the end of his way, he always hits his head against the wall. If he cannot perform an operation, he simply does not perform it, and nothing happens.

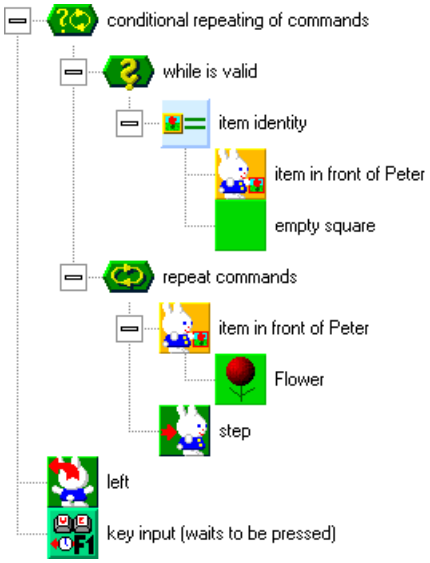
If you want to do some more experimenting, try to increase the number of repetitions in the main (outer) cycle. Peter will keep on walking around the border of the window. If the number of repetitions is too high, you might not want to wait until the end of the program. In such a case, quit the program by clicking the ✕ button (in the upper right corner of the window), or by pressing **Alt+F4**.

7 The Garden with a Condition

The **item in front of Peter** element can be used not only for laying an item in front of Peter, but also for detecting what item lies in front of Peter. We will use such testing in another method of planting the garden, based on a cycle with a condition.

In the **Basic Elements, Trash** window, find the **conditional repeating of commands** element in the **program control** group. Drag it to the beginning of the program from the last chapter (in front of all commands). Move the commands for making a step and laying down a flower in front of Peter into the **repeat commands** cycle body. Leave the commands for turning left and waiting for a key to be pressed after the cycle. You can discard the rest of the commands.

The **while is valid** cycle element tests a condition, which specifies how long should the commands in the cycle be repeated. Into the condition, we will put a test detecting whether there is an empty square in front of Peter. To assemble the condition, we will use the **item identity** element. It is in the **Basic Elements, Trash** window, in the **sheet** group. Drag this element into the **while is valid** cycle element. Into the **item identity** element, we will insert two elements that will be compared. The first one is **item in front of Peter**; the second is **empty square** (from the **Global Variables and Functions** window). Here is the result:



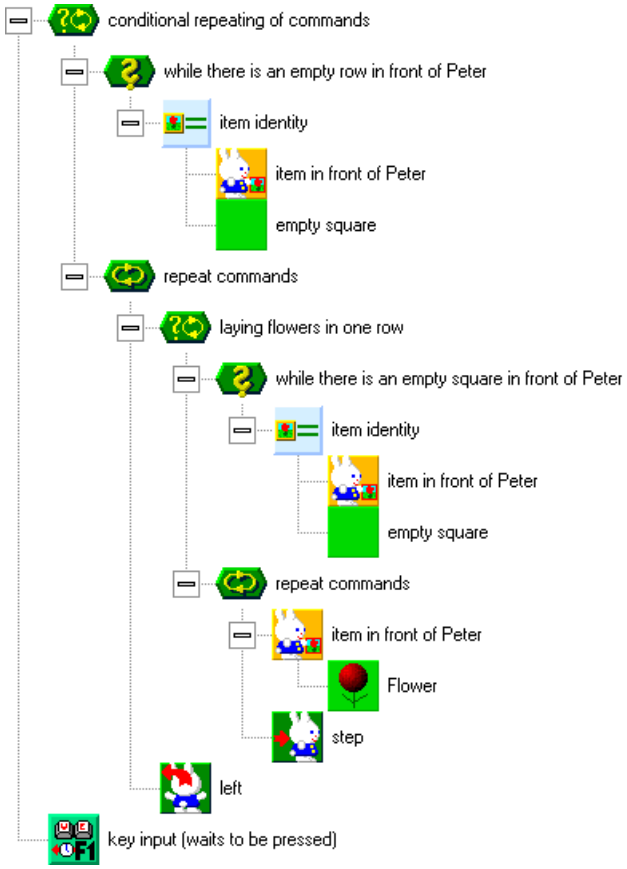
Try to run the program. Peter plants flowers to the right border of the window, and then he turns up and waits for a key to be pressed. If the function of the program is not clear to you, we can take a closer look at it.

How does conditional repeating work? The cycle elements descriptions tell us: "*while is valid (something), repeat commands (something)*". At a closer look, it means the following: In the beginning, the cycle detects, whether the condition is true. If it is, the commands in the cycle body are performed. Everything is repeated from the beginning then. The condition is evaluated again, and if it is true, the commands are performed again. If the condition is not valid, nothing is performed, the cycle ends, and the program continues by performing the commands after the cycle.

The program could be described in this way: In the beginning, the cycle asks the testing function that evaluates the condition: "*Is the condition true?*" The testing function here is the function for comparing items. It detects: "*Is there an empty square in front of Peter?*" If there is, it replies to the cycle: "*Yes, the condition is true.*" In that case, the cycle performs the commands in its body — Peter lays a flower and makes a step. This is repeated until Peter reaches the border of the sheet. The testing function detects that there is not an empty square in front of Peter now, and tells so to the cycle. The cycle does not continue. After that, Peter turns left, and the program pauses and waits for a key to be pressed.

After laying flowers in one row, Peter stays turned left, heading another row. We will test, whether there is an empty square in front of him, and if there is, we will tell him to plant another row. When he gets back to his original position, there will not be an empty square in front of him, but a flower that he has planted, and so he will stop.

This means that now we will take another **conditional repeating of commands** cycle. We will put it to the beginning of the program, and into its body, we will move (using the left mouse button) the previously created cycle and the command for turning left. The command for waiting for a key to be pressed stays at the end of the program. Into the outer cycle condition, copy (using the right mouse button) the condition testing an empty square in front of Peter (you do this by dragging the **item identity** element). Here is the result:



The program is ready; all that remains is to test it.

8 Peter Walks on Marks

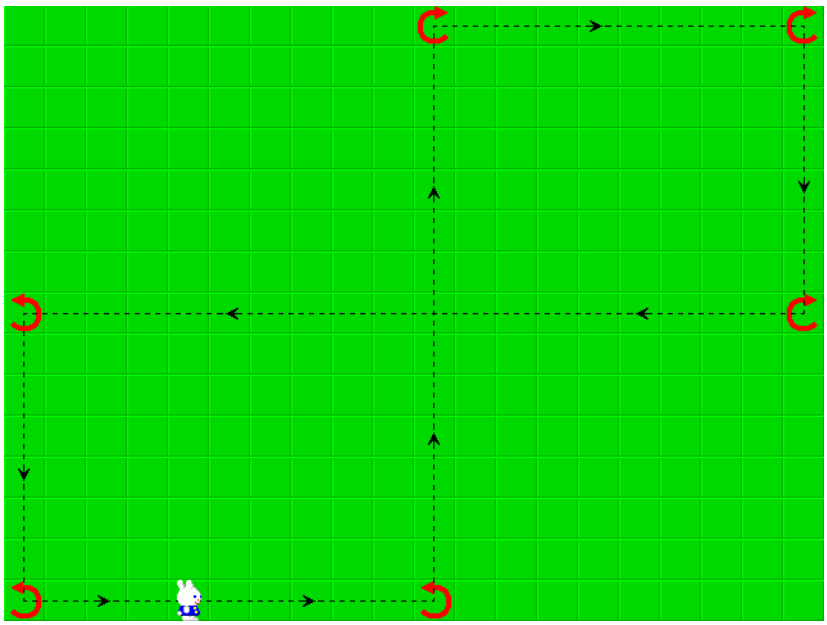
In another program, we will learn how to use conditional performing of commands. We will let Peter walk around the sheet, onto which we will put various marks. According to the marks, Peter will choose his direction.

First, we will create a new program. As you sure remember from chapter 4, a new program is created by clicking the **New** button in the program window. The new program will be called **Marks**.

In the program, we will need two items that will be used as marks. In the **Global Variables and Functions** window, copy (with the right mouse button) the **empty square** element into two new elements. Name them **Left** and **Right**. To remind you: You can assign a name by, for example, selecting the element by clicking its icon, and pressing **Alt+Enter** to edit the name.

Put the new items into editing mode by double-clicking their icons. Into the **Left** element, draw an arrow turned to the left; into the **Right** element, draw an arrow turned to the right, e.g. like this: and .

We will prepare the sheet. Double-click the **sheet** element. In the editing window, the program sheet appears. We will create a path for Peter. Put the **Left** and **Right** elements on the sheet in such a way that Peter can follow them in a closed track. Remember that Peter starts from the bottom left corner to the right. To put the items on the sheet, drag both of them from the **Global Variables and Functions** window to a place on the sheet first, and drop them there. Then you can move the items with the left mouse button, or copy them with the right mouse button. An item that is not necessary can be deleted by moving it out of the sheet. The sheet may look like this (Peter's path is indicated by the dashed line with arrows):



When the program sheet is prepared, we can start to assemble our program. This time we will begin with the main cycle. Peter has to keep moving along the marks, which means that the program will be based on a never-ending cycle. For this reason, we will prepare the **conditional repeating of commands** element. Actually, the cycle will not be never-ending; it will be possible to end it by pressing the **Esc** key. However, we will learn a few facts about the keyboard first.

When a key on the keyboard is pressed, it sends its numeric code to the computer, which is something like a serial number of the key on the keyboard. The computer converts this code into a character, which can be passed to a program, e.g. as a letter or a number. In our program, not only the characters, but also the numeric codes of the keys are available. The function for character input is used for typing texts from the keyboard, as we know it from common writing of texts (e.g. holding **Shift** generates capitals). The function for key input serves for controlling programs and games (even control keys, like **Ctrl**, generate key codes, although they do not generate characters).

When a key is pressed, its character or code is stored in a stack. The reason is that when the key is pressed, the program might not be ready to accept it. When the program is ready, it accepts the key from the stack. The key is deleted from the stack then. When you use keyboard input functions, you have to remember that when you load the character or the code from the keyboard, it is discarded, and the next loading returns something else.

Characters and keys have separate stacks, which are independent of each other. They are as separate data flows. One channel is used for characters, the other for key codes.

Now we can prepare the condition for quitting the main program cycle with the **Esc** key. We will use a function for the input of the key code from the keyboard. We know that it returns a numeric code, and so we will need a function for comparing numbers. Such functions are located in the **calculations** group under **comparisons**. We will use the **is not equal to** function. Drag it into the **while is valid** cycle condition.

The first element to be compared is **key input (does not wait for press)**. We will drag it to the comparative function from the **controls** group, **keyboard** sub-group. We have already used a similar element (waiting for a key to be pressed) in the previous chapters. The difference between these two elements is that this element does not wait for a key to be pressed. If no key code is ready, it returns the code of a situation where no key is pressed (it is the number **0**, but we do not have to know the value, as we have a **no key pressed** symbolic code).

The second element to compare is the **Esc** key. It is into the same group as the key input function, but it is nested deeper, under **keys** and **control keys**. This element is a numeric constant with the value of the **Esc** key code, which means that we do not have to know the code. Put this element under the key input function element in the comparative function.

What does the cycle do now? Read the notes besides the elements: "*While it is valid that key input is not equal to Esc key, repeat commands (something).*" This sounds rather complicated, but the meaning is clear, perhaps. The cycle will be repeated, until the **Esc** key is pressed on the keyboard.

Let us fill in the **repeat commands** cycle body. During his walk, Peter will decide his direction accordingly to the square in front of him. The decision will be made using a new element, **conditional executing of commands**. It is in the **program control** group. Drag it into the **repeat commands** cycle body.

When you insert the element for conditional execution of commands, you may notice that it contains another three elements. The first one is **if valid**. It is a test of a condition. We already used a similar element in the conditional cycle. The element tests a condition, and if it is true, the commands in the **then do** elements will be executed. If the condition is not true, the commands in the **else do** element will be performed.

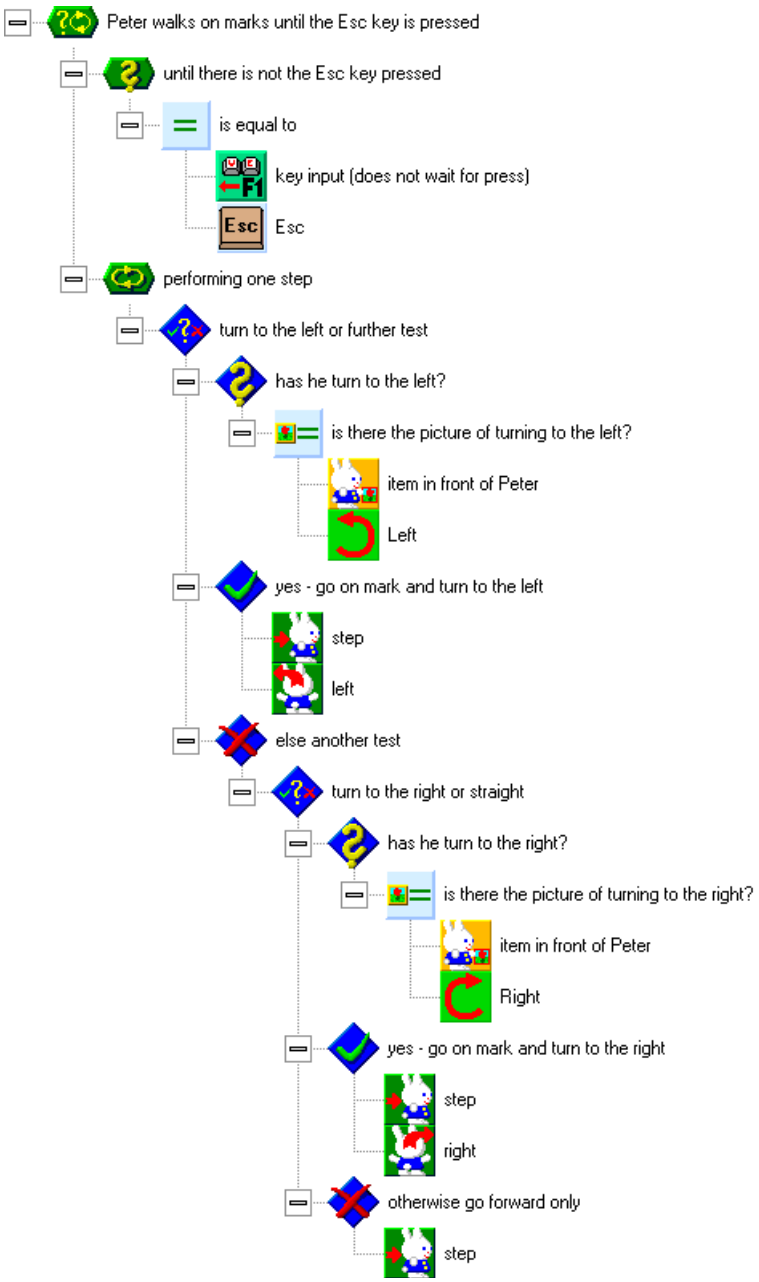
In the condition's test, we will detect if there is an item for turning left in front of Peter. We know a similar test from the previous chapter, and so it is clear that we will use the following elements: **item identity**, **item in front of Peter** and **Left**.

Into the first branch (**then do**), we will put two commands, **step** and **left**. The conditional command construction has this meaning now: "*If there is an item for turning to the left in front of Peter, Peter will make a step and turn left, in other cases, he will do something else.*"

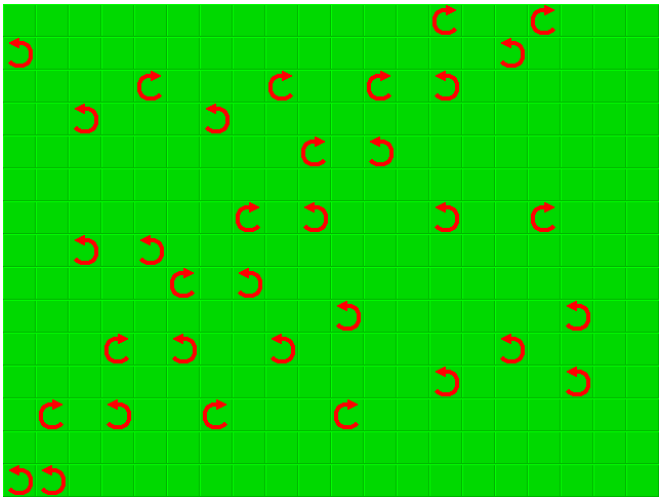
Now we will create that "*something else*". Using the right button, copy the whole construction of the conditional command to a lower place. Drag the newly created conditional command with the left mouse button, and drop it in the second branch of the first conditional command (the **else do** branch). This is a quick way to prepare the part of the program for the second case, turning to the right.

In the new conditional command, replace **Left** with **Right**. In a similar way, replace **left** with **right**. Now you probably know what Peter will do if there is not an item for turning left in front of him. He will detect, whether there is an item telling him to turn right. If there is, he will make a step and turn to the right. If not... What should he in fact do otherwise? We do not want anything more; it is enough if he makes a **step**. Copy the element for a step with the right mouse button from the second branch of the condition.

Did it look complicated? Don't worry — the program is ready (its picture follows). Run the program. If everything is all right, Peter will run along the marks and turn to the left or right on them. Do you want to make Peter go faster? Double-click the **Peter** element to edit Peter's sprite. Click the **Properties** button. A window for setting the sprite properties appears. Into the **Phases per Step** field, type **4**, and then press **Enter**. Run the program again. Peter will run like crazy now, but he will not gasp for his breath.



Try to prepare a more complicated track for Peter. Here is one example:



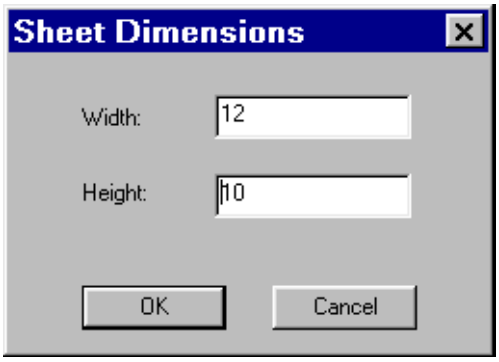
9 Peter in a Maze

Peter has wandered into a large maze. He cannot find his way out, and we have to help him.

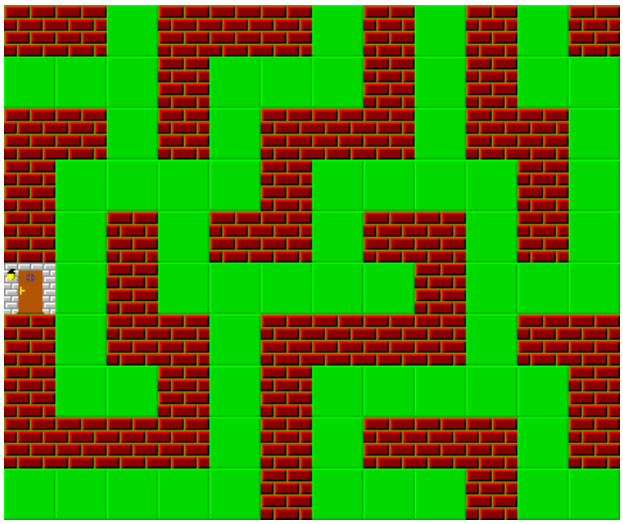
We will start by creating a new program. In the program window, click the **New** button. Type **Way** as the program name.

In the **Global Variables and Functions** window, we will prepare two items for the maze — a wall and a door. We could draw them, but we may use the items that are already prepared in the item library. The library is in the bottom right window called **Library of Variables and Functions**. There are data elements like number, item, or picture in it. Expand the **item** element, and in the **[examples]** groups, **Cottage** subgroup, find the elements labeled **Door** and **Wall**. Drag them into the **Global Variables and Functions** window. This is a quick and easy way to create new items, which have already been named and drawn.

Now we will create the maze sheet. Double-click the **sheet** element. We will make the sheet smaller, so that the maze does not have to be too big. Click the **Dimensions** button. A window for specifying sheet dimensions appears. Type **12** as the width and **10** as the height.



Use the **Door** and **Wall** elements to create the following maze:



The program will be based on a **conditional repeating of commands**. We want Peter to repeat his steps around the maze until he finds the door. In the cycle condition, we will use the **item on Peter's position** element. It is similar to the **item in front of Peter** element, but it applies to the square, on which Peter is standing. It can be found in the **Peter - extension** group.

The cycle condition could be expressed like this: "*If it is not true that the item on Peter's position is the door, repeat…*" The word *not* tells us that we need the opposite of the item identity test. For this, we have an element called **is not valid that**. The element is called **logical negation**, and its function is to invert the result of the comparison from **true** to **false**, and from **false** to **true**. It is located in the **calculations** group, **logic operations** subgroup.

First, we will place the **is not valid that** element into the **while is valid** cycle condition. We will attach the **item identity** element, and insert the **item on Peter's position** and **Door** elements into it.

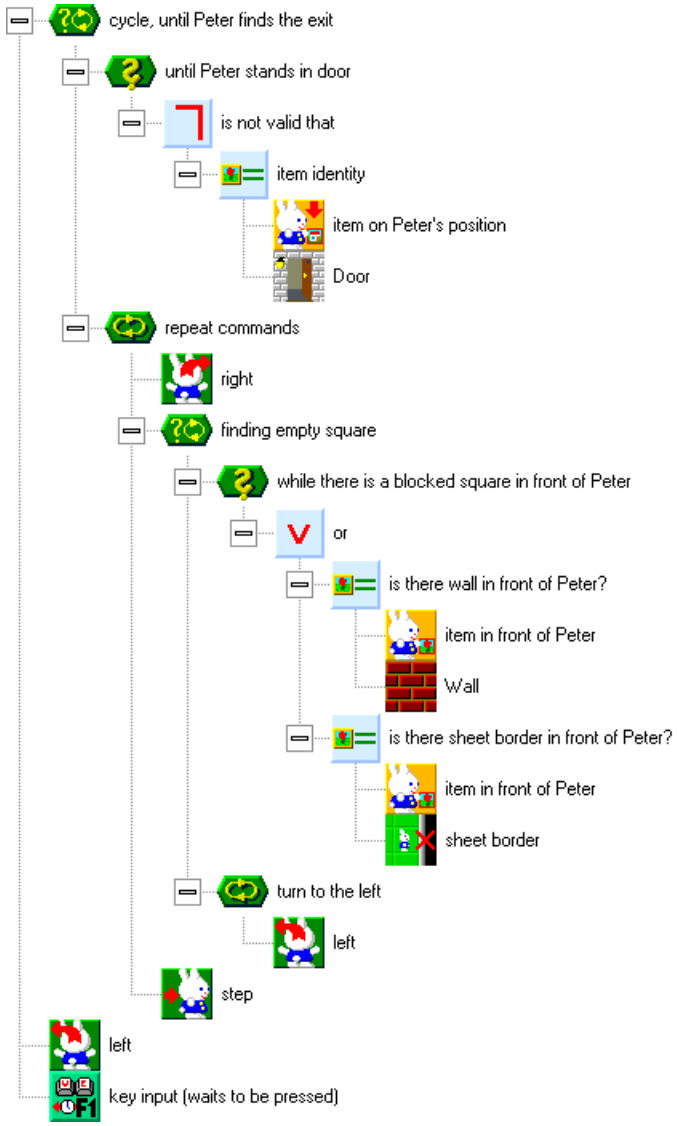
In the cycle body, we will ensure that when walking around the maze, Peter will stick to the border on his right. In the beginning of the cycle, we will tell Peter to turn right by using the **right** command. The following command will be a **conditional repeating of commands** cycle. This will tell Peter to turn to the left, until he will find a square onto which he can go. For this, we will use the **left** command.

In the second (inner) cycle, we will test if there is a blocked square in front of Peter, i.e. if he should turn further to the left. The first blocked square is the **Wall**. Peter also cannot go through the border of the sheet. This can be tested by a special element called **sheet border** (located in the **sheet** group). It is special by the fact that it is not a data variable, but a constant, and so you cannot edit it. Its value can be kept in variables and tested, but it cannot be viewed by putting it on the sheet.

We have two tests of items in front of Peter, and we need to express this: "*As long as there is a wall or a sheet border in front of Peter, repeat left.*" The word *or* is a logical element again — **or**, and it is called **logical sum**. It is located in the **calculations** group, **logic operations** subgroup. We will insert the **or** element into the cycle condition and add two comparisons of items in front of Peter, with the wall and the border.

Peter is facing an empty square now, so we have to add one last element into the main cycle — **step**. This tells Peter to go to the empty square.

Finally, we will tell Peter that when he finds the door, he should turn towards us and wait until we press a key. Behind the main cycle, add the **left** and **key input (waits to be pressed)** elements. The resulting program is shown on the following page.



Run the program. Peter walks around the maze, until he finds the exit. It is just behind the corner on the left side, but he does not know it and tries to find the exit from the right, so it takes some time.

Let us replace the **left** and **right** elements. This changes Peter's search method, so that he will stick to the wall on his left. In our maze, he finds the exit much sooner.

10 Keyboard Control

Let us stay with the **Way** program from the last chapter. We will edit it, so that we can control Peter in the maze with our keyboard. We will try several methods of control, and we will keep all of them in the program for the sake of comparisons. We will also keep the method, in which Peter searches for his way himself.

From the **program control** group, drag the **group** element into the main program cycle. Move all of the elements that are in the main cycle now (the element for turning right, the cycle for finding a free square, and the step command) into this group. Collapse the group and add a note to it saying **Peter Walks Himself**. This hides the automatic search method, so that it will not be mixed with other methods that we will create. Try to run the program for testing purposes. It should run just as before the modification.

Click the new group icon to select it. On the toolbar, click the **Turn Off** button. The group and all of the elements in it become gray. This function enables you to turn off parts of a program when you don't need them. When you run the program, you see that Peter stands in the bottom left corner and nothing happens. The program behaves as if the group and its commands were not in it at all.

We will assemble a new part of our program inside the main program cycle, right after the previous group. We will use a new element — **multibranch control structure** (sometimes also called "rake" among programmers). It is in the **program control** group. Drag it into the main cycle under the **Peter Walks Himself** group.

The multibranch element branches programs into several parts, depending on a variable or expression value. Usually, branching is based on numeric values. Later we will also use it for other types of data, such as texts.

This element contains another four elements. The first element, **for value of expression**, contains a variable or an expression, which is the basis for branching. We will put the **key input (waits to be pressed)** element here. It will pass a numeric code of the key pressed to the branching structure.

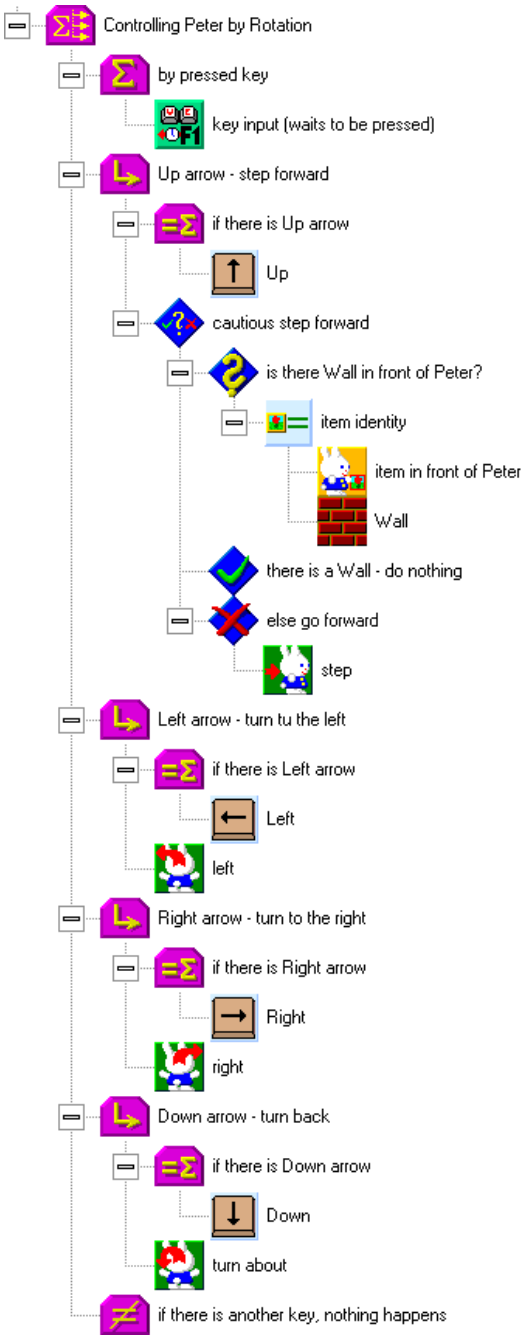
Two connected elements follow — **execute commands in case of** and **expression is equal to**. This is one branch of the construction. Into **expression is equal to**, you place a value, for which the commands in the branch will be executed. There can be any number of branches, as well as any number of values tested in a branch. Copy the whole of the **execute commands in case of** branch for three more times.

Now we have four branches of the construction. Into the tested value in the first branch, put the **Up** key code from the **controls**, **keyboard**, **keys**, **control keys** group. This branch will be executed when you press the up arrow key. Insert a step as the branch command. We will ensure that Peter does not walk through a

wall. It is not necessary to check the sheet border; Peter cannot run away from the sheet. Instead of a simple step, we will use a conditional command. In the test, we will compare items to see whether there is a wall in front of Peter. If there is, nothing will happen. If there isn't, Peter will make a step.

Into the following three branches, insert the **Left**, **Right**, and **Down** keys as the tested values, and **left**, **right**, and **turn about** as the commands.

The last element in the branching construction is **else execute commands**. As its name suggests, the commands here will be executed, if the tested value is not found in any branch. In our case, this element will be empty.



Run the program. Peter stands in the bottom left corner. If you try the cursor keys (arrows), you will see that Peter turns to the left, right, and back, and that he makes a

step forward. You can lead Peter around the maze. When you get to the door, Peter turns to you, and the program ends after the next key is pressed.

This method of control is typical for cars. It is used when we need to turn an object exactly to a direction we want, and when the speed of movement around the sheet is not that important.

If you need to move quickly and easily, you can use the following method of control. It does not control the turns of the character, but walking in the direction of the arrow.

Using the right mouse button, copy the previous branching construction once more to the bottom. Disable the original construction by using the **Turn Off** button. Label the new construction **Controlling Peter by Directions**. Delete the commands for turning to the right, left, and back from the branches.

We will edit the first branch of the construction (for the up arrow) to make Peter turn up and make a step. The step is already prepared. Inside it, there is a test of a wall in front of Peter. The direction into which Peter turns can be set by the **direction** element. It is located in Peter's extensions. As a parameter, add the **up (1/2 pi, that is 90 degrees)** element from the **calculations** group, **angle, direction** subgroup.

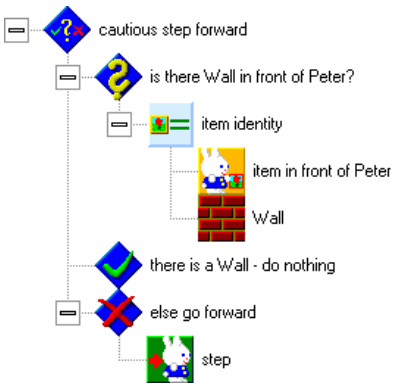
Create the direction setting commands in the remaining branches as well, changing the directions accordingly to the arrows. The command for the cautious step might be copied from the first branch, but we will use a **function** instead.

What is a function? Function is an element that contains a part of a program. Instead of several occurrences of the same part of a program, we will use a function to perform the part of the program several times. The program will be easier to read and modify.

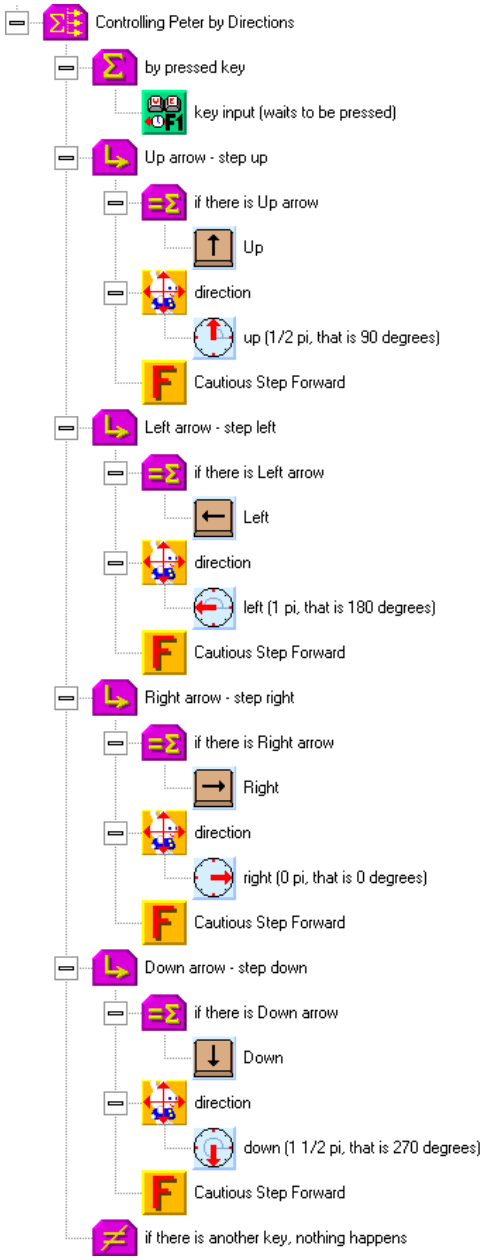
To create a new function, drag the **function** element from the **Library of Variables and Functions** window into the **Global Variables and Functions** window. Label the function **Cautious Step Forward**. When you double-click the function icon, an empty editing window appears, as the function does not contain any command yet.

We could create the step command in the function, but there is an easier way, as it is already prepared in the program. Switch back to the main program function (using the **Previous Edit** button), and find the conditional command with the wall test and the step forward (it is in the branch for the up arrow). Click the conditional command to highlight it. Click the **Copy** button (or press **Ctrl+C** on the keyboard). The highlighted part will be saved in the clipboard for data transfer.

Return to the **Cautious Step Forward** function (by clicking the **Next Edit** button). Click the **Paste** button (or press **Ctrl+V**). The transferred part of the program appears in the window.



Now we can delete the conditional command from the branch for the up arrow, and replace it with the function for the careful step. We will also add it into the remaining branches behind the direction setting commands.



You can run and test the command. You will see Peter walking in the direction of the arrow pressed.

We will try one more method of control, which can be used for more complicated and complex games. You may have noticed that if you hold a key for a while in the previous method, Peter walks on even after you release the key. This is because the codes of the keys pressed are generated more quickly than Peter walks, and so they are kept in the key stack. This can be solved by the **flush out of key buffer** command, which clears the keys out of the stack. We will use it in the beginning of the actions for every valid keyboard character, before Peter's steps.

Another thing that can be problematic in the previous two methods of control takes effect if you control quick actions or need to control graphics sharply. In the Peter sprite's settings, try to change the **Phases per Step** value to **1** (after the test, return it back to **8**). Peter will jump the squares quickly. If you hold the key on a longer free path, Peter jumps one square, waits a little while, and then jumps to other squares quickly. This is caused by the way that the keyboard generates codes — there is a pause after the first code, and then there are quick repetitions. This is useful when you write text, but may be a bit of a trouble when playing games.

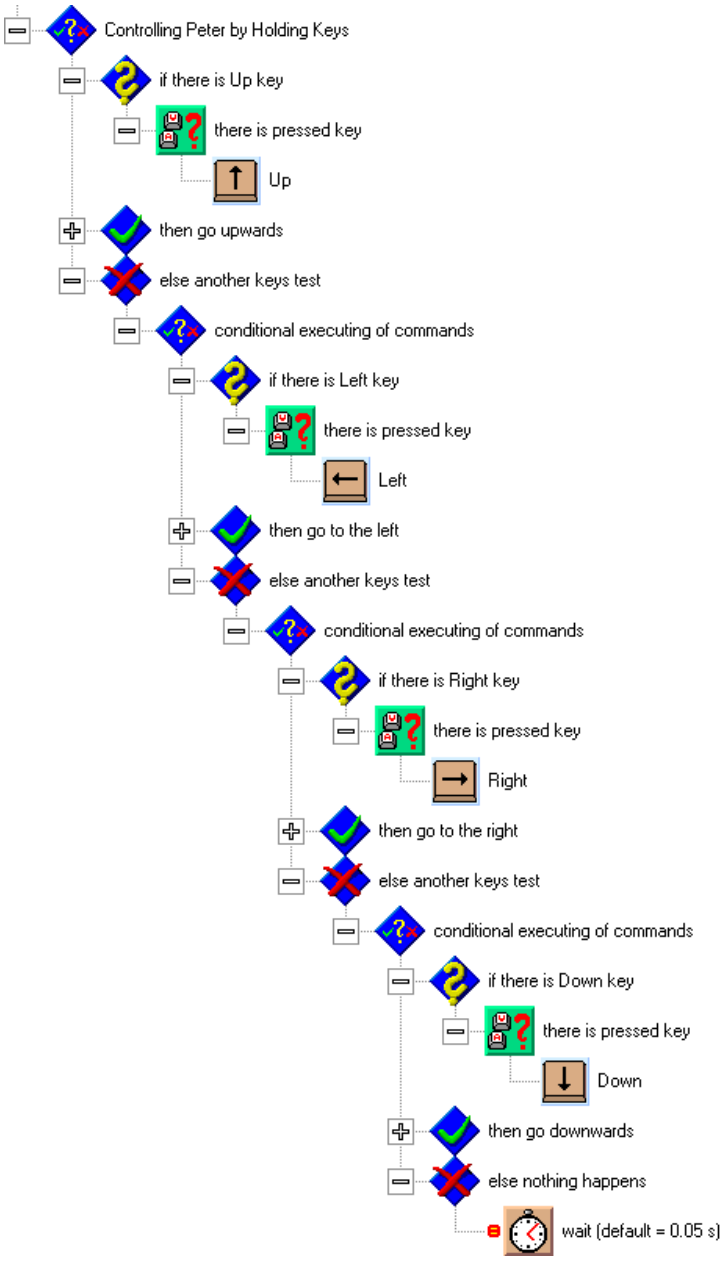
Our request is that Peter should move when we hold a key, without the initial pause, and without the inertia after the key is released. This can be done by using the **there is pressed key** function from the **keyboard** group. The function tests if the appropriate key is pressed, and returns a **yes/no** flag.

First, click the **Turn Off** button to turn off the previous method again. Under the method, prepare a **conditional executing of commands** element; you can call it **Controlling Peter by Holding Keys**. Into the **if valid** condition test, insert the **there is pressed key** element, and attach the **Up** key as its parameter. Into the **then do** branch, insert the commands for making a step upwards, as in the previous control method. The construction now reads: "*If the Up key is pressed, then turn up and make a cautious step, else test other keys.*"

In the **else do** branch, create the same construction, this time for going to the left. In its invalidity branch, create a construction for going to the right, and finally, create a similar construction for going down. Into the invalidity branch of the going down construction, insert the **wait (default = 0.05 s)** command, which can be found in the **program control** group. The small equals sign in front of the icon indicates that there is an optional parameter of this command, which means that it is not necessary to add one.

Why a wait command? Windows is a multitasking operating system, which means that several programs can run at once. Each program should ensure that it does not consume an unnecessary amount of the computer's performance. A good programmer can be distinguished by a considerate behavior of his or her programs in their environment. Advanced users can run the **System Monitor** program and monitor the load on the processor. An inconsiderate program consumes almost all of the computer's performance, and causes the computer to be hard to control.

The picture shows the complete method of controlling Peter. The constructions for steps are collapsed, as we already know them from the previous method.



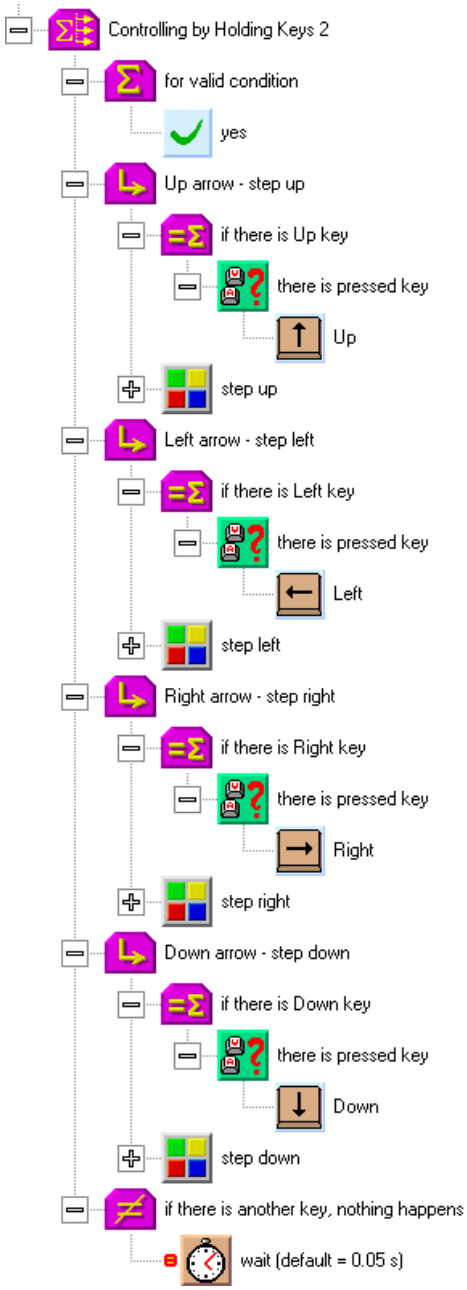
A disadvantage of such a long chain of conditions is that it can be hard to orientate in. We will try another possibility of building a similar construction.

Click **Turn Off** to disable the last method. Copy the **Controlling Peter by Directions** method and rename it to **Controlling by Holding Keys 2**. The conditional construction worked in such way that just one of the branches was executed, depending on the key pressed. Delete the keyboard input function from the **for value of expression** element, and replace it with the **yes** element (from **calculations**, **logic operations**).

In the tested values of the individual branches, replace key codes with combinations of a key code and the **there is pressed key** element. Do the following: Click the key code icon to select the key code element. Click the **Cut** button (or press **Ctrl+X**) to

move the key code element into the clipboard. Insert the **there is pressed key** element instead of the key code element. Click the **Paste** button (or press **Ctrl+V**) to attach the key code element to the pressed key test. Finally, insert the **wait (default = 0.05 s)** element into the **else execute commands** branch, so that the program does not overload the computer in moments of inactivity.

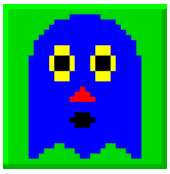
The resulting construction is on the following picture. How does it work? In the beginning, the program detects that there will be a test of a logical value of validity in its branches. It goes through the individual branches and searches for a matching value, that is, for a valid condition. It performs the commands in the branch it finds, and continues behind the end of multibranch structure (not paying attention to other conditions that might be valid, nor to other branches). In another words, the first (and only) branch with a valid condition (a key pressed) is performed, else the very last command is performed — wait.



11 Plenty of Monsters

We will try to make a little game. You would not believe what has happened. Peter has found monsters in the maze. However, do not worry, we will arm him, and he will handle them. We will continue with the **Way** program from the previous chapter, in which the last method of control is turned on — **Controlling by Holding Keys 2**. The other possibility is to open the **Monsters** program, which is prepared as a sample program.

First, we will prepare the monster. In the **Global Variables and Functions** window, make a copy of the **empty square** element. Rename it to **Monster** and draw a monster into it, e.g. like this:



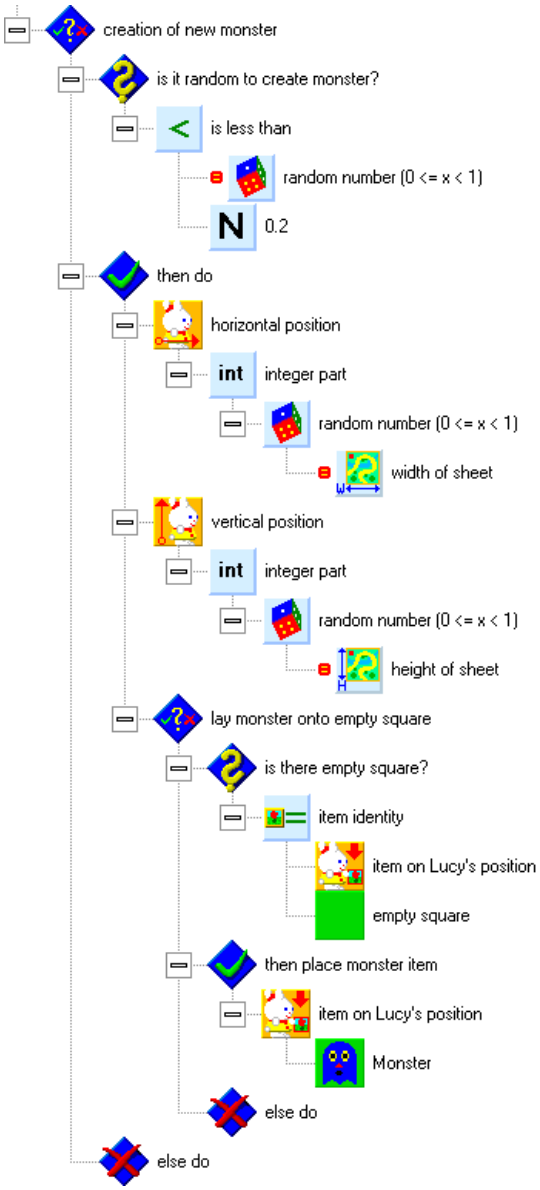
Monsters will be generated randomly. Add the **conditional executing of commands** element into the main program loop. Into the **if valid** condition test, insert the **is less than** comparative function (from the **calculations** group, **comparisons** subgroup). The first parameter for comparison will be the **random number (0 <= x < 1)** function (from the **calculations** group, **functions** subgroup). The second parameter will be a **numeric constant** set to **0.2** (either as the element text or using the digits and decimal point elements). In a while, we will add the monster creation construction.

What will the random function do? The elements texts tell us: "*If it is valid that a random number is less than 0.2, create a monster.*" The random number is a decimal number with a random value between zero and one. **0.2** is one fifth of one. The random number is less than **0.2** in every fifth case. This means that the monster will be generated in one fifth of cases.

To create the monster, we will use Peter's friend — Lucy. First, we will specify a random place where the monster will be created. Into the condition validity branch, drag the **horizontal position** and **vertical position** elements (from the **Lucy** group, **Lucy - extension** subgroup). Add the **integer part** function to both elements (from the **calculations** group, **functions** subgroup), as the squares' coordinates are integers. Add the **random number (0 <= x < 1)** function to the integer elements. The random number can have a parameter specifying its range. For example, if you add 10, the random number is generated in the range from 0 to 10. We will add the **width of sheet** element to the random number for the horizontal position, and the **height of sheet** element for the vertical position (both elements are from the **sheet** group).

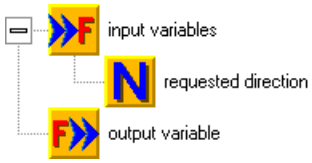
Let us take a closer look at the calculation of the **horizontal** random position. The referential element in the horizontal direction is the **width of sheet** . This element returns a number indicating the width of the sheet as a number of steps (squares). In our case, it will be 12, which is the width of sheet that we had set before. This number is passed to the **random number (0 <= x < 1)** function, which generates a number in the range from zero to the width of sheet (without the border value), that is, a number between 0 and 11.99999999. The random number is passed to the **integer part** function, which truncates the part of the number behind the decimal point and returns only its integer. This creates a number between 0 and 11, which are the coordinates of the first and last square in the horizontal direction. You may have noticed that the squares are numbered from the bottom left square, starting with zero.

Now we have Lucy on a random position on the sheet. We could lay the monster item onto the sheet now, but first we have to verify that the square is not occupied, e.g. by a wall. If the square is not free, nothing will be performed and no monster will be created. Here is the result; try to run the program:

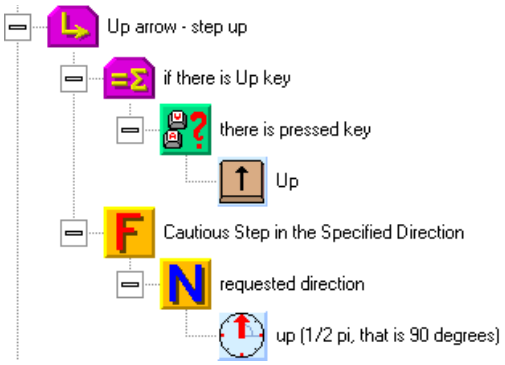


Now we will improve the routines for controlling Peter's steps. In the **Global Variables and Functions** window, use the right mouse button to copy the **Cautious Step Forward** function. A new function called **Cautious Step Forward 2** is created. Rename it to **Cautious Step in the Specified Direction**.

Double-click the new function to edit it, and look at the bottom left window. It is called **Local Variables and Functions**. We don't have to worry about the function of this window; we will only use the **input variables** element. From the **Library of Variables and Functions** window, drag a new numeric variable **number** into this element, and call the variable **requested direction**.

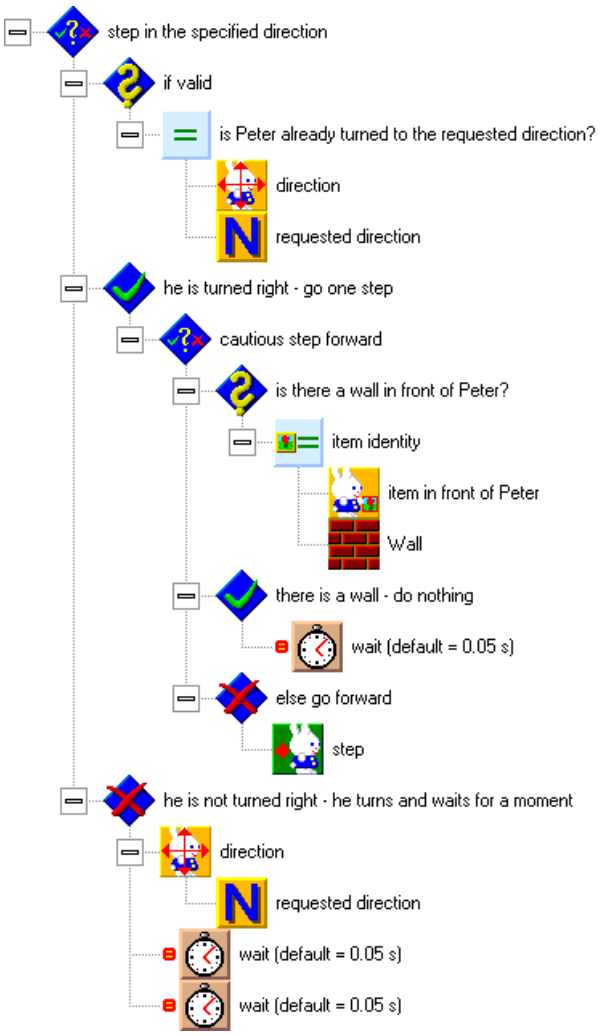


Switch back into the main program function. In the **Controlling by Holding Keys 2** construction, find the first branch (for the Up arrow). Insert the new function called **Cautious Step in the Specified Direction** here. When you drop the function, you can see that the numeric variable called **requested direction** is connected to it. It is the one that we have just created. It will pass the requested direction of the step to the function. From the direction setting command, drag the **up** element to the function parameter. You can discard the remaining commands from the branch (the direction setting and the **Cautious Step Forward** function). Adjust the branches for the remaining directions in a similar way. This will be the construction for the step upwards:



Now we will prepare the contents of the **Cautious Step in the Specified Direction** function. Switch into it. First, for testing purposes, try to add the command for setting Peter's direction, and add the **requested direction** variable as the parameter. This restores the original functionality of the program, and we can verify that we did not make any mistake. Run the program and test it, it should work as before. If everything is all right, you can delete the direction setting command.

If we used the original control method for shooting, it would be unpleasant that we would not be able to turn to the target without making a step towards it. For this reason, we will improve the control. If Peter is not turned in the requested direction, he will turn first, and only after that, he will go. By pressing the key once, we will turn Peter around, and by holding the key, we will tell him to go. Edit the function accordingly to the following picture.



The function begins by comparing Peter's actual direction with the requested direction, which is passed as the function parameter. This way, we will check if Peter is already turned to the requested direction.

If Peter is turned in the appropriate direction, he can make a step forward. However, he will do so cautiously. First, he tests if there is a wall in front of him. If there is not, he can make a step. If there is, he will stay on his place, and the wait command will be executed. The main program loop has to last for at least one wait interval, so that new monsters can be generated evenly. One wait interval is performed in the main loop, if no key for movement is pressed. The waiting when a key is pressed is handled in the movement function. The program does not wait after the step command, the step ensures the waiting internally.

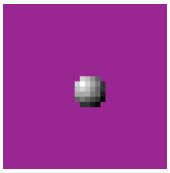
If Peter is not turned to the requested direction, he turns, and the program waits for a while. The waiting ensures that Peter does not start to go when a key is pressed shortly.

Run the program and test its control. Pay special attention to testing Peter's turning on his place and changing directions during walking.

Now we will handle the shooting at the monsters. Once again, we will use Peter's friend Lucy. Lucy will be the shot. She will surely not mind and will be glad to do this for Peter.

Double-click the **Lucy** sprite in the **Global Variables and Functions** window to edit it (note — sprite is a moving animated object). You will see a sheet with 4 x 5 pictures of Lucy. Click the **Properties** button. A window for setting the sprite properties appears. Change the **Phases per Step (0 = Immediately)** setting from **8** to **2** (the shot will fly quickly), and change the **Moving Phases** settings from **4** to **0** (the shot does not have to change its appearance during the flight). Press **Enter** (or click **OK**). The pictures of the sprite have changed, so that Lucy is only in one column now.

Drag the first picture of Lucy with the left mouse button, and drop it outside the pictures sheet. The picture disappears; we have deleted it from the sprite. Double-click the freed empty square. The sprite picture editor appears. Draw a picture of the shot — for example, a small gray ball (use the **sphere** tool and white color):



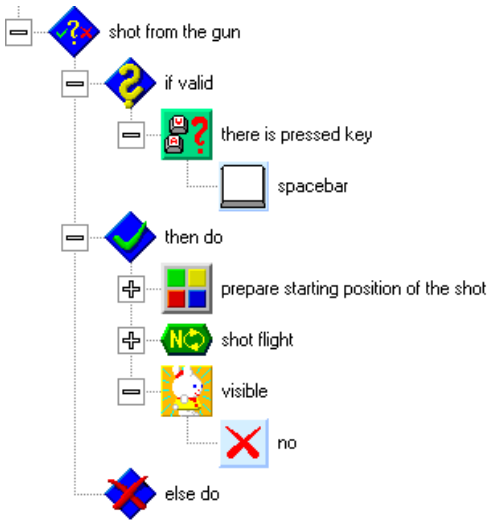
Keep the original violet color as the background. It is a transparent color, through which the original contents of the square around the shot can be seen. In the color picker of the editor, it is the color in the upper left corner.

Click **Previous Edit** to switch back to editing the Lucy sprite. Drag the first modified picture with the right mouse button to copy it into the remaining sprite squares. Later you can move the pictures, so that the shots will fly exactly from the barrel of the gun.

Test the modified sprite now. Click the **Test** button. A window with a green sheet appears, with the sprite of the shot in the middle. Click somewhere into the sheet, and the shot moves to the specified location. You can quit the test by clicking **Cancel**.

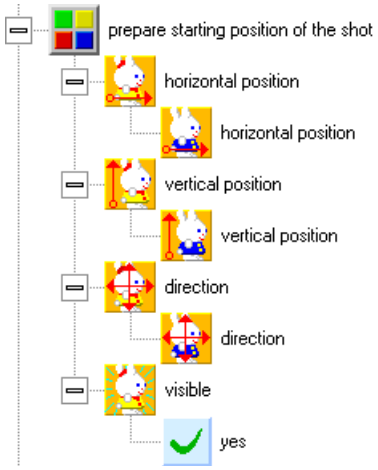
The shot is prepared, and now we have to handle its control. Return to the main program function. Into the main loop, right behind the construction for generating monsters, insert a new **conditional executing of commands** element, and call it **shot from the gun**. As shooting will be activated by the spacebar, insert the **there is pressed key** function with the **spacebar** element (from the **keys** group, **character keys** subgroup) into the **if valid** condition test.

The whole construction that handles shooting is on the following picture. We set the shot (Lucy) in the starting position, then the shot flies, and when it reaches a target, we turn it off.



In the beginning of the shot's flight, we set its position and direction accordingly to those of Peter, and then we make it visible. We know the position and direction elements, but what is a visibility element? All sprites (moving objects, including Peter and Lucy) have two basic states — visible and invisible. When in the visible state, the sprite is animated, and moves slowly. If it is invisible, it moves to a new position immediately. Peter becomes a "Super Peter", as quick as lightning.

Lucy's visibility will be set by the **visible** command. As the sprite visibility parameter, we usually use a logical constant **yes** or **no**; this way, we switch between the "Slowcoach" and "Superman" modes. The activation of the shot is on the following picture. For the sake of readability, it is in a separate group.

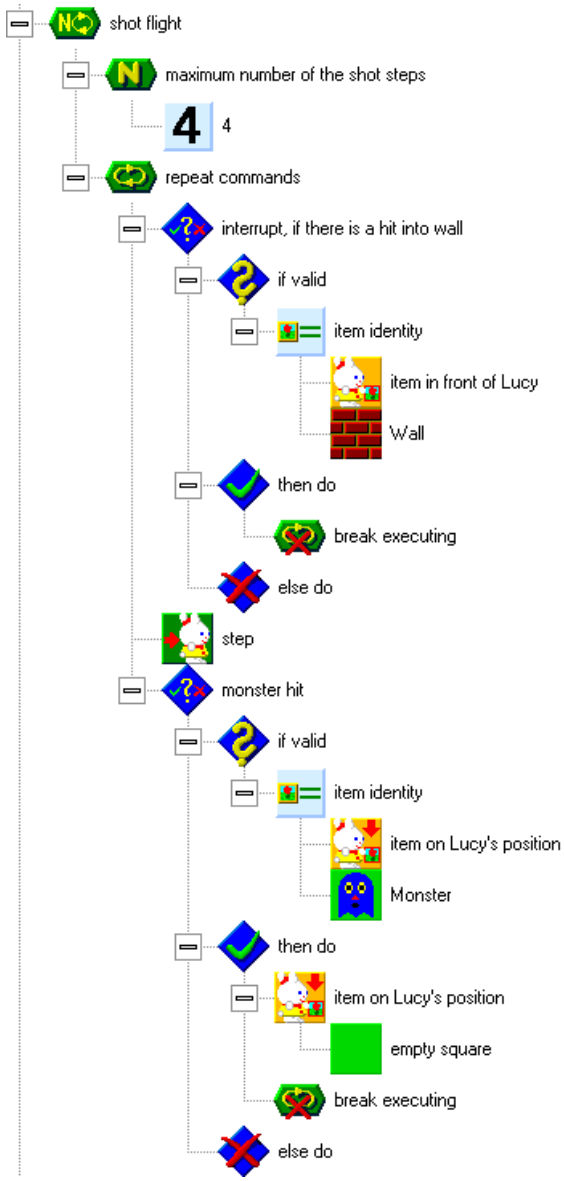


Now we will handle the movement of the shot. Behind the group for preparation of the starting position of the shot, add a **command repeating with specified run number** cycle and label it **shot flight**. As a number of repetitions, type **4**. This is the maximum distance that the shot can fly.

The flight will be interrupted, if the shot hits a wall. For this reason, in the beginning of the cycle body, add a condition testing if there is a **Wall** item in front of Lucy. If there is, the execution of the cycle will be interrupted by the **break executing** command (from the **program control** group). After we check for the wall, we can add the **step** element for the shot's movement.

When the shot moves one step, we will test if it has hit the target. We will use a conditional command that will test, whether the item on Lucy's position is a **Monster**. If it is, the monster will be deleted by laying down an **empty square** item, and the cycle of the shot's flight will be interrupted. Some time in the future, you can add other elements here, such as a hit counter or a monster's moan.

The whole construction for the shot's flight is on the following picture. Add an element for turning off the visibility of Lucy behind it, and you can test the program.



If you want to make further improvements, you can equip Peter with a gun. You can also add a sound of the shot by using the **play sound** element. From the bank of sounds, drag e.g. **[examples]\Weapon\Rifle and Pistol\Pistol** sound into the **Global Variables and Functions** window, and use it in the play sound feature. You can also add a moan of the hit monster — e.g. the **[examples]\Human\Shouts\Au 2** sound.

12 Feeding the Snake

Our next program will be a game called **Snake**. You have probably already seen such a game before. A snake moves around the sheet, eating food, and it gets one piece longer with each gulp. We will improve the game. The snake will eat various kinds of food, and its pieces will vary accordingly to what it has eaten.

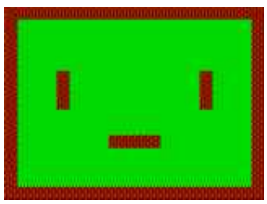
Create a new program called **Snake** (or **Snake 2**, if a program with this name already exists). An empty program opens. In a while, we will fill the program window with a fine new game. If you want, you can open the **Snake** program, which is prepared among Peter's sample programs.

We will prepare the sheet by enclosing it with a wall. From the item library, drag the **Wall** item (from the **[examples]\Cottage** group) into the **Global Variables and Functions** window. Double-click the **sheet** element to invoke the main sheet editor. Put the **Wall** item into the upper left corner.

We will look at a new sheet property. Each square contains, apart from an item picture, five logical flags and three numeric values. A logical value can keep a **yes**/**no** state, which can be tested. In our program, we will use the first flag to signalize that there is a wall on the square. We won't have to evaluate the item on the square, we will just ask the flag: "*Is there a wall on the square?*" We can even use several variants of walls.

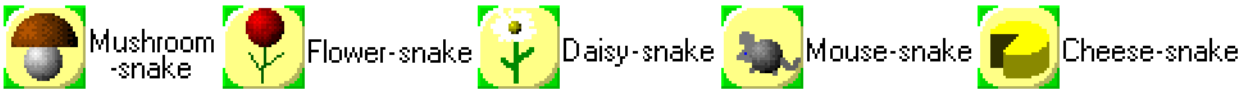
The sheet editor toolbar contains (on the right side) a drop-down box for types of editing. Select the **Flag 1** function here. The appearance of the sheet changes; a little dark red square appears in the upper left corner of each square. The little square is a state indicator for the first flag in the square. Now it is off (its state is "**no**"). Click the **Editing** button on the toolbar. This activates the square contents editing mode. Click the square in the upper left corner with the **Wall** item. The little square changes to light red color and a check mark appears. Flag 1 on the square is now on (its state is "**yes**"). Turn off the editing mode by clicking the **Editing** button again.

Now we want to raise a wall around the whole sheet. There is an easier way to do it than repeated copying with the right mouse button. Click the **Filling** button on the toolbar. Press and hold the left mouse button on the wall item in the upper right corner of the sheet, and drag the mouse to the upper right corner of the sheet. Release the mouse button there. The part of sheet that you have selected by this will be filled with the starting item. Fill the remaining borders in a similar way. When you finish, click the **Filling** button again to turn it off. You can also add walls inside the sheet, e.g. like this:



Prepare food for the snake. Create five new items and draw different kinds of food into them. For example:

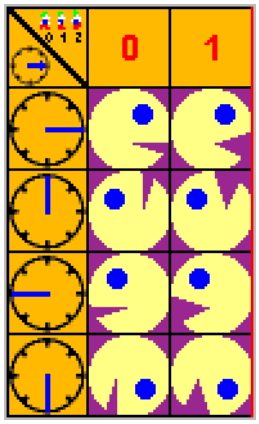
We will create the pieces of the snake. You can copy the food items and modify them like this:



In a similar way, create three items with pictures of poisons, and one piece of the snake with a picture of a skull:

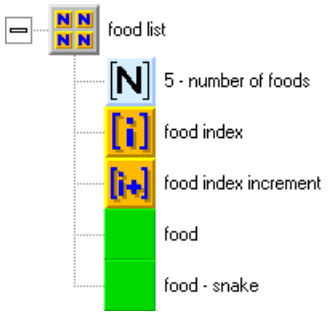
Our last graphical task is to create the snake's head. We will use Peter for this. Double-click the **Peter** element in the **Global Variables and Functions** window to edit it. Click the **Properties** button to open the window with sprite properties. Change the **Delay Between Phases** value from **55** to **165** (time between animation phases), change **Phases per Step (0 = immediately)** from **8** to **0** (the sprite moves immediately), and change **Standstill Phases** from **1** to **2** and **Moving Phases** from **4** to **0**. Close the window by pressing **Enter**. The Peter sprite has four directions and two standstill phases now.

Discard the first picture of Peter on the upper left. Double-click the picture to edit it, and then draw a snakehead looking to the right. Click the **Previous Edit** button to return into the sprite editor. Using the right mouse button, copy the first picture into the second picture in the same row. Edit the head, so that the mouth is open. Copy both pictures into the remaining rows for other directions. Turn the heads on the pictures accordingly to the directions in the sprite rows. You can do this by using the **Edit** function. Finally, test the sprite by clicking the **Test** button.



We have several food items in our program, and we might want to add more of them in the future. With our present knowledge, we would have to handle each type of an item separately, which would not be very elegant. For this reason, we will learn how to use a new element — a **list**.

At first, we will prepare the list, although we don't know its function yet. From the **program control** group, drag the **list** element into the **Global Variables and Functions** window. Call it **food list**. When you expand it, you will see three elements. Rename the first one, **number**, to **5 — number of foods**. The second element, **index**, should be renamed to **food index**. The third one, **index increment**, will be called **food index increment**. Add two new items in the list, and call them **food** and **food — snake**.

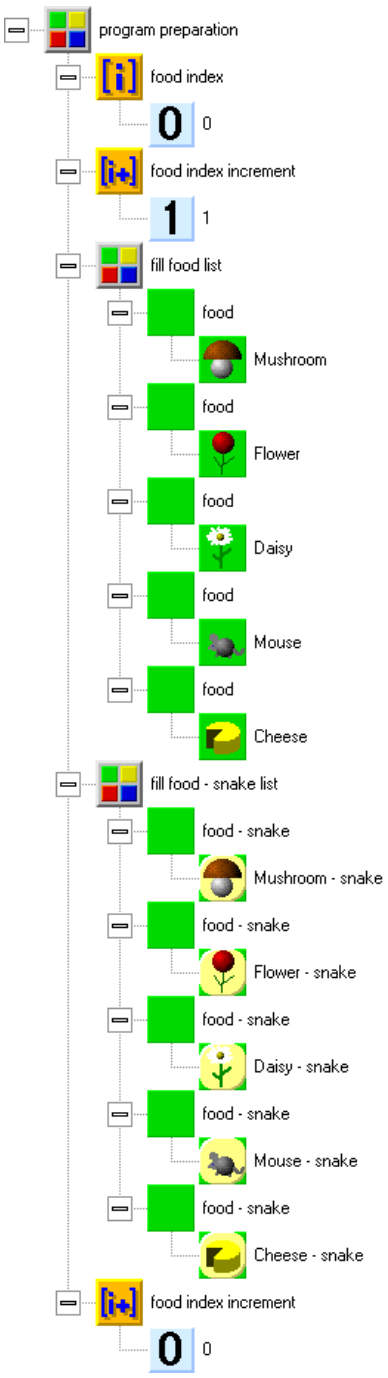


Now we will explain what is a list. A list is something like a book. All pages in the book look similar, for example each page contains one picture and one text, but all of the pictures or texts are different from the rest. Our list will contain the items of food types and snake pieces, and it will make the items easier to use. Each page in the list will contain a different picture of food and a different picture of a snake piece. The first item in the list identifies the number of pages. In our case, there are five types of food. The second item is a number of the current page in the list — a list pointer (the numbering starts with page 0). The third item specifies the number of pages that the pointer automatically moves with each access to the data in the list.

We have to fill in the food list during the start of the program. It means that we have to specify, what item will be on which page. The commands for filling in the list are shown on the following page. Put them into the beginning of the main program function.

Before we start to fill in the food list, we will specify the initial page, from which we will start filling in the list. We will set the **food index** element to **0**. We will use automatic increasing of page number, and so we will set the **food index increment** to **1**.

First, we will fill in the **food** item on all pages of the list. Gradually fill the **food** elements with individual food type items. The items are stored in the list, and after each item is set, the page number automatically increases by one. After the last item is set, the page pointer automatically moves to the beginning. We will start to fill in the **food — snake** elements. The page number increases automatically again. When we finish, we set the automatic increasing of page numbers to **0**, as we won't use automatic increasing in the program anymore.



If the player fails, that is if the snake hits the wall or eats poison, the game begins anew. For this, we will need to redraw the starting sheet of the program, without food and poison. We could clean it programmatically, but there is an easier way. We will save the main sheet into a variable of the sheet type, and simply restore the sheet in a new game by using this variable.

Create a new sheet variable (copy the **sheet** element from the **Library of Variables and Functions** window to the **Global Variables and Functions** window), and call it **saved sheet of the game**. Put a command for saving the main sheet to the beginning of the program:

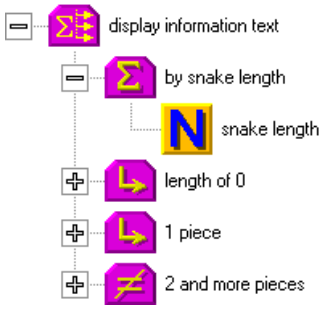
The score will be displayed during the game. Create a numeric variable called **maximum score**, and set it to zero.

Create a new function called **new game**. The function will run at start, and so we will add it to the beginning of the program after the group for filling in the food list. It will be assembled accordingly to the following picture.

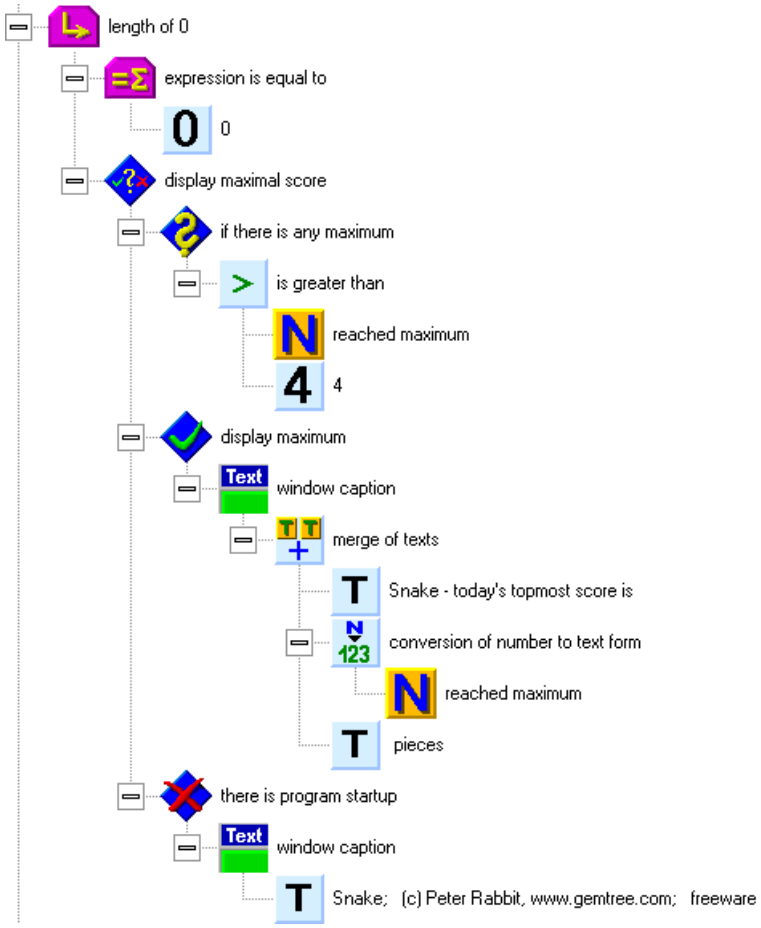
Create a logical variable called **snake movement flag**, a numeric variable called **snake length**, a numeric variable called **next step counter** and a function called **display the number of snake pieces**. The **snake movement flag** logical variable will be used to indicate the beginning of the snake's movement (when the game is started, the snake stays in its place, waiting for a key to be pressed). Set the variable in the **new game** function to "off". The **snake length** variable indicates the number of snake pieces, and it increases during the game. In the beginning of a new game, it will be set to **0**. The **next step counter** counts **0.1**-second time impulses, before the snake is moved one square (it decreases from **5** to **0**). In the beginning of a new game, the counter will be set to **1**, so that the first step is performed immediately after the first key is pressed.

The following command in the new game recalls the saved sheet. Use the same command as when saving the sheet, but reverse the order of the elements. The following commands set the snake's initial position. Turn off visibility for Peter, set his coordinates to **X=9** and **Y=8**, turn him to the right, and make him visible again. In the end of the **new game** function, put the **display the number of snake pieces** function.

Now we will pay attention to the function that displays the number of snake pieces. Many games need to tell something to the player, e.g. the score. If we don't want to pay too much attention to displaying information, we can quickly and simply write them into the window title. We will inform the player about the length of the snake, and optionally also about the highest score. While displaying the number of pieces, we will stick to grammatical rules, and so we will distinguish between 1 piece and 2...3...4... pieces. Here are the contents of the function:



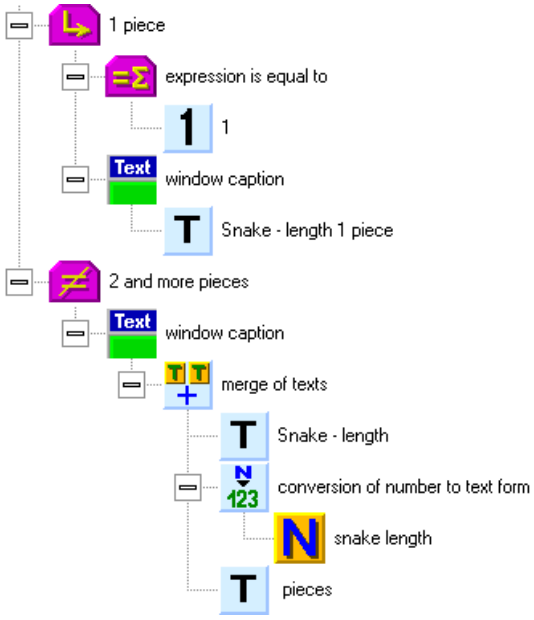
The snake length of **0** will be set in the beginning of each game. In this moment, we can display information about the author of the program, or later, about the maximum score.



The picture contains a new element — **window caption** (from the **controls** group, **dialogs** subgroup). The element is a text variable (it can be set or read) representing the text in the window title bar. The **merge of texts** element serves for connecting more texts into one. The connection is made from the top to the bottom. Normal text can be specified by typing it besides the **text constant** element. The

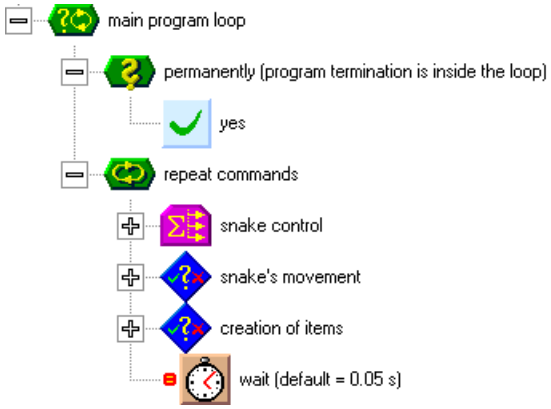
conversion of number to text form element converts numbers to their textual representations.

The branches for other lengths of the snake are assembled in a similar way. Note that we can use one branch of the construction for more values of length.

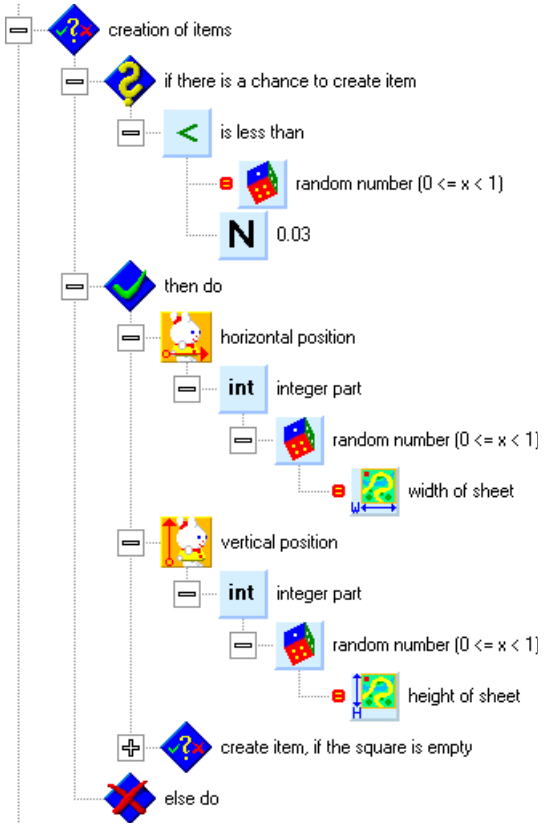


A note on the window title usage: When a program starts, the name of its main function appears in its window title. If you don't need to change the window title during the program run-time, it is enough if you edit the main function name. The main function name in new programs is preset to the program name.

Now we will pay attention to the main program loop. Put the **conditional repeating of commands** element to the end of the main function. Into the condition test, put the **yes** element. This will create a never-ending cycle. The program closing will be handled inside the procedure for keys as a reaction to the **Esc** key. In the beginning of the main loop, we will create the procedure for keys, then we will handle the snake's movement, and finally we will handle the creation of new items (food and poison). The last command in the loop will be a waiting procedure, which will take care of the program timing. You can insert the wait command into the loop now, and gradually, we will put in the remaining procedures.



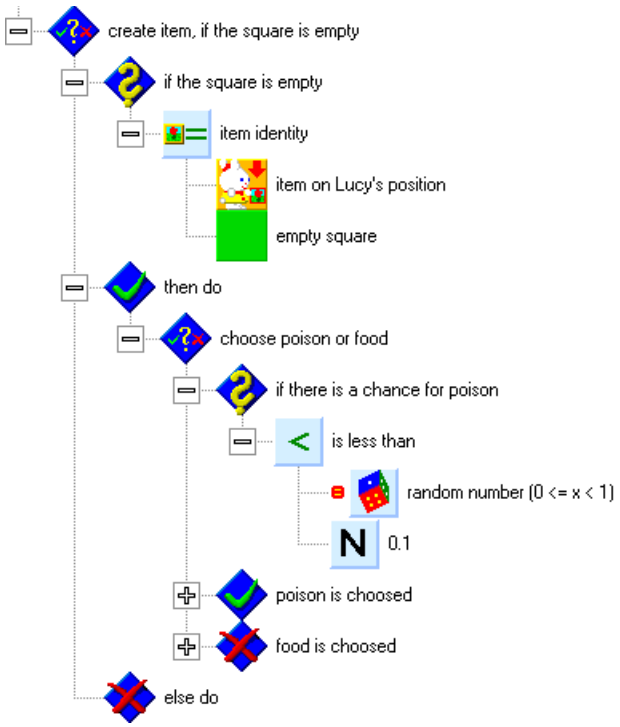
We will start with the creation of new items. In front of the waiting, insert the **conditional executing of commands** element, and call it **creation of items**. Its contents are on the following picture.



New things will be generated randomly, with a certain probability. For this reason, we will put a comparison of a random number with a constant into the condition test. By setting the constant to **0.03**, we will ensure that a new item will be created during **3** out of **100** passes through the main loop. The probability of a creation of a new item is **3%**.

We will create items on the sheet by using Lucy again. First, we will put her onto a random square on the sheet. We will use the **width of sheet** element for her horizontal position. This element will pass the value of **20**, which is the width of the program sheet. It is possible to specify the number directly, but it is better to work with general numbers. This will save us from troubles if we change something in the program in the future. We will pass the width of sheet to the **random number** function, which will create a random number between 0 and 19.999999999. The following function, **integer part**, will delete the decimal part of the number. This will create a random integer between 0 and 19. These numbers represent the horizontal coordinates of the first and last square on the sheet. You surely remember that the squares are numbered from the bottom left corner to the top and right, starting with zero. Similarly, we will create a random vertical position.

After we set a random position, we will use a conditional command, which will test if the square is empty. If it is, the program will use another conditional command to decide between creating food or poison. We will compare a random number with the value of **0.1**, which will ensure that poison will be created in **1** of **10** cases. In other cases, food will be created. All of this is on the following picture:



The following picture shows how a new poison item will be created. We will create different types of poison with different probability. Most often, we want the toadstool, and we want the box of poison in least cases. We will use a **multibranch control structure** . The branching expression will be a random integer between **0** and **5**, created by using the **integer part** and **random number** elements, and the number **6** .

The first branch will create a skull in 2 out of 6 cases (for the values of **0** and **1**), which means that there is a 33% chance that a skull will be created. The second branch will give a smaller chance to the creation of the poison box — 1 of 6 cases (for the value of **2**), so the chance is 17%. In the remaining 3 cases out of 6, a toadstool will be created (for the values of **3**, **4** and **5**), which means the probability is 50% here.

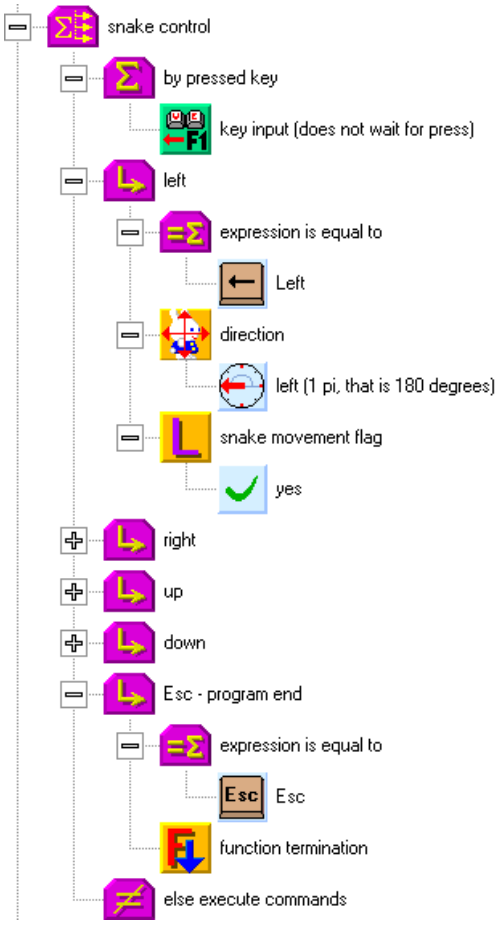
The creation of the poison will be finished by turning **Flag 2** on. As you remember, we had used **Flag 1** to indicate that there is a wall on the square. Flag 2 will be used to indicate that there is a poison on the square. This will make our work easier later, as we won't have to test all of the types of poison used in the program.

Food will be created by a different method. As you can see on the next picture, it is quite simple, because we can use the list of food types.

In the beginning, we will choose a random page in the list, and we will take the food item from it. We will set the food index to a random value between **0** and **4**. These are the indices of the first and last page. Instead of specifying the range of random values directly, we will use an element representing the number of pages in the list: [N]. This will enable us to add food types in the future without having to change the method for their creation. We will keep the number of the food in the **numeric value 1 (range 0 to 1023)** element for later use. The following command will lay the food item on the position of Lucy. The last command will turn **Flag 3** on. Flag 3 will indicate that there is a food on the square.

That is all for the creation of new items — you can test it now (be sure not to omit the waiting in the end of the main loop). A snakehead opening its mouth should appear in the middle of the sheet, and new items should be appearing around it. You can close the program by clicking the ☒ button on the right side of the window title bar.

We will continue by handling the snake control. In the beginning of the main loop (in front of the procedures for creating items), put a **multibranch control structure** element and rename it to **snake control**. The method is on the following picture.

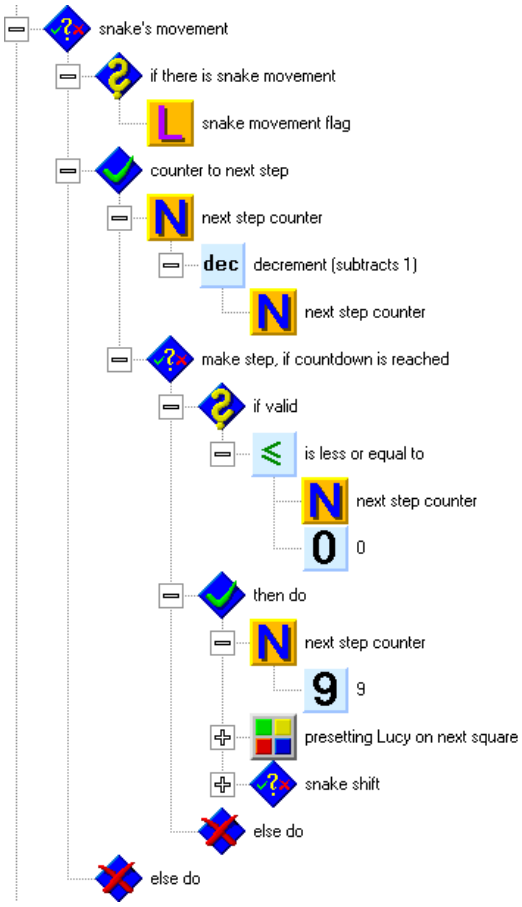


This means that as the branching value, we will use the **key input (does not wait for press)** function. The **Left** key will turn Peter (i.e. the snakehead) to the left and turn on the snake movement flag. The other directions will be similar. In the end, we will handle the **Esc** key — it will use the **function termination** command (from the **program control** group) to quit the function. As it is the main program function, it will quit the whole program.

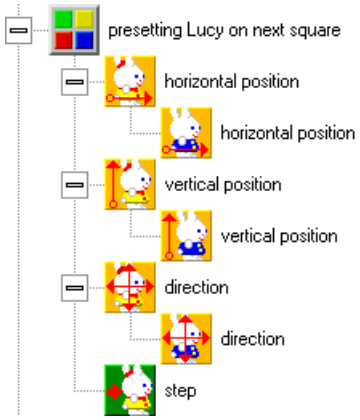
Test the program. You can use the cursor keys (arrows) to turn the snakehead, and the **Esc** key to quit the program.

Our next task is to put the snake into movement. We will choose approximately **2** steps per second as the snake speed. For the sake of handling keyboard input and items generation, the main program loop works a little faster (**18** passes per second). For this reason, we will use a counter for the next step, which will move the snake during every ninth pass.

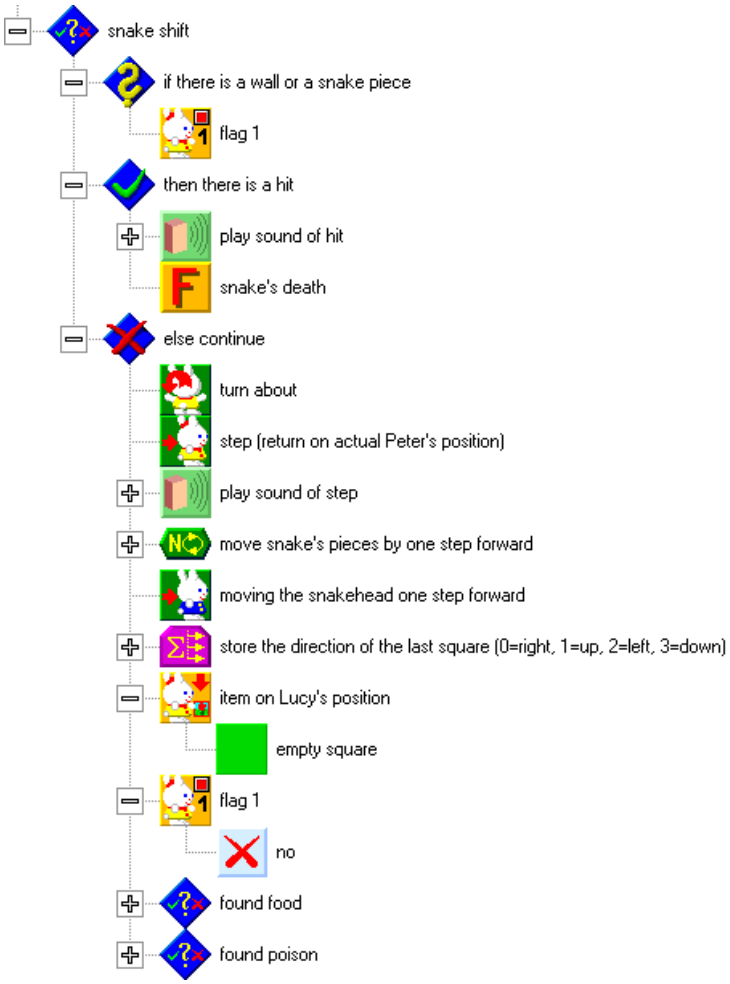
The structure for snake movement is on the following picture. Put it behind the structure of snake control, but before the generation of new items. In the beginning of the structure, you can see a test, which checks if the snake is moving (this is because in the beginning of the game, the snake stays in one place until the first direction key is pressed). When the snake moves, the counter for the next step decrements (decreases by one). When the counter reaches zero, the snake can move one step. First, we will set the next step counter again. We have chosen the number **9**, which represents **2** steps per second (the basic time unit of the computer is **55** milliseconds, which equals to **18** units per second).



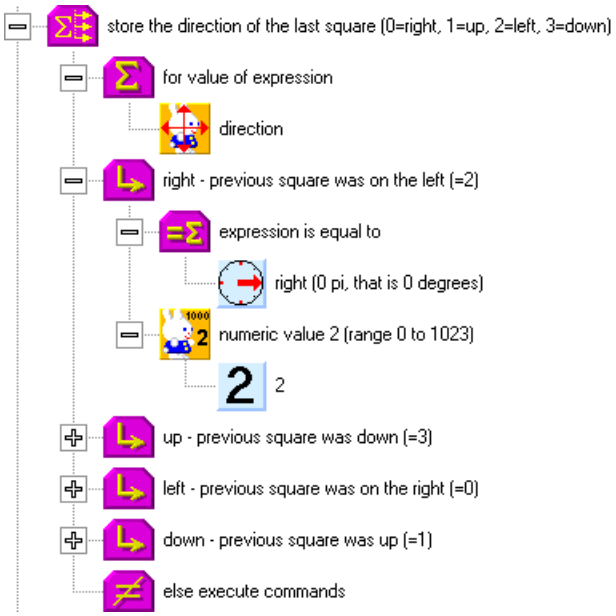
Before the snake moves, we will test whether there is an obstacle in front of it, which it cannot pass. Such an obstacle is a wall, but also the snake itself (that is, one of its pieces). We will use Lucy for this test. We will put her onto Peter's position, turn her in his direction, and make a step forward.



You surely remember that we have indicated walls by **Flag 1**  . For the sake of simplicity, we will assign this flag also to the squares with the snake pieces. If the flag is turned on, it means that there is a wall or a snake piece in front of the snake, and the snake passes away. We will play a sound of a hit and animate the snake's death. We will pay attention to this later. In other cases, the snake can move forward.



In the snake movement, we will begin with Peter — the snakehead. In approximately the middle of the structure of the movement, you see Peter making a step forward. The following branching structure stores a number, representing the direction of the last square, on the new square. Later, this number will help us find the way to the end of the snake.

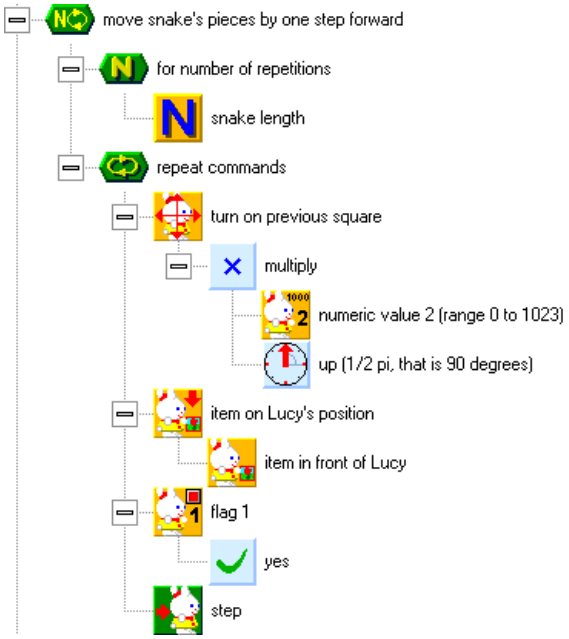


We will keep the number representing the direction to the previous square as **numeric value 2**  . When the previous square is to the right, we will use the number **0**, when it is up, we will use number **1**, number **2** will represent the left direction, and when the square is to the bottom, we will use the number **3**. The structure for the previous square on the left is shown on the picture, the remaining directions are similar.

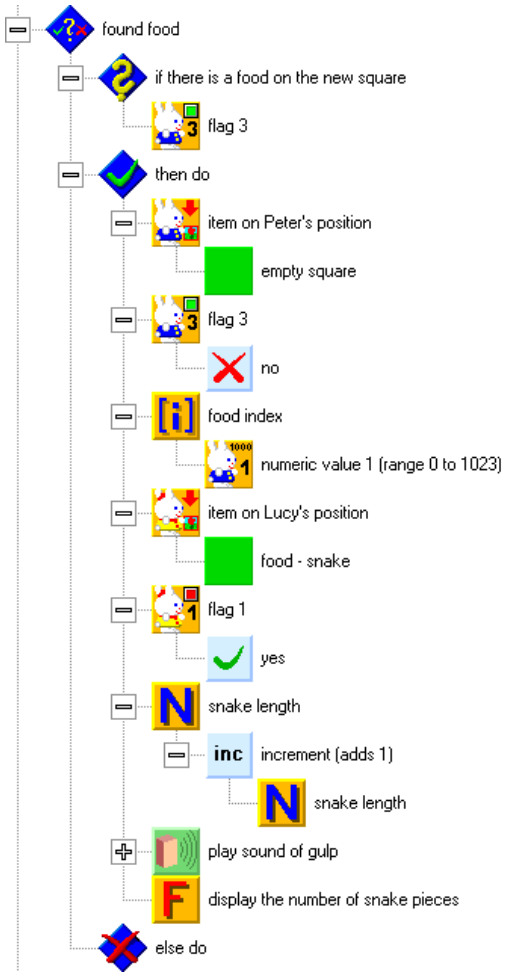
Let us return to the beginning of the structure for moving the snake one step forward. We have left Lucy on the next square, where she tested the state of flag 1. Now we will turn her back and move her one step, which will get her to the same position as the snakehead. We can add a sound for the snake's step.

What follows is a cycle that moves the snake's pieces, as you see on the following picture. The number of the passes of the cycle is given by the snake's length. In the beginning, Lucy is on the position of the head. Each square with the head or a piece of the snake keeps a number representing the direction to the previous square. The number is a number of quadrants (i.e. quarters of a turnaround) that Lucy must turn counter-clockwise from the zero angle (zero angle equals the direction to the right).

First, Lucy turns to the previous square. Using the following command, she carries the item from the previous square to the square she stands on. This way, she moves a snake piece one step forward. The next command turns on **Flag 1**  . As we know, this flag indicates a wall or a snake piece. The last command makes Lucy go one step forward, i.e. to the following square towards the end of the snake.



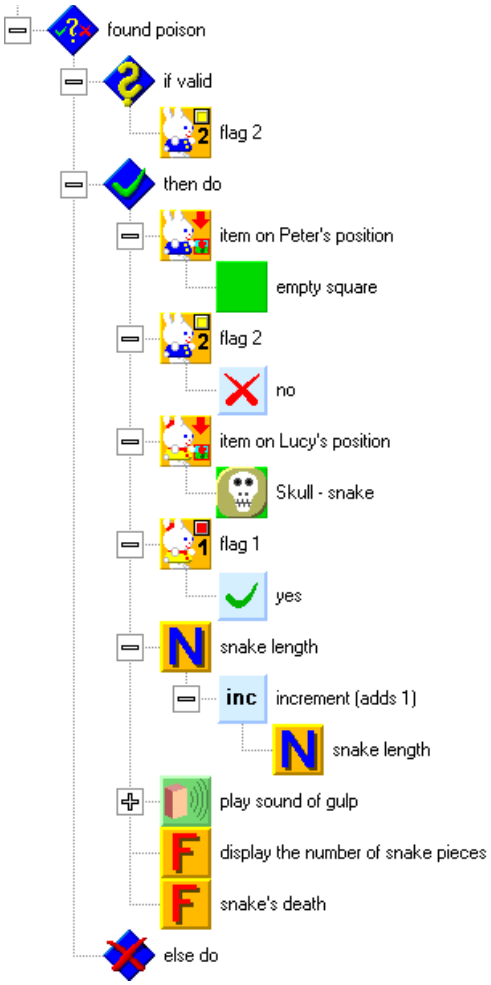
After all of the passes through the cycle, Lucy remains on the square behind the new end of the snake. This is valid even if the snake length is **0**, when no pass through the cycle is performed. What follows is moving the snakehead one step forward and saving the direction to the previous field, which has already been described. We will replace the contents of the square behind the snake with an empty square (which will be performed by Lucy), and turn off the flag indicating that there is a piece of the snake on the square. If a food or a poison is found, we will add a new snake piece later.



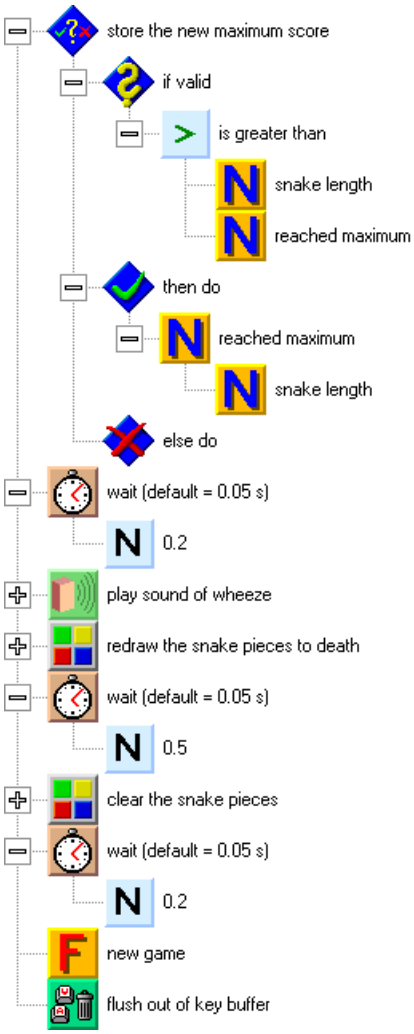
The snake is moved. Now we will pay attention to the contents of the square on the new position of the snakehead. First, we will test, if there is food on it. Food is indicated by flag 3, so we will test **Flag 3**  . The structure for food is on the previous picture.

If there is food on the square, we will delete it by laying down an empty square, and turn off **Flag 3**. The numeric variable 1 of the square stores the number of the food from the food list. We will take **numeric value 1** of the square and use it as the food index to get the appropriate snake piece for the type of food. That means that we will take the appropriate snake piece from the list and lay it down on Lucy's position. As we know, Lucy was left at the square behind the end of the snake, where we had put an empty square for a while. This creates a new snake piece with a food inside. We will turn on **Flag 1**  to indicate a snake piece, and increase the length of the snake. Finally, we can play a gulping sound and show the new snake length.

The structure for poison is similar, as you see on the following picture. Poison is indicated by **Flag 2**. If we find it on, we will clear the square and turn this flag off. We will perform similar steps to the structure for food. We will always use the same new piece, without respect to the type of poison. We will also turn on flag 1 on Lucy's position, increase the snake length, play a gulping sound, and display the new length. Only this time, this will be finished by the death of the snake.



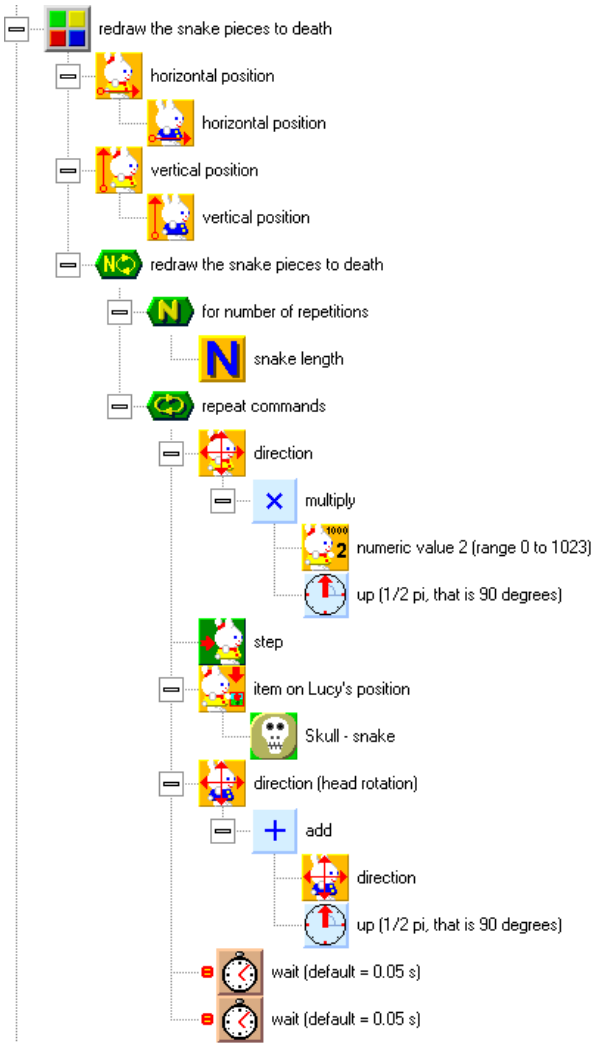
Our last task is to create the function for the snake's death. Its contents are on the next picture.

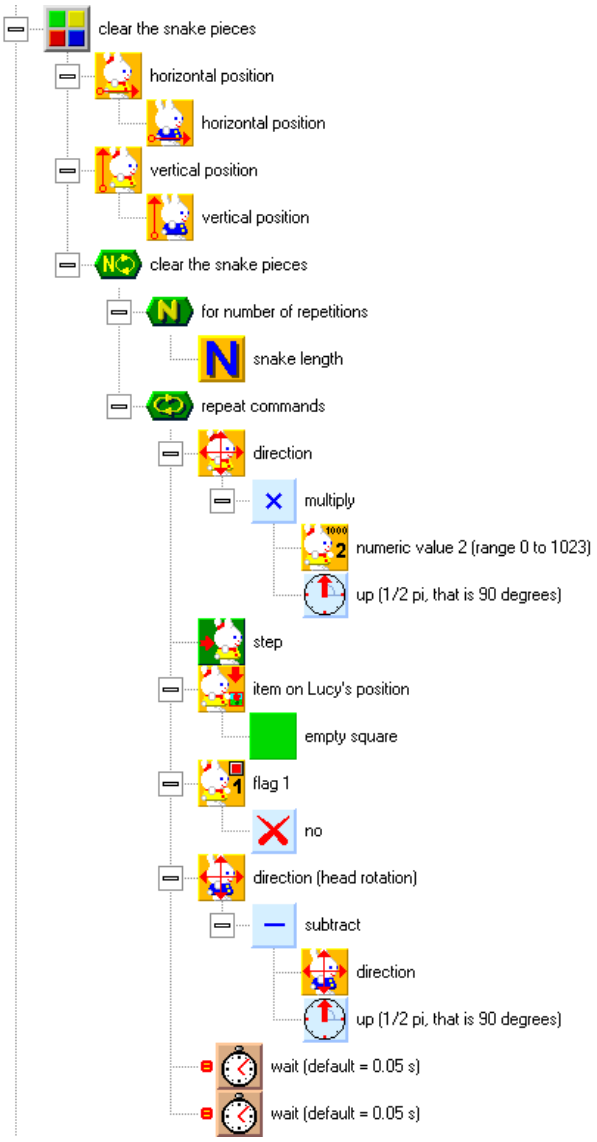


First, the length of the snake is stored as the new maximum score. After a short pause, there is a wheezy sound. The death will be indicated by redrawing all of the snake pieces to pieces with a skull. After the redrawing and a short pause, we will clear the pieces of the snake (this will indicate the process of dying). There is a short pause again, and then a new game starts.

After the beginning of a new game, it is important to add a command for flushing the key buffer out. Players who get absorbed in the game often still press keys when the game is being ended. This could cause an accidental start of a new game.

The following pictures show the redrawing of the snake pieces to death and the clearing of the pieces. Again, we use the directions to the previous squares stored in numeric value 2 of the squares. During that, we animate the snakehead to turn around (the snake is dizzy). In the first case, the snake turns clockwise; in the other, it turns counter-clockwise.





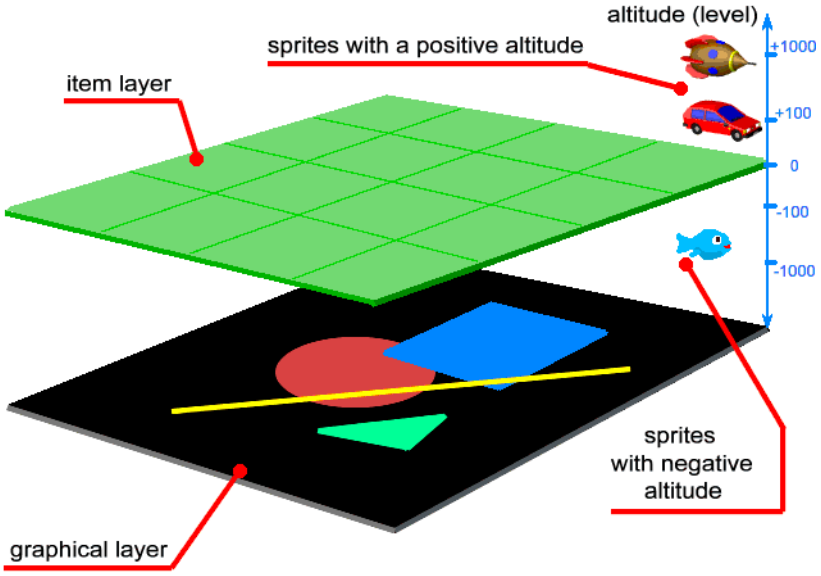
Now you can start the program and test it. Have fun with your first real game.

13 Beginning with Graphics

Our programs have only used items and sprites so far. Now we will get into a more interesting area — graphics.

Graphics bring unlimited capabilities, where item animation is not enough. We can draw graphical elements, such as dots, lines, and circles; what is most important, yet, is that we can also render pictures. We can use pictures to display photographs, move objects, write texts, or draw buttons for special controls.

The first things we need to know is where and how are graphics rendered. Look at the following picture.



The picture shows the display layers of a program window in Peter. The basis is at the **item layer**. We have already used it in previous chapters. It is the layer where we put items, and it is called the **main sheet** of a program. Above this layer, there are sprites with a **positive** altitude (height level). These include Peter and Lucy (unless we change their altitude to a negative number). Under the item layer, there are sprites with a **negative** altitude. At the very bottom, there is a **graphical layer**. All graphical elements are rendered here.

You may imagine the layers as a sea. The item layer is the surface. Ships (sprites with an altitude of 0) move on the surface. Birds (sprites with a positive altitude, e.g. 100) fly above it. Fish (sprites with a negative altitude, e.g. -100) swim under it. The seabed in the very bottom is the graphical layer that may be used for drawing.

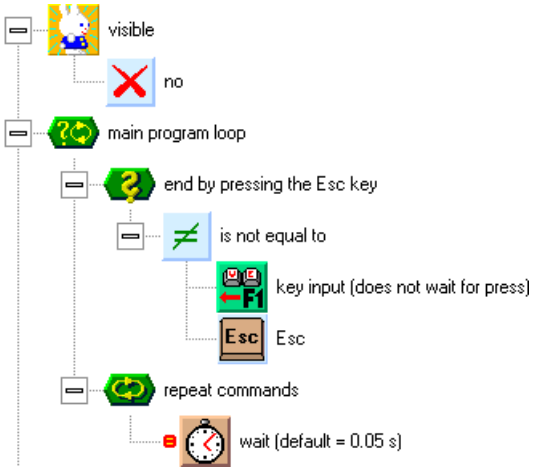
When we look at a program window, it is as if we have watched the sea from above. We see items on the sheet and sprites moving above it (Peter, Lucy), but don't see sprites with a negative altitude, and neither do we see the graphical layer. To see these, we have to make items transparent. We will learn how in the following chapter.

14 Mishmash Drawing

If you don't like the word "mishmash" in the chapter title, you can replace it with anything else, but it is probably the best designation for what we are going to do.

First, we will try the basic graphical commands for drawing. Create a new program called **Graphic** or open the sample program.

We won't need Peter in our program, and so we will turn his visibility off by the **visible** command with the **no** parameter in the beginning of our program. The program will be based on a **conditional repeating of commands** cycle that can be ended by pressing the **Esc** key. Put this command behind the one that turns Peter's visibility off, and add a wait command into it.



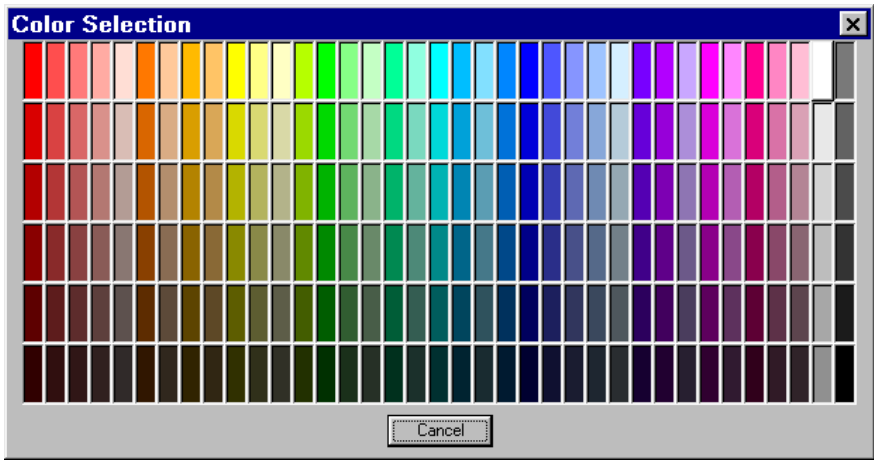
Another very important task is making the squares in the item layer transparent. Edit the **empty square** item. In the color picker, choose the top left color (purple). It is a **transparent color**, which will enable us to see through the square with an item. Choose **filled box** as the drawing tool, and redraw the whole item with the transparent color.

If you will run the program now, you won't see anything but a black sheet. You see the graphic layer, which will be the basis for our next programs. Later, we will use both items and graphics together. After you try the first graphical command, you can draw something into the empty square item. You will see the graphic and a net of empty square pictures over it.

In the **Basic Elements** group, find the **graphic** group, **drawing** subgroup. It contains commands for drawing graphical elements. Drag the **point** command into the main program loop. It contains another four elements that specify the parameters of the point rendered.

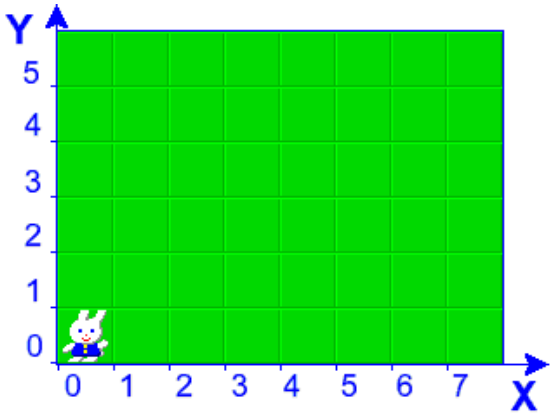
The first parameter, **pen color**, specifies the color of the point. Color is a number, which can be kept in a numeric variable. We are not interested in its value so far, as we can use special functions to specify color. In a new command, the **pen color** element contains a **color** element. This is a **color constant**, which passes the

selected color to the command. You can select a new color by double-clicking the color constant element. The color selection window appears. The selected color is indicated by the elevation of its field. Clicking selects a different color. The color will also appear in the color constant element.



In our program, we will render points with a random color. For this reason, we will discard the color constant element, and use the **compound color components to color** element (from the **graphic** group) instead of it. This component contains another three elements, **red component (0 to 1)**, **green component (0 to 1)** and **blue component (0 to 1)**. The components specify the level of individual colors in the resulting color. The closer is the value to 1, the brighter the component is. For example, yellow has the values of 1/1/0. A random color can be created by inserting **random number (0 <= x < 1)** components into the color components (the range for a random number is 0 to 1 as well).

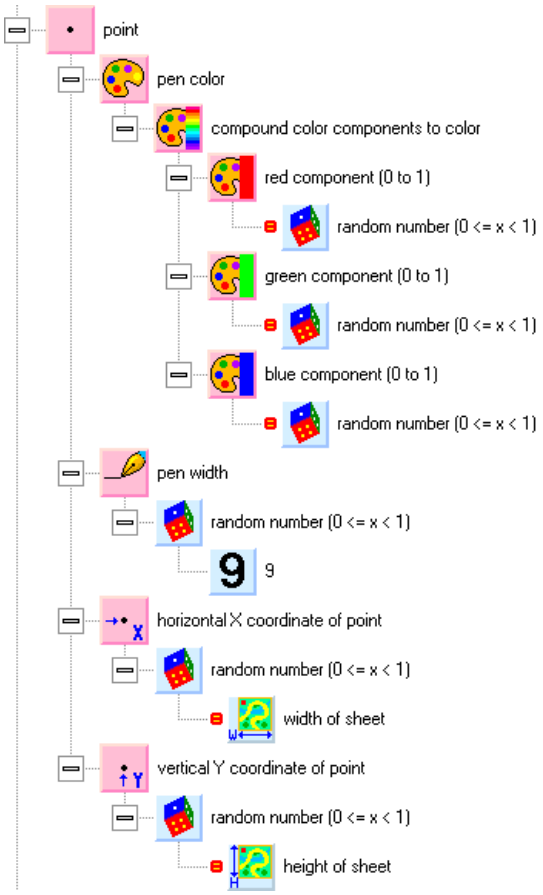
Another parameter for the point command is the **pen width** element. It is a numeric value, which specifies how many graphical points wide will the point be. The point size will be generated randomly, and so we will insert a random number with the parameter of **9** here, which will create random points with the size of 1 to 9 graphical points (we don't have to care about zero, the point command modifies it to 1).



The last two elements of the command, **horizontal X coordinate of point** and **vertical Y coordinate of point**, specify the position, where the point will appear. The previous picture shows how graphical coordinates are numbered. It is the same as the numbering of squares, as we know it from previous chapters. The basic unit of Peter's coordinates is a **unitary step**. The length of the step is the same as a square

width. The beginning of the coordinates is in the bottom left corner. As the coordinates are expressed by decimal numbers, we can specify the coordinates for squares, graphics and sprites in the same way, without any conversions.

The coordinates for the point will be specified randomly, as we know it from previous chapters, by using random numbers with the parameters of the width and height of the sheet. The resulting command for the random point will look like this:



Run the program. Color points of different sizes start appearing on the sheet. The rendering is not very quick. After each point in the loop, the program waits a little moment (55 ms), which means that the speed of rendering is 18 points per second.

This is a good time for discussing the program timing. As we already said, the **wait** command has, besides the waiting function, also a function for cooperation with other programs (and with the core of Windows). An advanced user can use the **System Monitor** program to verify that the **Graphic** program is a minimum load for the computer, and that the program speed is independent of the computer speed. To be accurate — the wait command does not represent an actual pause of a given length; it represents synchronization with the inner clock of the computer. For this reason, the program speed does not depend on the speed at which the commands between two wait commands are executed.

Try to fill a value of **0** into the wait command. The program will now produce very many points. It does not wait for the interval now; it runs at maximum speed. Yet, it ensures that the window sheet is rendered on the display at each pass through the loop. This is another function of the wait command — **it ensures rendering**. The program does not know when all graphical operations are finished, and so when it is a good time to

render the window on the display. If the program rendered the window itself, the background of a game could appear, but not the characters in it. As a result, the background would show through the characters. For this reason, rendering is performed during the waiting, when all graphical operations are probably finished.

A tip for program optimization: The program redraws a rectangular part of the window, where changes were made. If you want to increase the program speed, render only the changed area of the graphic. On the other hand, this could lead to rendering many small parts of the graphic, which could be more demanding than rendering the whole window at a time.

Timing with the value of **0** is useful in situations, when you need higher program performance, or when the timing after 0.055 seconds is too rough. It ensures fluent redrawing of the window, maximum program speed, and sufficient cooperation with Windows. The program speed is dependent on the computer speed now. We could use **System Monitor** to see that the program consumes most of the computer performance, even if it seems to be in a relatively quiet state.

You can also try to disable timing completely by using the **Turn Off** button. If you run the program now, you see a noise of colors instead of points. Now, the program really runs at maximum speed, without any pauses. You may notice that the color noise is not fluent; the image is a bit choppy. It is so because the program ensures at least a minimum redrawing of the window to the display. If there is no wait command for about 0.2 seconds, the program redraws the window itself.

Let us get back to graphical commands. Add other graphical commands to the main program loop — **line**, **box**, **filled box**, **circle**, **filled circle**, **sphere**, **triangle** — and try to render them randomly. Use a random radius of **2** with circle, filled circle and sphere. Skip **Filling**, as it would not have much effect here.

The **text display** command is also interesting. It has more parameters than you may be used to, but don't worry. You don't have to set the parameters that don't interest you. Their default values will be used then. It is probably clear what the **text to be displayed**, **pen color**, **horizontal X coordinate of text** and **vertical Y coordinate of text** parameters mean.

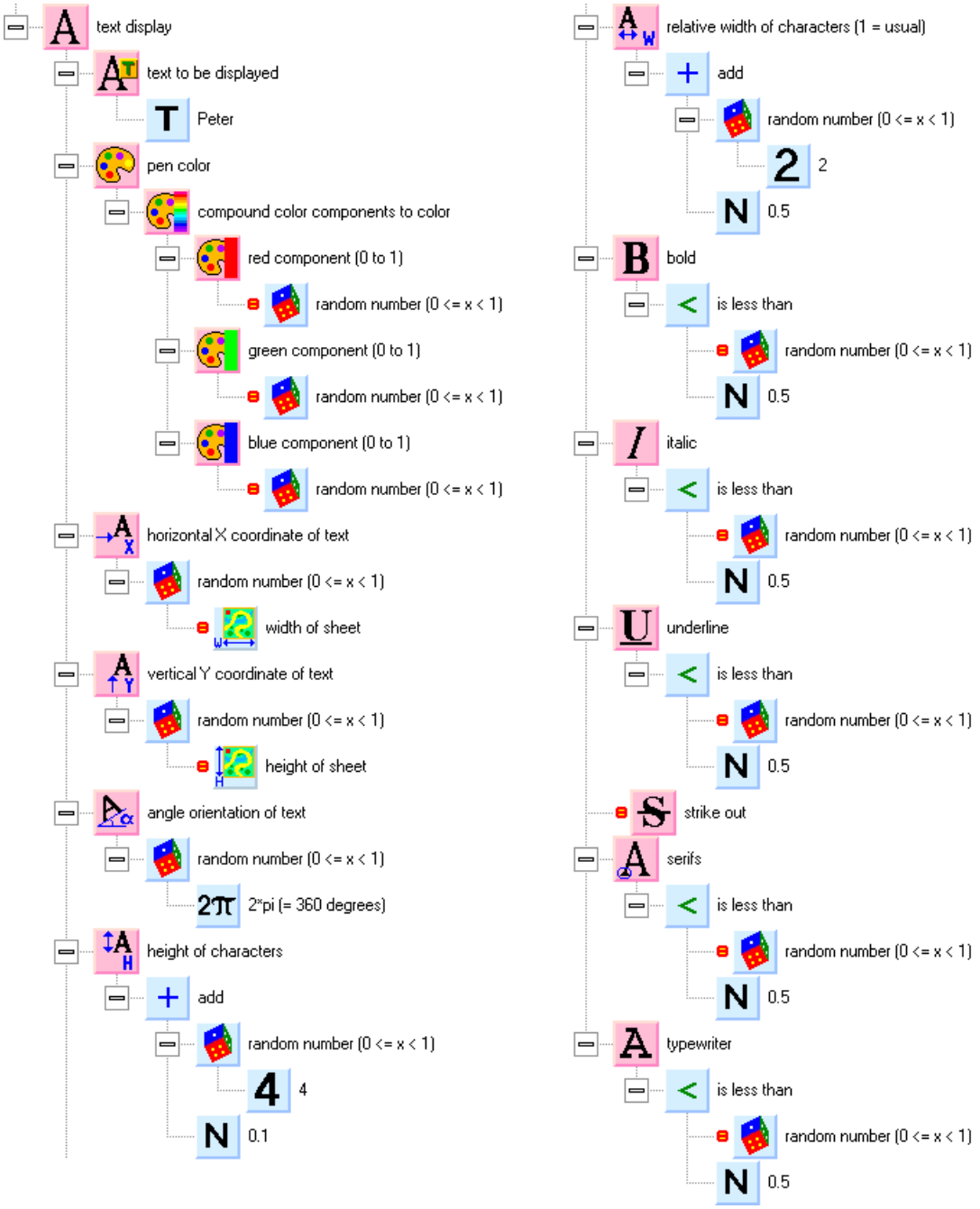
A new element is the **angle orientation of text**. It is the angle, at which the text is turned around the bottom left corner. It is specified in radians, just like other angles in Peter. We can use the direction constants that we know from setting the directions of Peter and Lucy. If not specified, a direction of **0** is used (horizontally from left to right).

The **height of characters** element specifies the height of the characters in the text. It is specified in unitary steps. If not specified, the height of **0.5** will be used (half of the square height).

The **relative width of characters (1 = usual)** element represents a relative number indicating the width of the characters when compared to the normal width. A value of **1** is the normal width. Numbers higher than **1** specify wider characters, smaller numbers specify narrower characters. A special value is the number **0**. This sets the

recommended width for the type of characters used. It is similar to the normal width, but may differ slightly. If not specified, the default value of **0** is used (the recommended width).

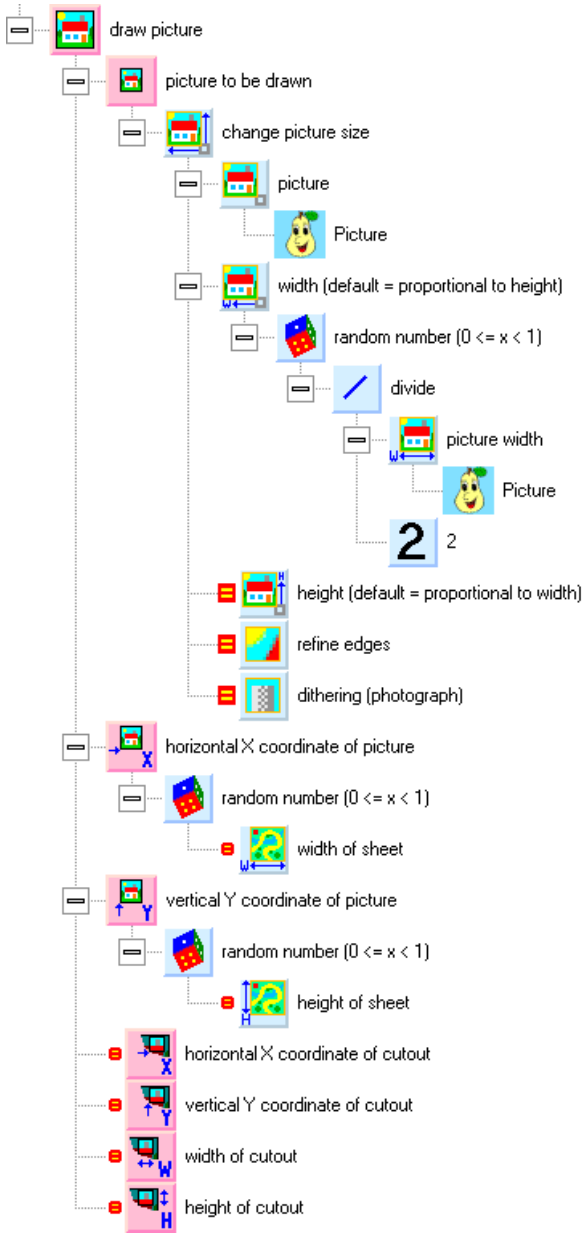
The **bold** , **italic** , **underline** and **strike out** elements are logical switches that turn on special effects for the text. By default, all of these are turned off. The **serifs** switch turns on serif characters. By default, it is also turned off. The **typewriter** switch turns on characters with the same spacing and width. When it is turned off, characters have different widths (**I** is narrower than **M**). Unless specified otherwise, this switch is off as well.



The whole command for a random text is on the previous picture. It is quite long, and so the picture is separated into two parts. Test the program after you create the command.

One of the most powerful aspects of graphics are pictures. You can use pictures for anything that is not predefined in Peter. You can create moving characters, a window with animated buttons, a moving background for a game, or snapshots.

The commands and functions for pictures are in the **graphic** group, **pictures** subgroup. The basic command is **draw picture**. Besides obvious parameters, such as the picture to be rendered and its coordinates, the command also has more parameters that make it possible to draw only a specific part of the picture.



In our test program, we will try to render a picture randomly. Before you start drawing, you can look at the **Library of Variables and Functions**. With Peter, you also get many pictures. Open the **[examples]\Drawing** group. You will surely find a picture that you will like. When browsing, click to select the first picture in the group, and then scroll the selection cursor up and down by using the arrow keys. The editing window will show previews of the pictures.

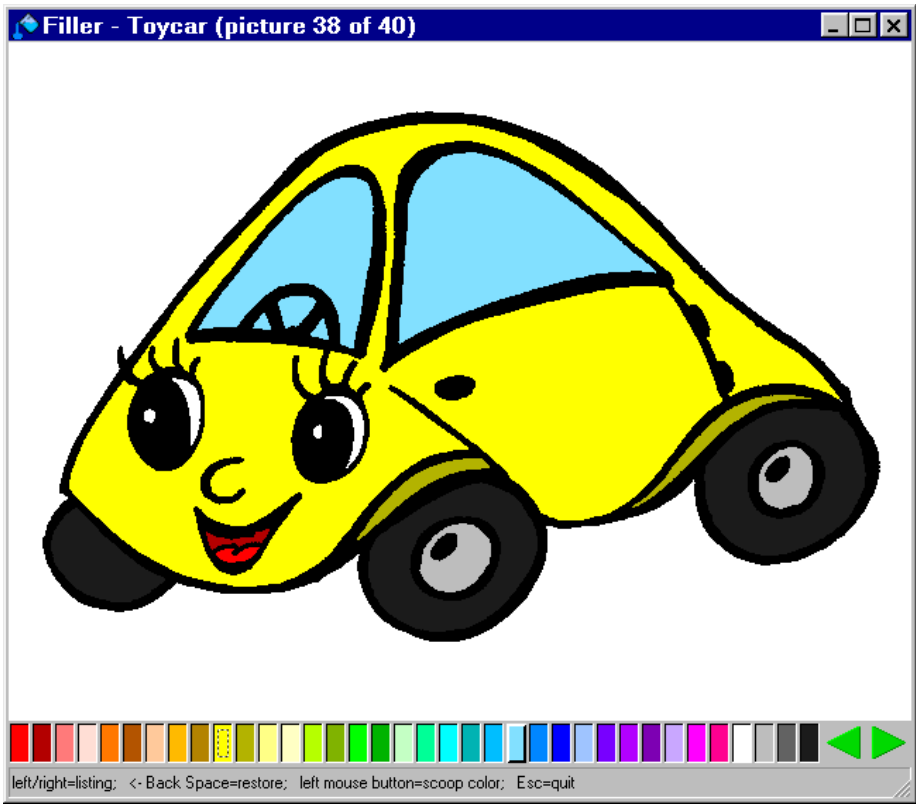
Drag the selected picture into the **Global Variables and Functions** window. If the picture has a one-color background, you can fill the background with the transparent color (by using the **Filler** function) to display the picture without any background.

The command for random drawing of the picture is shown above. If you want to draw a picture with a random size, you can use the **change picture size** function. The new picture width will be random, and will be based on the original size shrunk to approximately one half. It is not necessary to specify a new height; it will be adjusted to the new width automatically.

Note: It is better to use a larger picture and a higher level of reduction. This ensures a higher quality of the resulting picture than using a smaller picture.

15 Fill Your Own Colors

Our next task in the world of Peter the rabbit is to create a graphical editor enabling the user to fill colors into prepared pictures. During that, we will learn how to work with the mouse and with files. The program should work like this: It will find all pictures (BMP files) in its folder. The user can browse through the pictures, and add colors by filling areas. Black will be reserved for the outlines of the pictures.



Let us say a few words about files and folders. A **file** is a separated piece of data stored in the computer. It can be a picture, a letter, a spreadsheet, or a program. User data files are often called **documents**. A **folder** is a "pack of files". Folders are similar to **groups** in Peter. If we want to express the location of a file or a folder on the disk, we use a notation called **path**. A path is a list of folders (sometimes also with the disk) through which we have to go to get to the file (folder). They are separated by backslashes "\". For example: **C:\Program Files\Peter\Peter.exe**. Notice that the disk is labeled by a letter and a colon. At the end of the path, you can see a program name. This notation is called **full file name**. It contains the disk, the folders, the file name, and the extension of the file name. The **file name extension** is the part behind the dot, and it specifies the file type. For example, the **EXE** extension labels programs.

First, create a new program called **Filler**, or use the sample program prepared in Peter.

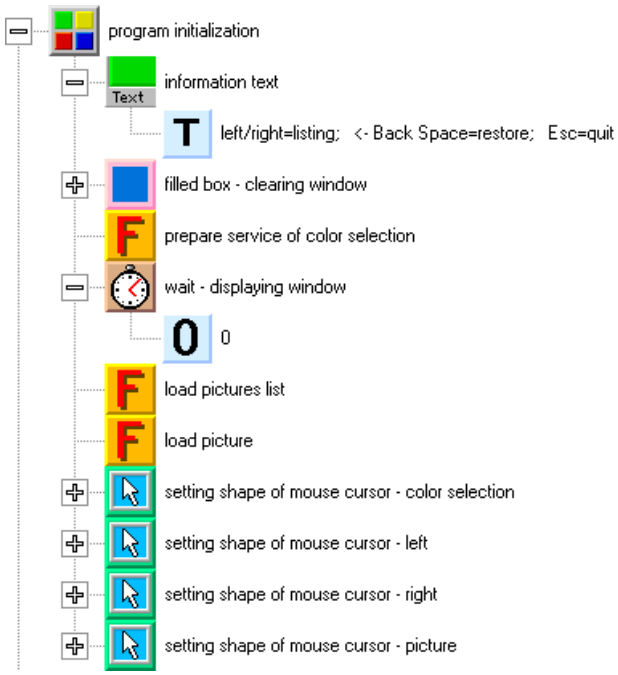
We will adjust the size of the sheet. We will edit pictures with a standard size of 20 x 15 squares (640 x 480 graphical points). To select colors and pictures, we need a bar with the height of one square in the bottom of the window. For this reason, set the size of the program main sheet to 20 x 16 squares. It is still a good size for the screen mode of

800 x 600 points. This screen mode (video mode), but preferably higher, is also the recommended minimum for the users of our programs.

Create two new items — **left arrow** and **right arrow**. Into them, paint the pictures of control buttons for browsing to the left and right. The background will be gray. Lay the items into the bottom right corner of the sheet, as on the previous picture.

Create the main program loop — a never-ending **conditional repeating** (the cycle condition will contain a **yes** element). The cycle body will contain one wait command.

In front of the main program loop, prepare a group called **program initialization**. It will contain the preparatory operations required to run the program.



The first command will set the help text into the status bar in the bottom of the window. If the status bar does not contain any text, it is turned off. If you want to use the status bar, it is recommended to turn it on in the beginning of the program. It does not look well when the program window appears first and the status bar only a few moments later.

Tip: If you want to use an empty status bar, display the *space* character.

The second command will fill the program window with white color. In the **filled box** command, use just one parameter — the white color. If we skip the remaining parameters, we will fill the whole window. This command ensures that black background does not show through during the loading of the list of pictures and before displaying the first picture. It is used for purely aesthetical reason, but in general, the transitory states of programs should not be neglected. Small negative impressions can build the users' attitudes to programs. If something wrong can be seen, the user does not trust the program. The same thing happens when the program does not react to user actions quickly enough, or when the user does not know how to control it. Even

worse effects arise when an intuitive command leads to unexpected and undesired results.

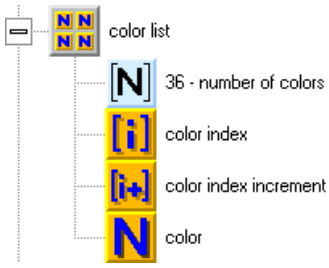
The third command is a **prepare service of color selection** function. It will define the colors for drawing and displays the color fields. For the time being, prepare an empty function.

What follows is a wait command with the parameter of **0**. Why is it here? As you know, the wait command ensures that the window is displayed. When the program starts, the window is not turned on right away. The program waits for the first wait command. When it comes, the program assumes that the window content is already prepared. Without this behavior, transitory effects could be seen during the program startup. The wait command with the parameter of 0 ensures that the window is turned on immediately after startup. Without it, the window would be turned on a little while later, after all files would have been found. This is not a mistake, but if the window does not appear soon enough, the user may doubt whether the program is starting at all and try to run it again.

The **load picture list** function finds all BMP files with pictures. The **load picture** function loads the current picture into the program and displays it. So far, only prepare empty functions. The last four commands redefine the cursor appearance. We will talk about them later.

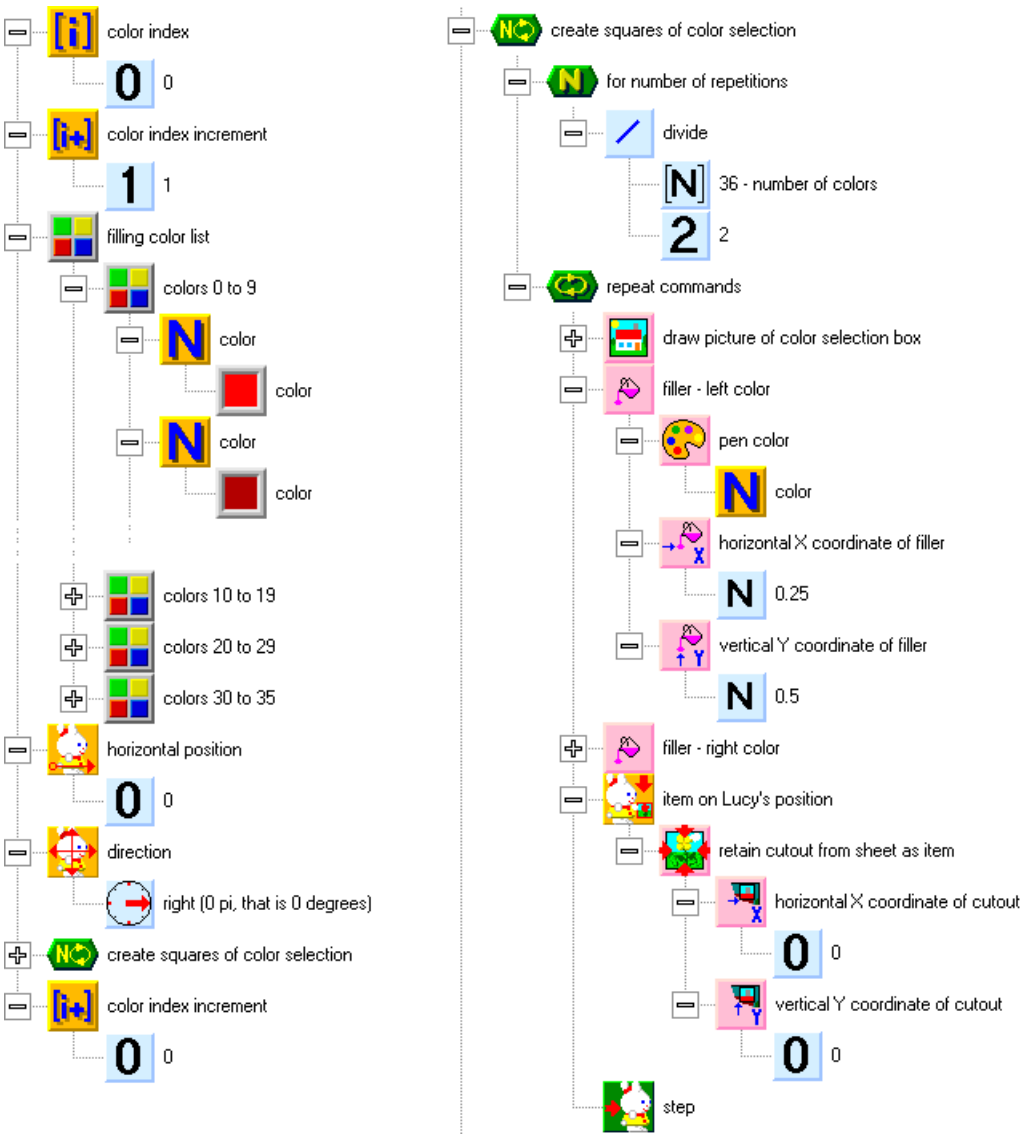
If you run the program now, you see a white sheet, two buttons in the bottom right corner, and texts in the window title and the status bar. It is nothing special, but it is just the beginning.

In the **Global Variables and Functions** window, create a new list called **color list**. Set the size of the list to **36**. This will be the number of colors used in the editor. Rename the pointer in the list to **color index** and the automatic increment to **color index increment**. Add one numeric data element called **color** into the list.



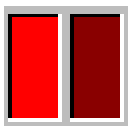
We will create the contents of the **prepare service of color selection** function. Switch into it by double-clicking its icon. The whole function is shown on the following picture. On the right side, there is the cycle that creates the color fields.

First, we will fill the list of colors. Set the color index to **0** and the color index increment to **1**. This ensures that the colors will be stored in the list from the beginning and that the pointer will automatically be increased by 1 when each color is set. The color setting commands follow. Choose **36** colors (not black — that will be used for picture outlines). Why 36? For the choice of color, we will have 18 squares, and each square will contain 2 colors.



The color choice fields will be displayed using items. Items are more suitable than graphical rendering here, because when we would fill areas, the color could also get into the color fields. Instead of drawing items by hand, we will create them programmatically. It is easier than editing 36 fields and it will be easier to keep matches between the fields and the actual colors, should we change the colors in the program later.

In the **Global Variables and Functions** window, create a picture (not an item) called **color selection box** with the size of 1x1 square. Into it, draw fields for the choice of two colors, like this:



To create items, we will render the picture into the program window, fill it with colors, cut it out of the window as an item, and lay it on the sheet. We will use Lucy to lay the items on the sheet. We will set her to the first field (with the horizontal coordinate of 0) and turn her to the right. A cycle for all color items (half of the number of colors) follows.

The first command in the cycle will draw the picture of the color selection box. It will have just one parameter — the picture to be drawn. If the rest of the parameters are skipped, the picture will be drawn in the bottom left corner of the window.

The second command will fill the left field with a color from the list. The coordinate will refer to the center of the field. In a similar way, we will fill the second field with the second color. Remember that the color index moves automatically.

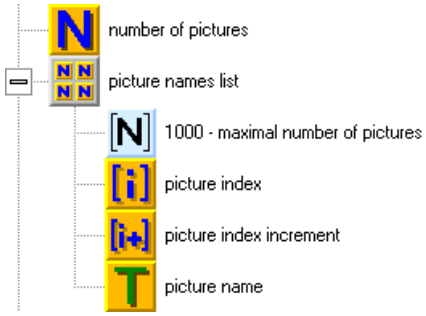
Cut the created picture from the window by using the **retain cutout from sheet as item** function, and lay it onto Lucy's position. Lucy will then go to the next square.

When all color fields are created, change the color index increment to **0**. From now on, we won't use automatic increments in the program.

Do you think that we perform too many operations in the window and that they must be visible? These preparatory drawing operations are actually very fast and we have time enough before the contents of the window will be rendered on the screen with the first wait command. The picture we drew will remain hidden under the first color selection item, and so we don't even have to clear it.

You can run the program now. If everything is OK, a bar with the color selection fields will appear in the bottom of the window.

In the **Global Variables and Functions** window, prepare a list that will keep the names of all files that will be found (like on the following picture). Label it **picture names list**.



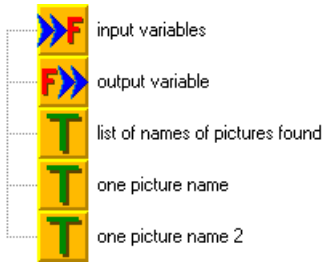
Set the list size to 1000. It may be a relatively huge redundancy, but it is not very important from the point of view of memory consumption. Generally, each variable in a list takes up 4 bytes; only numeric variables take up 8 bytes and logical variables 1 byte. Our list will use 4000 bytes. We can count on millions of bytes of free memory.

Call the list pointer **picture index** and rename the automatic pointer increment to **picture index increment**. Add a new textual variable called **picture name** into the list.

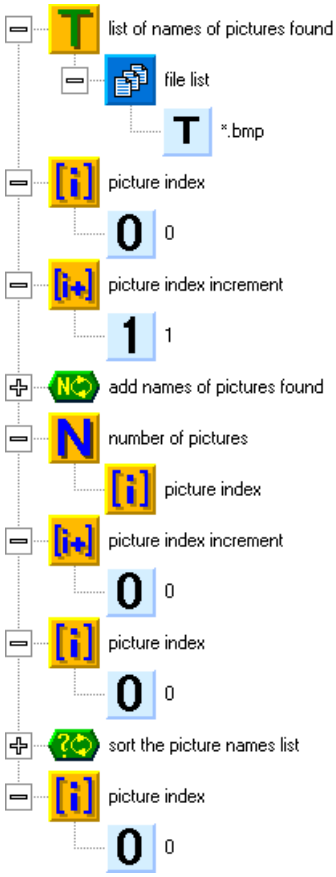
Besides the picture list, we will also create a numeric variable called **number of pictures**. We will probably not fill the whole list, and this variable will keep the actual number of the pictures in the list.

Now we will prepare the **load picture list** function. If you opened the **Filler** sample program, you can see a function that is a bit more difficult in it. The sample program supports multi-user environment, and so it loads files also from the program's home folder. We will use a simple variant, which will only load picture files from the current folder.

We will prepare the local variables of the function. The names of the files that will be found will be stored in a text variable called **list of names of pictures found**. We will sort the list alphabetically, and for this purpose, we will prepare another two textual variables — **one picture name** and **one picture name 2**.



The next picture shows the function contents. The function will load the names of picture files from the current folder, add the names to a list, and sort the list alphabetically.



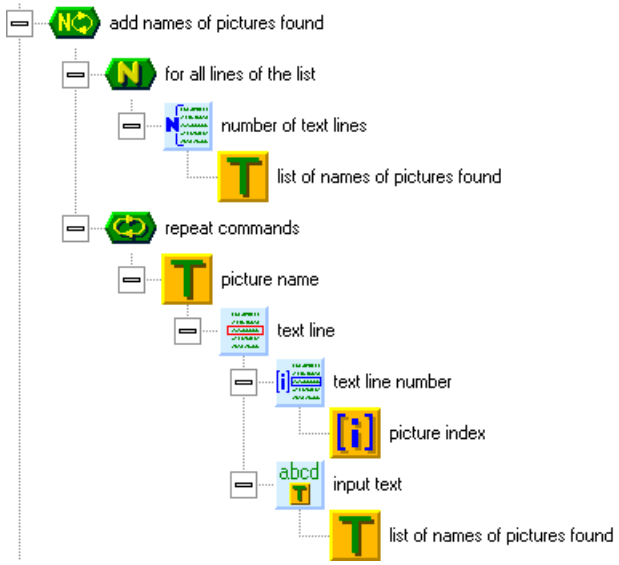
To find files in the folder, we will use the **file list** function. As the function parameter, we will use a textual specification of the files to find. The specification uses wildcard characters (a question mark "**?**" represents any character, an asterisk "*****" represents any group of characters). You can specify more entries. The individual entries will be separated by a semicolon "**;**". To find e.g. all BMP picture files, we will

type ***.bmp**; to find all BMP and JPG (another format supported by Peter) files, we will specify these two entries: ***.bmp;*.jpg**.

The **file list** function returns a multi-line text, in which each line contains the name of one file found, including its extension (but not its path). When searching for BMP files, we could get for example this text:

Bull.bmp
Car.bmp
House.bmp
Kite.bmp
Ram.bmp

We will add the names of the individual files into the picture names list. Initially, we will set the **picture index** to **0** and the **picture index increment** to **1** (we will use automatic increments of the index of the list). As the line number, we can use the picture index. The index will be automatically increased after the name is saved.



After saving all file names into the list, we will use the final state of the picture index to set the **number of pictures** variable, and we will set the picture index increment to zero.

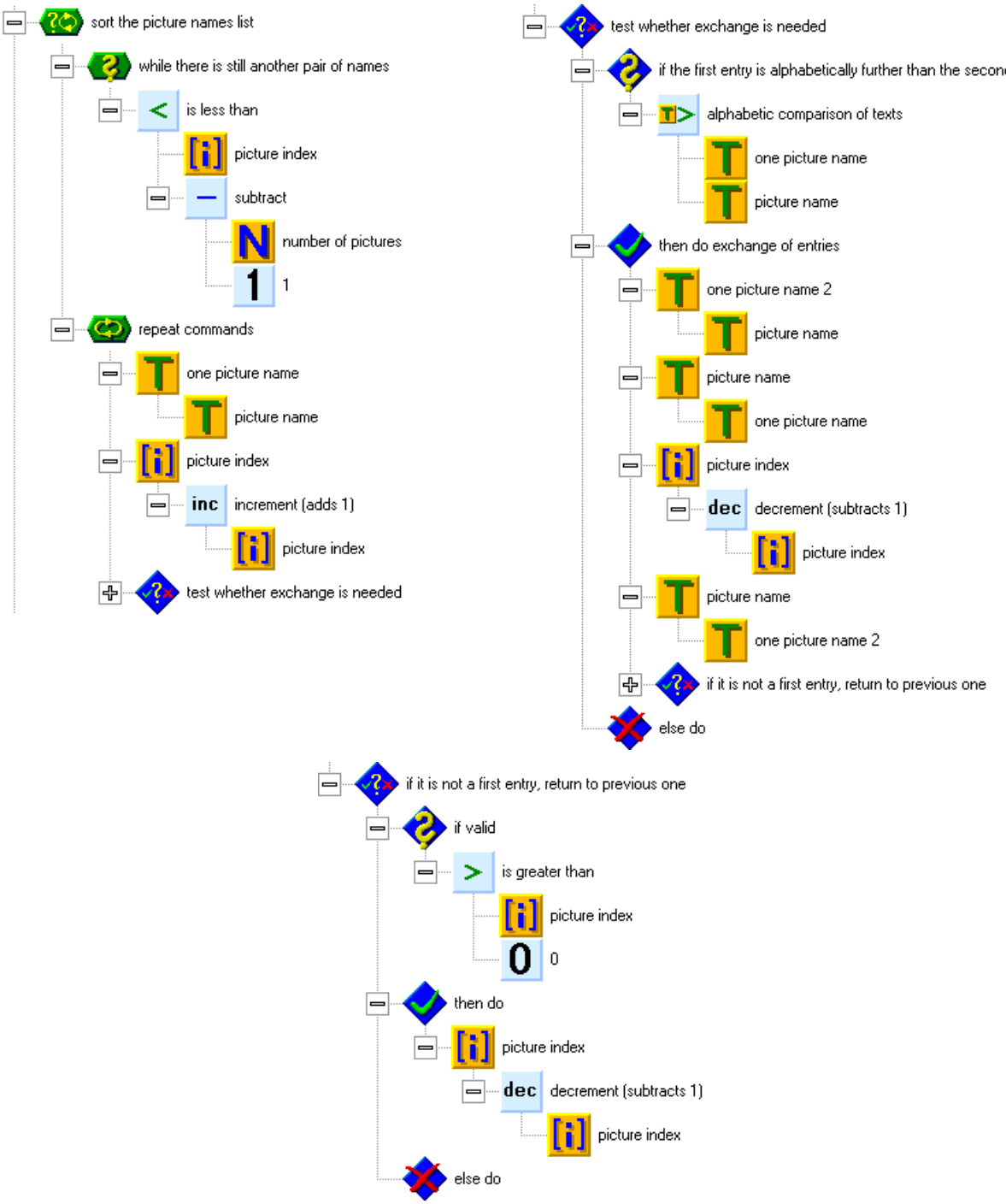
After this, we will sort the list of picture names alphabetically. We will go from the beginning of the list to the end, and each time, we will compare two neighboring names. If we find an unsorted pair, we will replace the positions of the names and go back to the previous pair to ensure that the alphabetically first name will be moved as far to the beginning of the list as appropriate.

The sorting will be performed in a conditional cycle. In the cycle condition, we will test if there is still another pair of names. In the beginning of the cycle, we will store the first name into a textual variable called **one picture name**. We will increase the pointer to the following name in the list, and test if the first name is alphabetically further (higher) than the second name. If it is, we will switch the names' positions.

When replacing the names, we will store the second name (to which the pointer is set now) into the **one picture name 2** variable. We will save the first name into the second name's position. We will decrease the pointer back to the first name, and save the

name that used to be second into the first name's position. The positions are switched now.

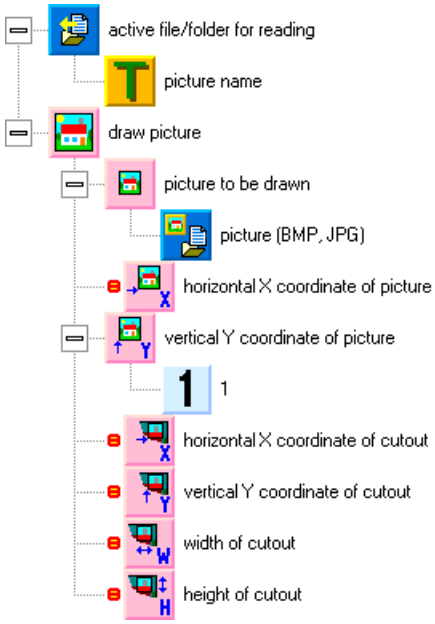
Finally, we will decrease the pointer in the list to be able to move the first name towards the beginning of the list (if it is not in the appropriate position yet).



Look at the sorting method and make sure that you understand how it works. It is not the fastest method, but it is simple and good enough.

The last command in the **load picture list** function sets the picture list pointer to the value of **0**. That will be the default picture to display.

We will create the contents of the **load picture** function. It is quite simple, as you see bellow. We need a couple of elements for working with files from the **files** group. First, we will set the **active file/folder for reading** accordingly to the picture name from the list. Then, we will put the **picture** element (**data** subgroup) in the draw picture command. That's it.

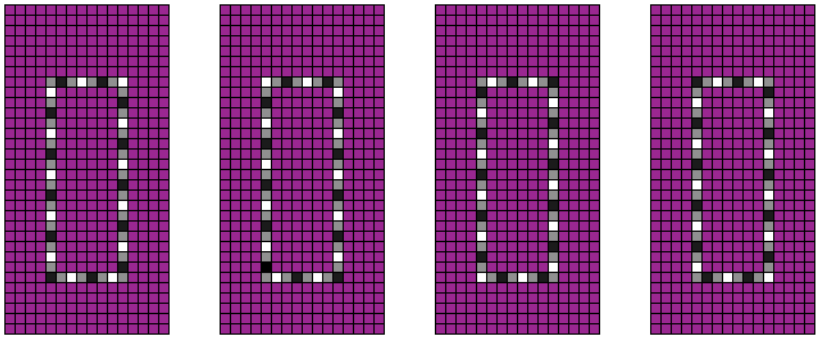


You may wonder what will happen if no picture file is found. In that case, the picture name will contain an empty text, and the functions for loading or saving a picture will not be performed. In the sample program, this is supplemented by displaying a text, which informs the user that no picture was found.

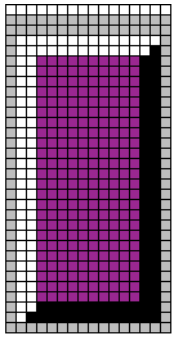
Now, we should prepare a few pictures, or at least one picture. We can use Peter's graphical editor for this. Set the picture size to **20x15** steps (i.e. **640x480** points). Draw the outlines with black color. Make sure that the outlines contain closed-up areas. Otherwise, the color will leak. After the picture is drawn, save it in the **Library of Variables and Functions** and use **Windows Explorer** to move it (typically from the **C:\My Documents\Peter\Picture** folder) into the folder containing the **Filler** program. You can also use Peter's sample pictures in the **[examples]\Drawing** group.

When you have at least one picture in the same folder as the **Filler** program, you can test the program. The alphabetically first picture should appear.

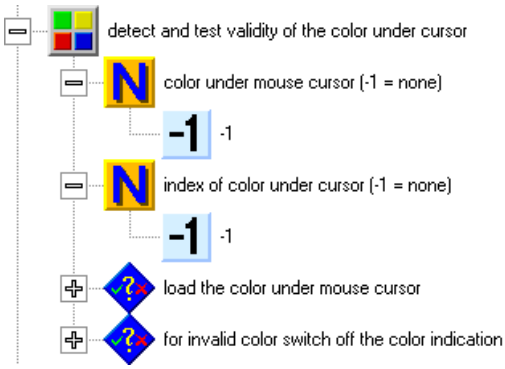
In the program, we will use Peter as the indicator of the selected color. Double-click the **Peter** icon in the **Global Variables and Functions** to edit the sprite. Change the sprite parameters (by clicking the **Properties** button) to the following values: Delay Between Phases = **110**, Phases per Step = **0**, Standstill Phases = **4**, Moving Phases = **0**, Directions = **1**, Picture Width = **0.5**. Delete the pictures of Peter from the sprite and double-click the first picture to edit it. Draw a frame into the picture using a white-gray-black-gray color sequence. Copy the picture to the remaining fields of the sprite. Each time, move the points by 1 point clockwise. When you test the sprite, you will see the frame "flowing" around the border.



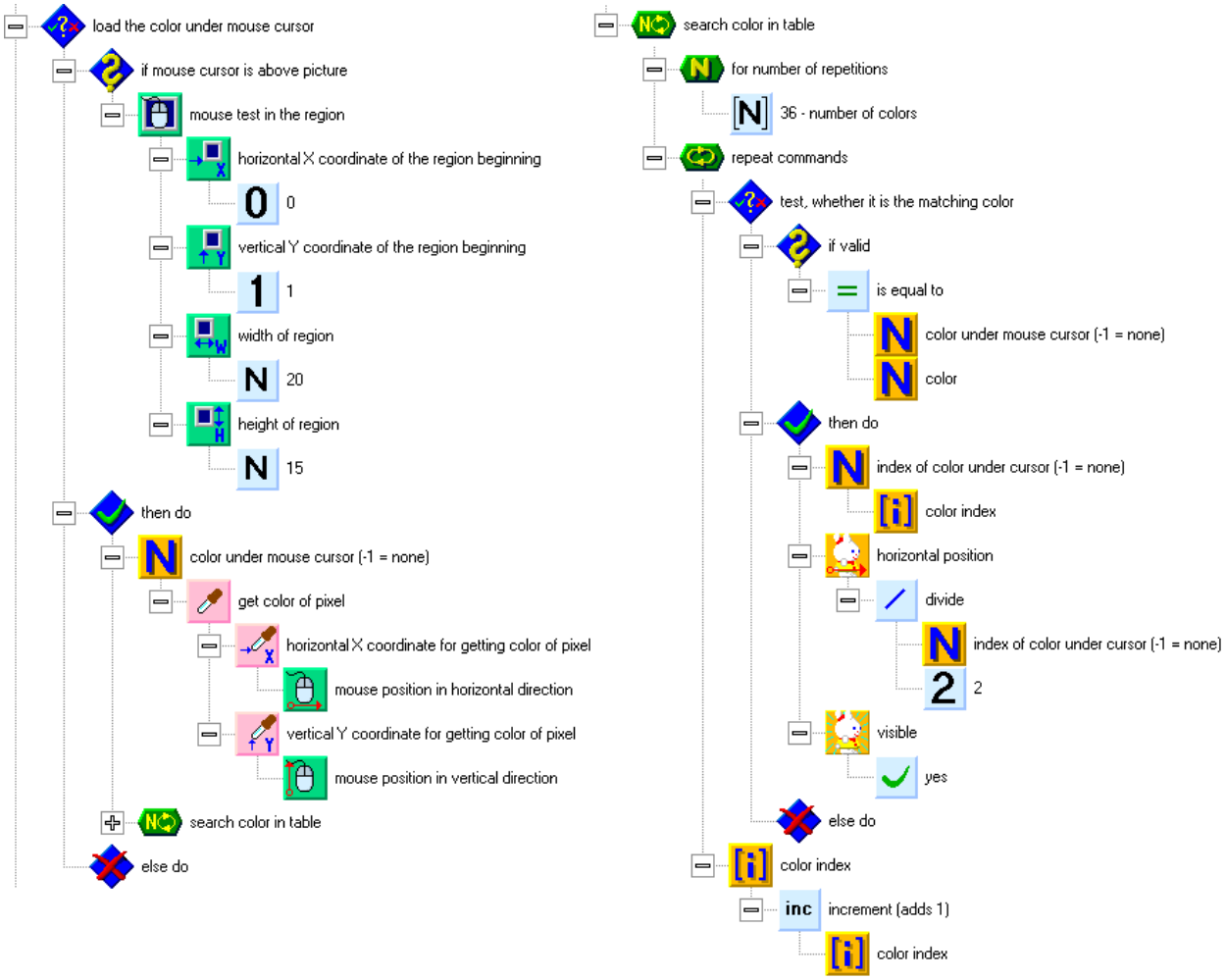
Lucy will be used to indicate the color under the mouse pointer. Start editing the sprite of Lucy and set the following parameters: Phases per Step = **0**, Moving Phases = **0**, Directions = **1**, Picture Width = **0.5**. Redraw the picture in the sprite to create an impression of an elevated color picker field. Leave the central part transparent.



In the main program loop (before the wait command), prepare a group called **detect and test validity of the color under cursor**. Here, we will continually test the color under the mouse pointer, which will enable us to indicate the appropriate color. The detected color will also be used in other structures in the program as well.



In the **Global Variables** window, prepare two numeric variables: **color under mouse cursor** and **index of color under cursor**. In the beginning of the group, set both of these variables to **-1**, to indicate there is no valid color under the pointer. What follows is a conditional command that tests if the mouse pointer is above the picture. The test will begin with a **mouse test in the region** element (**controls** group, **mouse** subgroup), with parameters set accordingly to the picture in the window. If the pointer is above the picture, we will get the color under it by using the **get color of pixel** function (**graphic** group). We will store the color in the **color under mouse cursor** variable. The coordinates of the point from which to get the color will be the same as those of the mouse: **mouse position in horizontal direction** and **mouse position in vertical direction**. In Peter's programs, information about the mouse (and other devices) doesn't change until the next wait command.

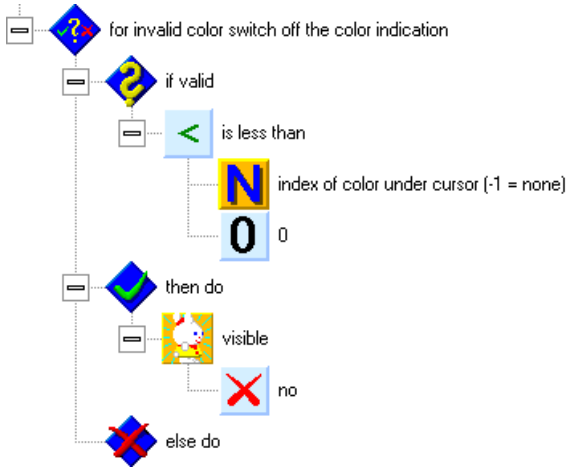


After reading from the point, we will try to find the color in the color table. The cycle for this is on the picture on the right. The cycle goes through the whole color table, comparing each color with the one under the pointer. If it finds the matching color, it stores its index into the **index of color under cursor** variable, sets Lucy (as the indicator of the color under the pointer) over the field of that color, and switches Lucy's visibility on.

Look closely at the cycle we have used. The color index in the color list identifies the color selected by the user. The cycle goes through the whole color list, and after the cycle is finished, the color index will have exactly the same value as before the cycle began. We have used an automatic return of the pointer to the beginning of the list after the end is reached.

Let us look at Lucy. Throughout the program, her vertical coordinate keeps the default value of 0. Her horizontal coordinate is set to the half of the index of the color under the pointer, as the color fields are half a square wide. If you wonder whether Lucy can also stand between squares, then the answer is yes. When we set the coordinates of Lucy, we can use any value, including values outside the window, as Lucy is a normal sprite. Only when being moved by a **step** command, Lucy is limited to the window sheet, and her target coordinate is truncated to the nearest square. The same applies to Peter.

Behind the conditional command that gets the color, there is a conditional command that switches the color indicator off when there is no valid color under the pointer. This can happen if the pointer is not above the picture, or if there is black (or unknown) color under it.



Run and test the program. The indication of the color under the pointer should work now. If you rest the mouse pointer over a color in the picture, the field with the appropriate color should be elevated over the window surface.

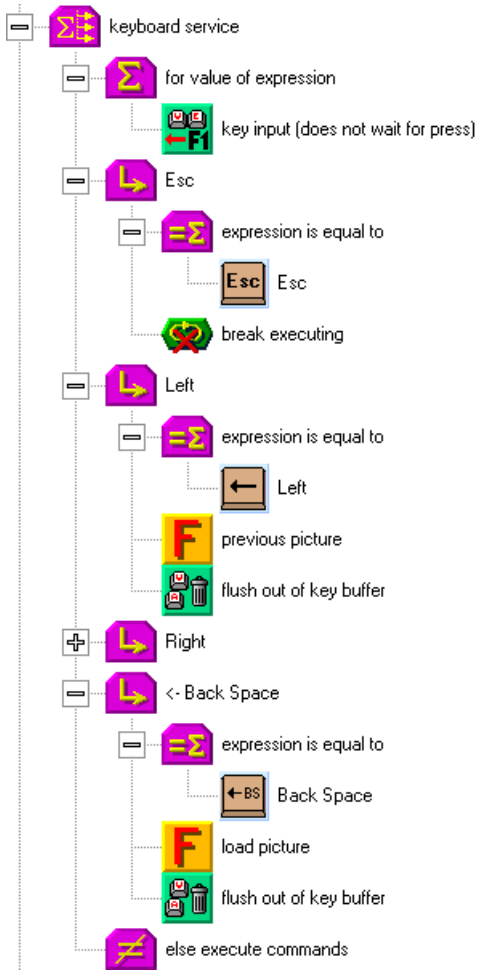
We will start working on the program control. We will begin with keyboard control. After the structure that detects the color, insert a **multibranch control structure** called **keyboard service**. As the tested value, use the **key input (does not wait for press)** function.

The first branch of the structure will handle the **Esc** key. Here, the **break executing** command will be used, which will quit the main program loop.

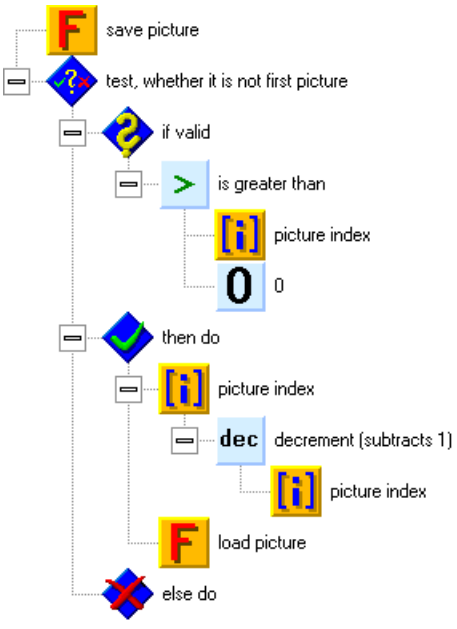
The second branch will take care of the **Left** key. Create an empty function called **previous picture** and insert it in the branch. Behind it, put a **flush out of key buffer** command. The loading of a picture takes a while. If the user holds the key, the key codes are coming too fast for the program to switch pictures. For this reason, the program would move through pictures even after the key would be released. The control would be subject to inertia, which is not pleasant.

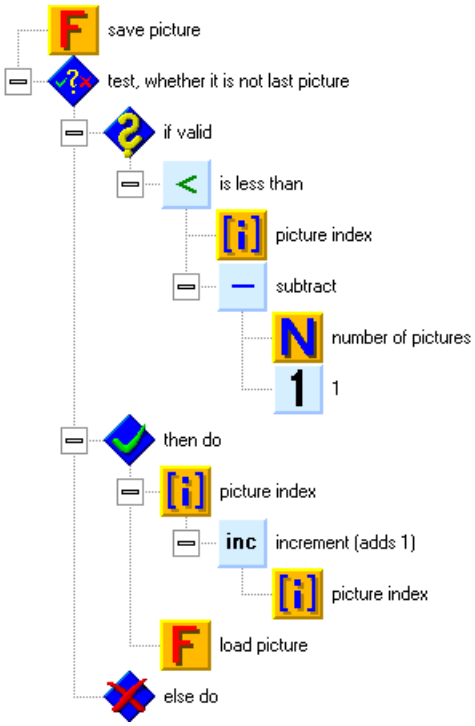
The branch for the **Right** key will be similar, only this time, we will use a **next picture** function. You can also add branches for other direction keys, e.g. **Home** to jump to the first picture, **End** to jump to the last picture, and **Ctrl+Left**/**Right** to move 10 pictures in the desired direction.

The last key will be **Back Space**. The user can use it to reset the picture to the original state (as it was before editing). This will be handled by the **load picture** function, which will read the picture from the file anew. After this, another flush out of key buffer command follows.



The pictures below show the contents of the **previous picture** (on the left) and **next picture** (right) functions. In the beginning of the functions, the current picture is saved (if it has been changed). So far, you may just create an empty **save picture** function.



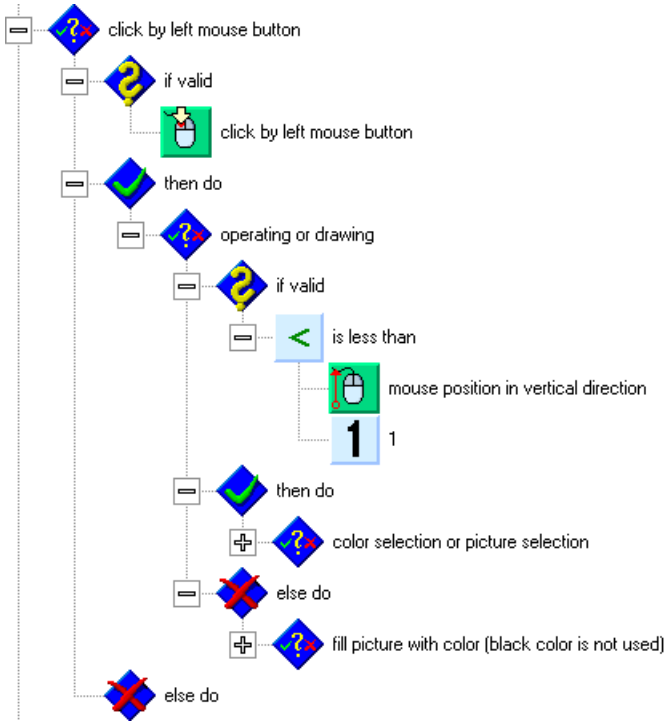


In the **previous picture** function, we will check if we're not working with the first picture. If we're not, we can move the picture pointer to the previous picture, and load and display the new picture. If we work with the first picture, nothing will be performed, but we could set the pointer to the last picture (which is **number of pictures - 1**) and move around the pictures in circles.

In the **next picture** function, we will check if we don't have the last picture in a similar way. The number for the last picture is the number of pictures decreased by 1. If we don't have the last picture, we will increase the pointer to the next picture and load the new picture. In the case of the last picture, we could move to the first picture to ensure cyclic browsing.

Test the program. You can move among the pictures by using the **Left**/**Right** keys, or press **Esc** to quit the program.

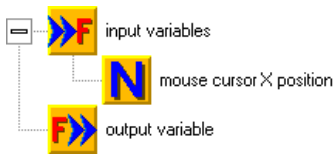
We will not create the function for saving the picture so far. Otherwise, we could overwrite a picture by an accident. First, we will create the structure for mouse control. In the main program loop, insert a conditional command called **click by left mouse button** behind the structure for keyboard service.



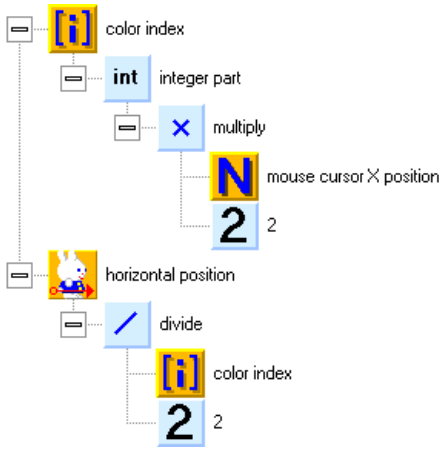
Insert a **click by left mouse button** flag into the condition test. It is a logical flag, and it is set each time that the user left-clicks in the program window. The flag is turned off when it has been read, e.g. by being used in a condition. If you want to test the left-click flag more times, store it into a logic variable.

If the condition test detects that the user has clicked the left mouse button, we will use another conditional command to distinguish (by the vertical coordinate of the mouse) whether the user has clicked into the picture or into the bar with colors and buttons. The border point here is the value of **1**, as the bar is **1** square high.

For the choice of color, we will create a new function called **color selection**. Into the input variables of the function, we will insert a **mouse cursor X position** variable. It will pass the horizontal coordinate of the mouse in the color selection bar to the function.



In the function, we will take the value of the input variable, multiply it by **2** to convert the coordinate to the number of a color, use the **integer part** function to delete the unnecessary decimal part, and use the result to set a new color index. We will set the horizontal coordinate of Peter to the half of the color index value, which corresponds to the position of the color field. Peter is used in the program as the indicator of the selected color. His vertical coordinate remains set to **0** permanently, and so we don't have to pay attention to it.

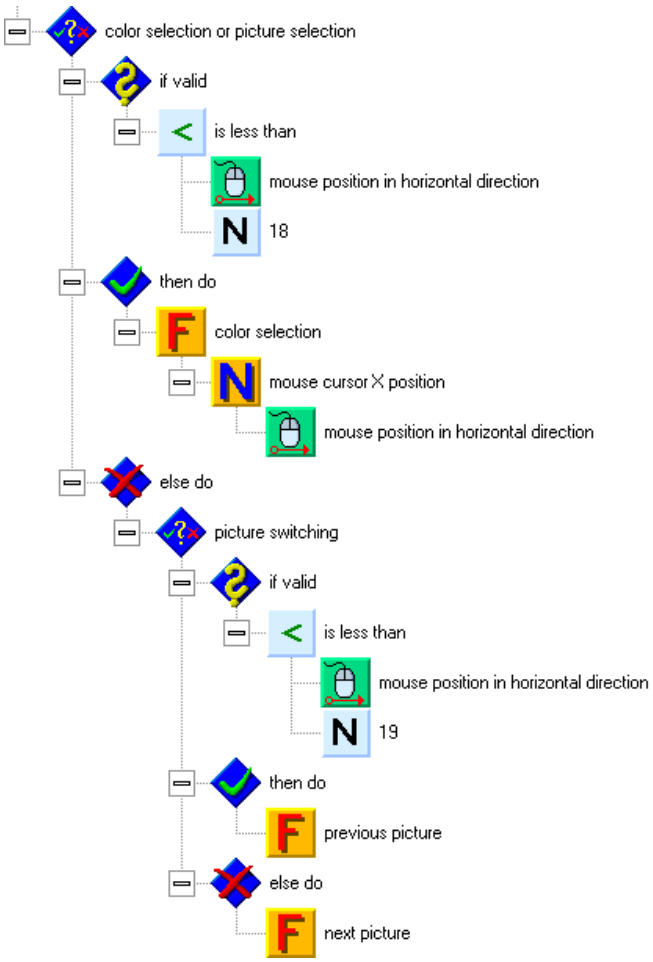


If the vertical coordinate of the mouse tells us that the user has clicked into the bar with buttons and colors, we will use the horizontal coordinate to distinguish, if they have clicked into a color field or on a button. If the horizontal coordinate is less than **18**, then the user selects a color. We will invoke the **color selection** function and use the coordinate as its parameter. In other cases, the user has clicked a button to move among pictures. A horizontal coordinate below **19** indicates the left button, and so we will invoke the **previous picture** function. In the remaining cases, we will invoke the **next picture** function. The whole structure for the color choice fields and direction buttons is on the following picture.

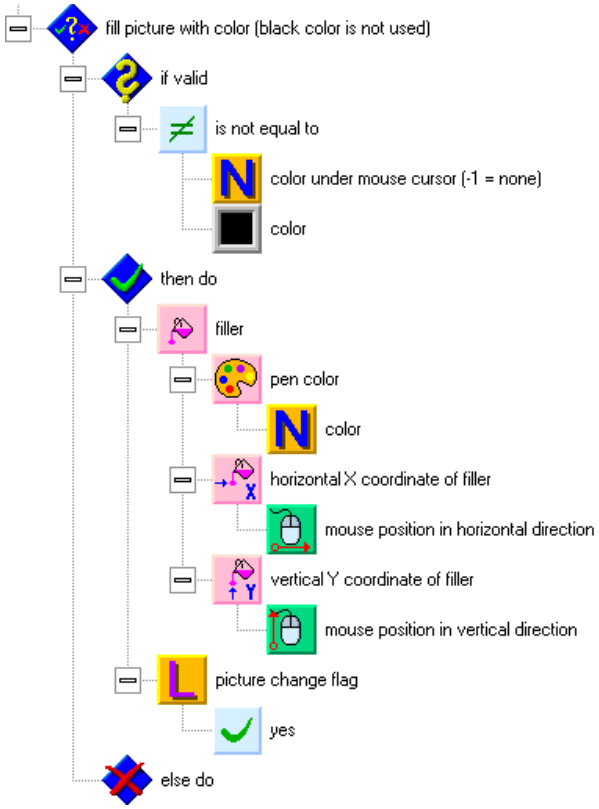
You can test the program. You can use the left mouse button to switch pictures, or to change the selected color in the color selection bar.

Now we will handle the cases when the user left-clicks into the picture. The structure will be based on a **fill picture with color** conditional command (see the bottom of next page). The condition test will check if there is black color under the mouse pointer. Black is used for outlines, and cannot be used for filling. The color under the pointer has been saved to a variable in the beginning of the program main loop.

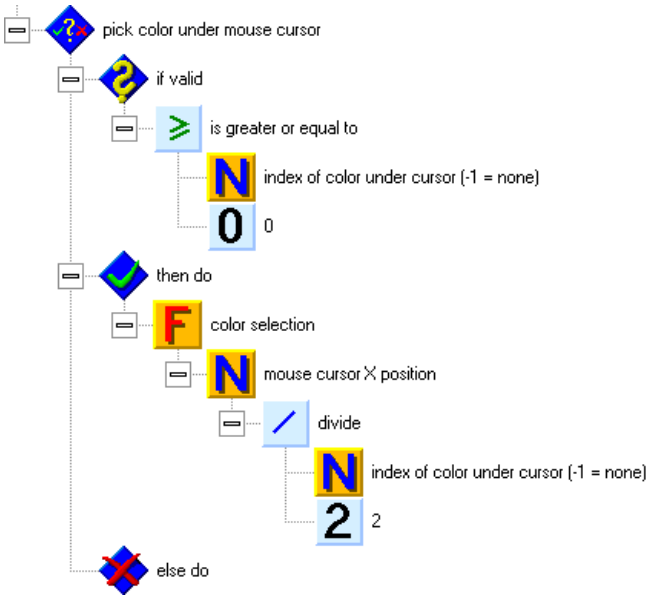
Then we can fill the picture with a **Filler** command. The color is set by the **color** variable from the color list. The coordinates for the filling will be those of the mouse. Finally, we will set a **picture change flag**, which we will prepare in the **Global Variables and Functions** windows. The flag indicates that the picture can be saved.



Run the program, try to fill the picture with a selected color, and verify that you cannot fill the picture with black color.



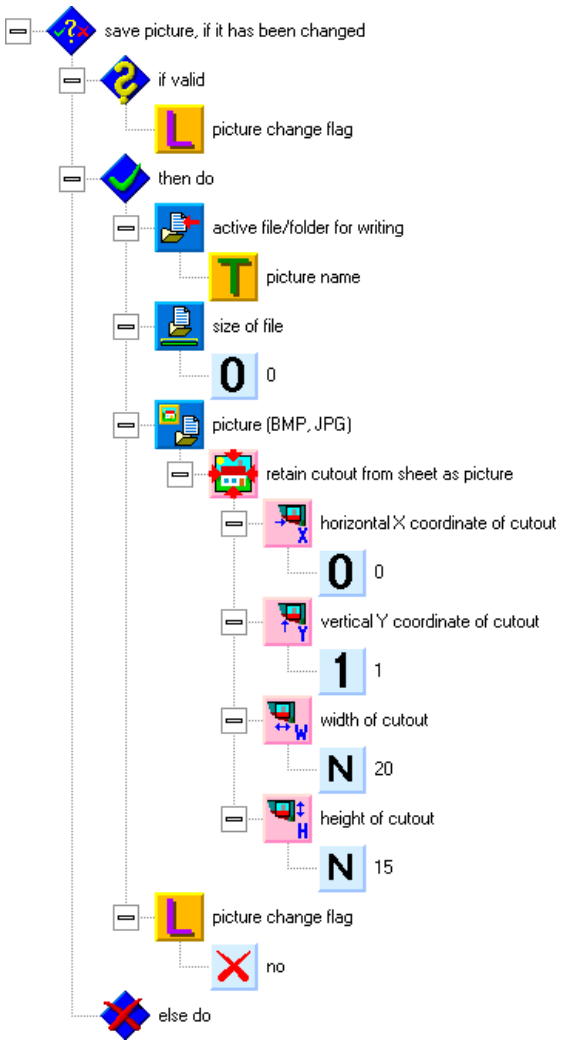
The structure for the right mouse button will be similar, and so we can copy the whole structure for the left button. In the condition test, we will replace the element that tests the left button with an element for the right button. The structure for buttons and color selection remains unchanged. The filling structure will be deleted and replaced by a **pick color under mouse cursor** structure. Here, we will test the **index of color under cursor** variable to check if there is a valid color under the pointer (to make sure that it is not black or a non-standard color). If the pointer is above a valid color, we will set it as the new selected color, recalculating the color to the X coordinate.



Run the program and test if you can control the choice of colors and pictures with the right mouse button in the same way as with the left one. By right-clicking into the picture, you can pick the color under the pointer and set it as the selected color.

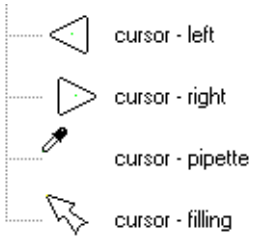
Finally, we can pay attention to the **save picture** function. Its contents are on the following picture. First, we will test if the picture has been changed and needs to be saved. If it does, we will use the picture name (from the list of picture names) as the **active file/folder for writing**. Then we will set **size of file** to **0** to make sure that no old data can remain in the file behind the picture. To save the picture, we will cut the picture from the window with the **retain cutout from sheet as picture** function, and then we will write the picture into a file by passing it to a **picture** element. Finally, we will turn off the **picture change flag**.

By adding the **save picture** function behind the main program loop, we will ensure that the picture will be saved upon quitting the program with the **Esc** key. The program has all the functionality now and you can test it. Verify that the changes in the pictures are saved upon both switching to another picture and quitting the program with the **Esc** key.

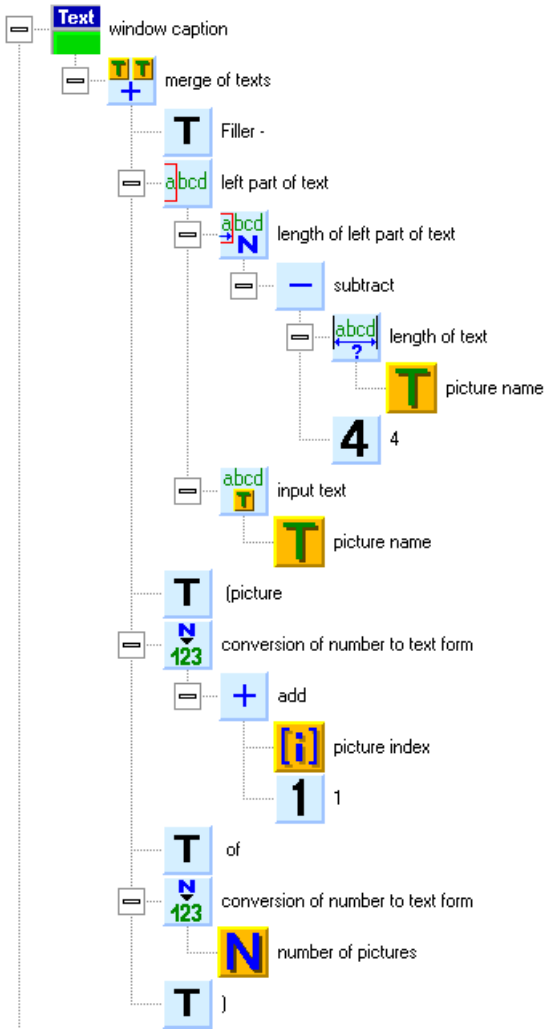


We can make further improvements to our program. For example, we can display the picture name, picture number and total number of pictures in the window title. We will put the structure for this into the beginning of the **load picture** function, as illustrated on the picture on the following page. This will result in a text like e.g. this one: "**Filler — House (picture 4 of 50)**". You cannot see it on the picture, but don't forget about spaces in the appropriate places in the text (behind the dash in "Filler — " and on both sides of " (picture " and " of "). We will display the picture name without the extension, and so we truncate the last four characters (the period and the BMP extension).

Another improvement will redefine the pointer appearance. In the **Global Variables and Functions** window, prepare four items with pointer pictures — two arrows, a pipette and a filling.



The pictures have black outlines, the surroundings are transparent, and the insides are white. On the tips of the pipette and the filling pointer, add a yellow dot as the positioning indicator (with color inversion). In the arrows, the positioning indicator stays in the center, which is the default setting.



In the **program initialization** group, we will use **setting shape of mouse cursor** commands to specify the appearance of the pointer in the individual parts of the window. We will define the pipette for the color choice fields, the arrows for the movement buttons, and the filling pointer for the picture area.

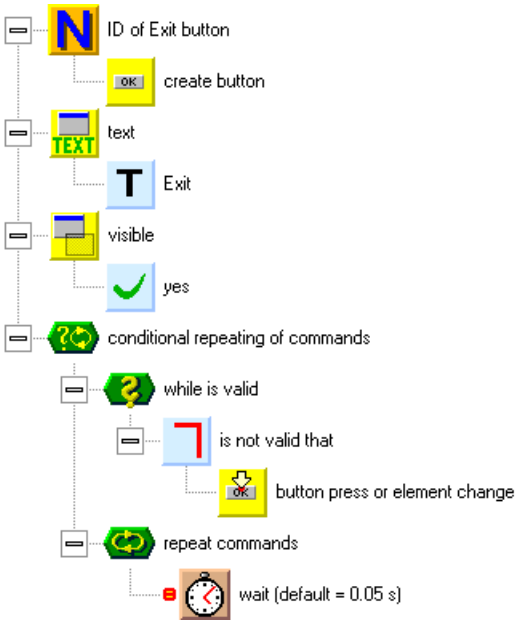
To understand the function of the command that defines the pointer appearance, you can think of the areas for definitions of the appearance as of rectangles, which (when added) overwrite the old definitions. The topmost definitions are always in effect. If we don't specify any pointer picture, then the standard appearance of a mouse in a window will be used. If colored pointer is defined in Windows, then the program will use this colored pointer. If we don't specify an area, then the mouse pointer is defined for the whole window. All definitions can be cancelled by inserting this command without any parameters.

16 Dialogs

After successful experiments with graphics, we will look at another area — dialogs. This is not talking to friends, as some might think. A dialog is a window used for communication with the user. Through it, the user decides on the next actions of the program, passes information to the program, and gets information from it as well.

Create a new program called **Dialogs**. The elements for working with dialogs are in the **controls** group, **dialogs** subgroup. We will begin by creating a simple button.

In the **Global Variables and Functions** window, create a numeric variable called **ID of Exit button**, and in the main function, create the program as illustrated bellow.



When you run this program, you will see a gray surface with a button saying **Exit** in the middle. By clicking the button, you can close the program. What is happening inside it?

Let's start with the numeric variable. The **ID of Exit button** means "*The identification code of the Exit button*". Each element in a window has its ID, which we can use to refer to the element. The identification number is assigned to each element automatically when the element is created. We don't have to care about the ID value; we will just store it and use it to refer to the element.

The first command in the program creates a normal button. The function creating the window element returns an identification code, which we will store for later use.

When we create the first window element, the program switches into dialog mode. The graphical sheet of the program disappears, and a one-color sheet (usually gray) appears. From this moment on, user input is carried through the window control elements. The graphical mode will be restored only when the last window element disappears.

When working with dialogs, the commands and functions are performed on the selected dialog element. The selected element is specified by the **element number** element, into which we pass the element ID. Don't confuse the selected element and the element with user input focus (e.g. an active text box, into which the user can type text). The selected element is just an internal pointer of the program and it specifies the element with which the program will manipulate. When a new element is created, it is automatically set as selected, and so we don't have to set the selected element in our program so far.

The second command in the program sets the **Exit** text into the button. For this, we will use the **text** element. In buttons, the text displays in the center as the button description. Other elements can have text as well — e.g. a radio button name, a group box title, an editing field text, a selected line in a list, or a window title.

The third command, **visible**, makes the element visible. All elements (even windows) are invisible when they are created. This allows us to set the necessary properties of the elements, without the elements "traveling" around the window, which would not look well. For this reason, don't forget to turn visibility on after you set the elements properties.

As we have said, the control over the program is passed to the window elements now. This will change the appearance of the program main loop. We will use conditional repeating again, with the condition **is not valid that** **button press or element change**. The **button press or element change** element indicates that an action was performed with the element. The type of action depends on the element. With buttons, it is pressing, with radio buttons, it is switching, with editing boxes, it is a change of the text in it. The flag is turned off after it is tested. The window has its change flag as well, which indicates that the change flag of one of its elements was set.

If you don't like the position of the button in the middle of the window, insert a **vertical coordinate** element with a numeric parameter of **1** after the button creation, but before it is turned visible. This moves the button towards the bottom of the window.

When using dialog elements, you will encounter a limitation of the number of colors. In the 256-color screen mode, the number of colors for the dialog elements is limited to 20 basic colors. Programs running in this video mode have to share the 256 possible colors. Dialogs usually don't require more colors, and so they are limited to Windows basic colors. In spite of this, you can use more colors even in dialogs by using the **create picture** element. The picture element uses the full range of colors, just like in the graphical mode of a program.

You can get most experience with programming by experimenting. If you don't understand the function of any element, highlight it with the mouse and press **F1**. This displays a comprehensive help for that element. The sample programs are also a good source of information. After you open a sample program, highlight the appropriate element in the **Basic Elements, Trash** window. The bottom right corner of Peter's window displays the number of times the element is used in the program. By clicking the **Previous Use** and **Next Use** buttons, you can move through the appearances of the element in the program.