Nils M Holm

# WRITE YOUR OWN PROGRAMS

# Contents

# First Contact

## What is Programming?

Programming is the art, craft, or job of telling a computer how to solve a specific task. It is an art, craft, or job depending on whom you ask. There are the pragmatic characters who sit in front of a screen from nine to five in order to bring home a monthly paycheck. To them programming is a job like any other, it just happens to be what they were trained for. Then there is the craftsman, who could be described as the more enthusiastic cousin of the dayjob programmer.

Craftsmen take pride in their trade. They want to do things right and create programs that are correct and reliable, and they can be found in both corporate and amateur environments. The term “amateur” is by no means an indicator of the quality of their work. The amateur just happens to do what he does because he likes it, and not because he expects some sort of compensation.

Some craftsmen are amateurs and professionals at the same time, because they work on their own projects in their spare time. Sometimes there is an overlap between their hobby and their profession. Most “Free Open Source Software” (FOSS) is created by people who would probably see themselves in the “craftsman” category. The term “software” is more or less just a fancy word for “program”. “Open source” means that the user of a program is — theoretically — able to change the program, and “free” means, well, “free”. In fact the latter is the most controversial part in the acronym FOSS, and we will get back to that later.

The artisan is someone who writes programs that are beautiful. He is not happy until the program not only does what it is supposed to do, but also does it in a specific way that is considered the be “elegant”. Elegance can be thought of as correctness and efficiency taken to the extreme with some amount of beauty added. The beauty can be expressed in the visual form of a program, in its inner workings, or both. Artisans are rare in

both corporate and FOSS environments. Most of them probably pursue programming as a hobby, but, again, being a “hobbyist” is no measure of quality here, but just points at the context in which the activity of programming takes place.

To become either a white-collar programmer, a craftsman, or an artisan requires knowledge and experience. What is common to all three of them is that they know how a computer works internally, how to use the tools that are used to create programs, and the mastery of at least one “programming language”.

A programming language is a language, much like English or other natural languages, but much simpler and with great emphasis on mathematics and logic. Programming languages are also very *formal* in the sense that there are often only a few ways — or even a single one — to express a certain thing.

# What is a Program?

A program is a set of instructions are are formulated in a programming language. The purpose of a program is to instruct a computer to perform a specific task. It instructs the computer in a very detailed way. When instructing a human being in this way, it would probably accuse you of “micro-managing” and walk away — and rightly so. A computer, though, requires very precise and detailed instructions.

For example, if you wanted someone to ask for your name and then greet you with the phrase “Hello, *name*”, where *name* is your name, this would certainly be a weird request but, if both humans are willing to play along, easy enough to achieve. Just tell the other person:

*Ask me for my name and then*
*greet me with “Hello” and my name.*

With a computer you would have to be much more precise. Of course computers usually do not talk nor can they understand spoken language. [1] So communication with the computer will be

[1] They are getting there, but programming such things is *far* beyond the scope of this book.

performed using the most basic way in the following: the computer will read *input* from the keyboard and write *output* in the form of text to the screen.

Input and output are fundamental parts of a program. Without output, no matter in which form, a program would have no *effect*. You would never know that it ran in the first place. And without input a program would always perform exactly the same task over and over again — which would not be very interesting.

The following is a program written in the Minimal Procedural Language T3X/0 or, in short, T3X. It does what has been discussed above: ask for a name, read the name from the keyboard, and print a greeting on the screen.

```
use t3x: t;
use console: con;
use input;

var UserName::50;

do
    con.setup(0);
    con.writes("What's your name? ");
    input.readln(UserName, 50);
    con.writes("Hello, ");
    con.writes(UserName);
    con.writes("!\n\r");
    con.shutdown();
end
```

Note the horizontal lines above and below the program. These lines indicate the beginning and end of program text.

The first block of lines in the program tells the computer about functions that will be used in the program:

```
use t3x: t;
use console: con;
use input;
```

In this case the program will use basic T3X functionality, the

“console”, and input from the user. The console or “console terminal” is an ancient device that consists of a screen for displaying text and a keyboard. Modern computers do not normally have a console terminal, but they simulate it in a window on a graphical display.

Every T3X program (and also most programs in other languages) have to declare at the beginning of the program text what they are going to use. “Using” is sometimes also called “requiring” or “importing”, and the used, required, or imported entities are known as “libraries”, “modules”, or sometimes “packages”.

The above program uses the modules **t3x**, **console**, and **input** and the “aliases” **t** for **t3x** and **con** for **console**. This means that the module **t3x** can also be referred to as **t** and the module **console** as **con**.

Parts of programs are often typeset using a “typewriter style” typeface, as used in the words **console** or **t3x**. This book also follows this convention.

The next part just says

```
var UserName::50;
```

It means: reserve 50 characters and associate that space with the name **UserName**. The space will be used to store the input from the keyboard later in the program. **Username** is a “variable” of the program and it is declared using the keyword **var**. In most languages every variable of a program has to be declared before it can be referred to. (In fact *everything* has to be declared before it can be referred to.) Note that this variable will contain *text*, i.e. characters received from the keyboard, and not numerical values. Unlike variables in simple mathematical formulas, variables in programs are not limited to numbers.

The main part of the program, also called the “main program” or “main body”, consists of the remaining lines:

```
do
    con.setup(0);
    con.writes("What's your name? ");
    input.readln(UserName, 50);
```

```
    con.writes("Hello, ");
    con.writes(UserName);
    con.writes("!\n\r");
    con.shutdown();
end
```

The keywords **do** and **end** merely enclose the “statements” of the main program. A “statement” is where the music plays, so to say. Every statement tells the computer to “do something”, where this “something” is typically a highly specific thing. For instance, the statement

```
    con.writes("What's your name? ");
```

writes the text “What’s your name? ” to the console screen. The **con** in **con.writes** indicates that the **writes** procedure is taken from the **console** module. Remember? **Con** is an alias of **console**. A “procedure” is a thing that does something, like writing text to the console screen.

BTW, did you notice the blank character between the question mark and the double quote character in

```
    con.writes("What's your name? ");
```

It is important! Such small details often influence the way in which a program operates and sometimes makes the difference between a working and a non-working program. It is a good idea to get used to paying attention to such small details. The program development tools will help to catch many small mistakes, but in the end it is the programmer who makes a program work.

Except for **con.writes**, **input.readln** is the only other procedure that needs to be know in order to understand the above program. [1] In the above program **input.readln** reads *no more than* fifty characters [2] from the keyboard and stores them in the space associated with the variable **UserName**.

The other thing that needs to be understood is that programs execute, or run, from top to bottom — unless told otherwise. So what does the above program do?

---

[1] We will ignore **con.setup** and **con.shutdown** for now.
[2] 49, actually, but we will get to that later

- It prints “What’s your name?” and a blank character;
- then, it reads user input into the variable **`UserName`**;
- then, it prints “Hello, ”, the content of **`UserName`**, and the strange text "!\n\r".

So here is some sample output (and input, in thin characters) of the program when it runs:

```
What's your name? Nasenbaer
Hello, Nasenbaer!
[]
```

It first prints the question and a blank character. The blank character separates the question from the user input, so that the input does not appear like it is glued to the question. The program then waits for the user to type their name and press the ENTER key. In the example the user enters “Nasenbaer”. [1] The program responds by greeting the user and moving the cursor (represented by the block character) to the next line.

The cursor is the point on the screen where text output will appear. It is typically a solid block or an underline character. On some displays it blinks, on others it remains solid. The characters

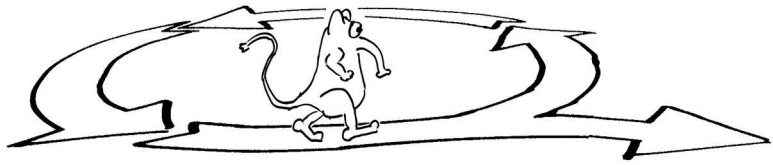
```
\n\r
```

move the cursor to the beginning of the next line when displayed on the terminal screen. The **`\n`** and **`\r`** characters are called “special characters”, because they have a special function. There are more of them and they will be explained later.

What is left unexplained in the program are the **`con.setup`** and **`con.shutdown`** procedures. For now it is sufficient to know that every program that uses the console device has to start with the former and end with the latter.

What is most important while learning to program is to *experiment* with programs: run them, change them, and run them again. So the next chapter will explain how to install a programming language on your computer.

[1] Nasenbär is the German name of the coatimundi, a cute little animal.



# THE CYCLE OF DEVELOPMENT

# The Development Cycle

Program development, the act of writing and improving a program, is divided into distinct activities that take place in a cycle:

1. Analyze the problem
2. Write or modify program code
3. Compile the program
4. Test (run) the program
5. Go back to step 1

In the first step, the problem to solve is understood to the best degree possible. It is not always possible to understand the problem in its entire complexity, and some facets of the task may only reveal themselves in later iterations of the cycle.

Programs like the one presented in the initial chapter are rather easy. The analysis would proceed like this:

- The program uses the console, so I need the console module.
- The program reads input, so I need the input module.
- The program first writes a question, then reads input, writes a greeting, and terminates.

At this point, writing the actual program is rather straight forward. If it does not look that way at this point, this is because knowledge about the details of the programming language and environment is missing. The logic itself is rather simple.

In the next step, a text editor is used to write the program text. A text editor is different from a “word processor”. It only processes pure, raw *text*, and cannot do things such as different typefaces, different text sizes, boldface and italic characters, underlining, etc. This means that the various “office” programs are not suitable for writing program text.

On a Windows system you could use the “Notepad” editor, but it does not interact well with DOSBox, because DOSBox will not see any changes to a file until it is closed and started again. There are

lots of popular text editors for DOS, though, and some research on the Internet will reveal a plethora of choices.

Unix systems typically come with at least one text editor pre-installed. If you are using the T3X compiler directly on Unix (not in DOSBox) and are familiar with one of the Unix text editors, just use it.

Then T3X/0 comes with its own editor called NOE. [1] [2] NOE has one big advantage under DOS: it can compile T3X/0 programs without having to leave the editor. Other editors may be able to do that, too, by running an external command, but in NOE it takes literally two key presses to compile a program and another two to run it. NOE uses the ancient WordStar user interface, which was very common in the 1980's.

After writing program code in the text editor, it is time to see if there are any obvious errors in it. Formal errors are typically found while compiling a program, and the compiler will report them. A program is compiled by saving the text in the editor and then passing the text file to the compiler. The compiler will translate the program to a form that can be executed (run) by a computer and create an "executable file", also just called an "executable" or a "binary".

If the compiler finds any formal errors in the program, it will report them and not create any executable. Any previously existing executable of the program will be deleted in this case. If this happens, step (4) — testing — has to be omitted and the formal error has to be corrected first.

If compilation succeeds, the resulting executable program is run and its behavior is compared to the expected behavior of the program. If the behavior of the program is as expected, the program development cycle ends, and the program is finished. When developing more complex programs, this point in the cycle is not as well-defined as described here, though. Subtle deviations

---

[1] Nils' Own Editor, a play on "Joe's Own Editor", which is an excellent WordStar-like editor.

[2] In fact there is another editor that comes with T3X/0. It is called VEE and resembles the interface of the Unix editor VI.

from the expected behavior may be found when the program is used later, improvements may come to mind, or additional ways to use the software may be discovered, which may require to extend the program code.

# Unpacking the Examples

All programs that will be discussed in this book can be downloaded from the book homepage at

**`http://t3x.org/wyop/`**

Download the file **`wyop-unix.zip`** (for Unix) or **`wyop-dos.zip`** (for DOS and DOSBox) or **`wyop-cpm.zip`** (for CP/M) and unpack it in a location that is convenient, e.g.:

```
unzip wyop-dos.zip
```

Under DOSBox, this would be inside of the DOS directory. On a Unix system, it does not really matter where you unpack it. The files from the ZIP file will be extracted to a subdirectory named like the file. To compile and try programs in that directory, go to the directory and follow the instructions in the subsequent section.

The disk image for the Agon Light 2 computer contains all the examples, so there is no need to install them.

# Using the T3X Compiler

There are four variants of the T3X/0 compiler, and which one you are using depends on the environment in which you are using it and how you built it. The four variants are

- The DOS compiler, which you would use under DOSBox
- The CP/M compiler, which you would use on a CP/M computer
- The native (“fast”) compiler on Unix
- The (“slow”) bytecode compiler on Unix

No matter which variant you are using, they all expect T3X/0 programs to be stored in files that end with the characters “.t” (dot-tee). For example, the program from the initial chapter is

contained in the file **hello0.t** (hello-zero dot tee) in the examples. When a program is contained in a file that does not end in ".t", the compiler cannot compile it.

## The DOS Compiler

The T3X/0 compiler for DOS is pre-configured to be installed in the directory **C:\T3X0**. If you install the compiler in a different location, you will have to run the program **TXCONF.COM** and enter the chosen location when asked.

The DOS compiler is very simple. Like all T3X/0 compilers it is invoked by the **tx0** (tee-ex-zero) command and it expects a file name *with the ".t" suffix omitted* as its only argument. For example, to compile the example program from chapter one, run the following command:

```
C:\T3X0PROG> tx0 hello0
```

(Do not type the **C:\T3X0PROG>** part, it is the DOS prompt.)

The compiler will then produce a file named **hello0.com**, which is an executable program. Like all executable programs, you can run it by just typing its name:

```
C:\T3X0PROG> hello0
```

That's all there is to it. Compiling and running programs on DOS or under DOSBox is as easy as typing **tx0 program** and then **program** (where "program" is, of course, the name of an actual program).

If you are using the NOE editor under DOS or DOSBox, this is even easier. Run the command

```
C:\> noe hello0.t
```

to load the program into the editor, and then press control-V (to activate help), and then control-O followed by "c". This will automatically compile the program. To run the compiled program press control-O and then "r".

Control-V is most often written **^V** by programmers, so it is a good idea to remember this abbreviation. To activate help, press **^V**, to

compile a program, type **^OC** (first **^O**, then **c** — the “c” is actually lower-case, no need to press “shift”). To exit from the editor, press **^KX** (or **^KQ** if you do not want to save the text).

## The CP/M Compiler

The CP/M compiler works in exactly the same way as the DOS compiler. It is configured to be installed on disk drive **H:**. If you install it on a different disk, you will have to run the **TXCONF** command and enter the chosen drive letter when asked.

When using the CP/M compiler on an Agon Light 2 computer, it is recommended to put the example programs on the same disk as the compiler. The **cpmh.dsk** image file already contains the example programs.

You can compile and run programs in the same way as on DOS. E.g. to compile the program **hello.t**, type;

```
H>tx0 hello
```

and to run it, type

```
H>hello
```

(Do not type the **H>** part, it is the prompt of the CP/M console command processor.)

The NOE editor is also available on CP/M and works in the same way as on a DOS system.

## The Unix Compiler

Both the native code compiler and the bytecode compiler for Unix are used in almost the same way as the DOS compiler. To compile the example program **hello0.t**, type

```
$ tx0 -c hello0
```

The **-c** part is called a *compiler option*. In this case it makes sure that the program will be able to use the console interface. The DOS compiler does not need this option, it automatically includes the console interface when requiring it in the program code. On

Unix the compilation process is more complicated and therefore the compiler needs a heads-up.

In case you wonder what happens if you omit the **`-c`** option, just try it. Curiosity is a good trait to cultivate as a programmer. Depending on the compiler you are using, either nothing at all will happen or the system will spit out *a lot of* error messages and compilation will fail. Do not worry, though! Nothing really bad can happen. Just supplying the option the next time you compile the program will solve the problem.

If, on the other hand, you use the **`-c`** option in a case where it is not needed, not much will happen at all. Compilation will succeed. Maybe the resulting executable program will be a bit larger than it needs to be, but this is something we are not concerned with at this point. So for now it is a good idea to always use the **`-c`** ("console") option on Unix.

If you are running the native code compiler, it will generate a file without any suffix. E.g. compiling **`hello0.t`** will generate an executable file named **`hello0`**. The bytecode compiler, on the other hand, will generate a file named **`hello0.tc`** (where "tc" stands for "Tcode", which is the kind of bytecode used by the compiler).

You can start a native executable by just typing its name, but it is good practice — or even necessary — to prefix it with the characters "**`./`**" (dot-slash) to make sure that the program *in this directory* is executed. So to run the program **`hello0`**, type

```
$ ./hello0
```

The bytecode program **`hello0.tc`** *cannot* be started in this way. In order to run a bytecode program, it needs to be passed to the Tcode Virtual Machine (TCVM):

```
$ tcvm hello0
```

A virtual machine is like a tiny (in this case) computer inside of your computer. It executes programs that have been compiled for it. The advantages of virtual machines are:

- it is much easier to compile programs for them;
- their code runs on every type of computer as long as it has the required virtual machine.

Their disadvantages are

- their code is slower than code for “real” machines;
- programs need to be passed to the virtual machine for execution.

Because virtual machines are so simple, the bytecode compiler is more likely to compile successfully than the native code compiler on a Unix system.

In short,

- Use “**`tx0 -c`** *program*” to compile a program under Unix.
- Use “**`./`***program*” to run a native program
- Use “**`tcvm`** *program*” to run a bytecode program

> BTW, when programming on Unix, it is a good idea to open two terminal windows: one for editing the program code, and one for compiling and testing. This way you do not have to leave the editor in order to perform the other steps of the development cycle.

Of course, nothing stops you from using the NOE editor on a Unix system,[1] if you are so inclined. It is in the file **`programs/noe.t`** in the T3X/0 source directory. You just need to compile it first. [2]

---

[1] Or the VEE editor, **`programs/vee.t`**

[2] Use the command

```
tx0 -c -s programs/noe noe
```


GETTING YOUR
FEET WET

# Getting Your Feet Wet

Being familiar with your tools is a basic prerequisite when writing programs. It is hard enough to develop the logic of a program — wrestling with the tools at the same time would be a frustrating experience.

## Familiarizing Yourself with the Editor

Most time in the development cycle will be spent editing text. So the compiler may be the tool that is *started* most often, but the editor is the one with which you spend most of your time. Editing text without thinking much about it makes the task much, much easier.

So the first step on the journey is to decide which editor to use.

No matter in which environment you are using T3X, if it has an editor that you are already familiar with, *do* use it.

There is one feature, though, that is pretty much indispensable, and that is the capability to move the cursor to a specific line number. Text files are divided into lines, so when the compiler reports an error, it will also report the number of the line in which the error occurs. Without a command for moving the cursor to a specific line number, it would be all but impossible to find the location of the error.

Another feature that is very convenient to have, especially on DOS and CP/M systems, is a command to run a “subshell” or “child process”, i.e. a way to run programs without leaving the editor. Some DOS editors have such a command, on CP/M it is rather rare.

Such a command allows you to compile T3X programs without having to end the editing session first. Without such a command, you would have to exit from the editor, compile the program, restart the editor, and go back to the location in the text where you left.

If your favorite editor should lack any of these features, it would probably pay off to look for alternatives.

Both the NOE and VEE editors that come with T3X/0 have both of these commands. While they cannot run arbitrary sub-commands, they both have a command for compiling and running the program that is currently loaded. Both editors are contained in the DOS and CP/M distributions of T3X, so if you are running T3X under DOS, DOSBox, or CP/M, you can just use them.

Unix usually provides better editors, but if you want to, you can use NOE or VEE even on a Unix system. You just have to compile them first. Using the native T3X compiler, the following commands compile them (when issued inside of the **t3x0** directory where the compilers sources are kept):

```
$ tx0 -c -s programs/noe noe
$ tx0 -c -s programs/vee vee
```

When two arguments are given to the T3X/0 compiler, it will interpret the first one as its input and the second one as its output file. E.g., the first of the above commands will compile the program **noe.t** in the directory **programs** and put it in the executable **noe** in the current directory.

The editors can also be compiled using the bytecode compiler:

```
$ tx0 -b programs/noe noe
$ tx0 -b programs/vee vee
```

(The **-b** option is not necessary when only the bytecode compiler is installed.) When compiled in this way, the editor, like any other bytecode-compiled T3X program, has to be started using the TCVM, e.g.:

```
$ tcvm vee programs/apfel.t
```

> In case you try the VEE editor and get stuck, you can exit from it by pressing ESC and then typing **:q!**.

# Your First Program

No matter which editor you have chosen, try it now! Open a new file named **test.t** and enter the following text:

```
do t.write(T3X.SYSOUT, "Hello, World!\r\n", 11);
```

Save the file and compile it. Try to do so without leaving the editor. Compilation will fail and the compiler will complain about an undefined symbol:

```
Error: 1: undefined: t
```

Oops. The program uses two symbols of the T3X core module: the procedure **t.write** and the other symbol **T3X.SYSOUT**. So the core module has to be required at the beginning of the program. Modify the program as follows:

```
use t3x: t;
do t.write(T3X.SYSOUT, "Hello, World!\r\n", 11);
```

Try to compile it again. The compiler will complain again, this time because the program ended at a point where it did not expect it to:

```
Error: 3: unexpected end of compound statement
```

The number 3 is the line number in which the error occurred. Note that it says line number three, while the file has only two lines. Sometimes line numbers are off by one, because the compiler had to keep scanning before it could find an error. In the above case, it moved past the end of line two before it could notice that nothing comes after it.

> Lines numbers in compiler error messages are sometimes one too big.

Every main program begins with the keyword **do** and ends with the keyword **end**, which is missing in the above case. So modify the program again and add the **end** keyword at the end of the

program. It is also a good idea to reformat the program to make the **do**···**end** more visible and add some space between the **use** block and the main program:

```
use t3x: t;

do
    t.write(T3X.SYSOUT, "Hello, World!\r\n", 11);
end
```

Indentation (moving text to the right) is usually done with TAB characters. So there should be only one character to the left of the text **t.write**, and it should be a TAB. It is inserted by pressing the key labeled "TAB" or "↹", which can usually be found on the left in the second row down on the keyboard.

Some programming languages prefer spaces instead of TAB characters, and even people using the same language may have different preferences. If you want to use spaces instead of TAB characters, feel free to do so. Just keep things consistent, do not use different ways of indentation in the same file.

Now compile the program again. It should compile successfully now. If it does not, double check that you entered exactly what the example says. Now run the program. It will print

```
Hello Worl
```

which is not the expected outcome. The **t.write** procedure expects the number of characters of the output in its third parameter, which in this case is 11. Count the characters in the text to be printed carefully, remembering that **\n** and **\r** are single characters. The author counted 15 — what about you? Update the program with the correct size and compile and run it again! It should now print the text

```
Hello, World!
```
[1]

and move the cursor to the next line. Well done! This is what the program development cycle looks like.

[1] This oft-repeated greeting first appeared on page 6 in [K&R1978].

# What is a Program? — Revisited

Here is the **hello0.t** example program from the first chapter again. This time, let us have a closer look.

```
use t3x: t;
use console: con;
use input;

var UserName::50;

do
    con.setup(0);
    con.writes("What's your name? ");
    input.readln(UserName, 50);
    con.writes("Hello, ");
    con.writes(UserName);
    con.writes("!\n\r");
    con.shutdown();
end
```

The first three lines declare the modules on which the program depends. A module is a collection of things that make the life of a programmer easier. For example, the **console** module contains procedures that write text to the console screen or, more probably these days, to a terminal window on the screen.

A procedure can be thought of as a small, more or less self-contained program that can be "called" by another program. Whenever a procedure is called, it does what it is programmed to do an then returns control the program that called it. For example, the procedure call

```
    con.writes("What's your name? ");
```

passes the text "What's your name? " (without the quotes) to the procedure **writes** that is contained in the module **console**. The module is here referred to by its alternative name, **con**.

The procedure then prints the text and returns to the program. Notice that different texts are passed to **con.writes** in the above program. The procedure always prints what is passed to it. Something that is being passed to a procedure is called a "parameter" or an "argument"[1] of a procedure. Arguments modify the behavior of a procedure. In the case of **con.writes** the only argument to the procedure modifies what it displays.

The program imports or requires three modules named **t3x**, **console**, and **input**, but it only seems to use **console** and **input.** This is because **console** depends on **t3x**. A module depends on another one when it uses procedures (or other entities) defined in that module. Pretty much all programs depend on the **t3x** module in one way or another, either directly or, like the above program, indirectly.

The **t3x** and **console** modules are modules that belong to the T3X/0 language, so every T3X/0 program can just require them without having to provide a copy of their program code. The **input** module, on the other hand, is not a standard T3X module, so its code has to be present in the same directory as the program that requires it.

Many modules, even standard T3X modules, are also written in T3X/0. We will have a look at some of them later.

The **input** module exports only a single procedure named **readln** (short for "read line"). It reads input from the keyboard and stores it in the storage indicated by its first parameter. In the example program, the storage is provided by the 50-character variable **UserName**. The second parameter of **input.readln** specifies the maximum number of characters to accept from the keyboard plus one, so at most 49 characters can be typed in the example.

The storage used to store the name is allocated and associated with the name **UserName** by the declaration

```
var UserName::50;
```

[1] sometimes an "actual argument"

A “declaration” is a statement that announces the existence of something or the dependency on something. The first three lines declare dependencies on modules, and the line starting with **`var`** (for “variable”) declares the existence of a variable that can contain a certain amount of text.

The term “statement” is a very generic one that describes many elements of a program. A declaration is a statement, but a stand-alone procedure call (like those in the example) is also a statement.

A special case of the statement is the “block statement”, also called a “compound statement”. It consists of the keyword **`do`**, followed by other statements followed by the keyword **`end`**, i.e. the keywords **`do`** and **`end`** enclose other statements and combine them to form a single statement. The main body of a program is a compound statement. Remember the compiler complaining about the “unexpected end of a compound statement” when the **`end`** keywords was missing in an earlier example? Every **`do`** has to have a matching **`end`** somewhere.

## Making Decisions

This is all very fascinating — hopefully! — but also maybe not very impressive. Asking for a name and then printing a greeting — surely programs can do more than that!

Let us start by finding an obvious flaw in the example program. Compile it, if you do not already have an executable around, and run it again. When it asks for your name, just press ENTER. The program will dutifully print

```
Hello, !
```

and terminate. Technically correct, but not very beautiful.

This behavior can be improved by making a decision. The decision will depend on the amount of characters entered by the user. Fortunately, the **`input.readln`** procedure returns the number of characters typed before pressing ENTER. Technically, this turns the procedure into a “function”, although the distinction is arbitrary in T3X. Every procedure returns a value, but some procedures

return a value that is interesting, and some return a value that is nondescript.

The value returned by a procedure can be stored in a variable by "assigning" the value returned by the procedure to a variable. An assignment looks like this:

```
k := input.readln(UserName, 50);
```

It assigns the value delivered by **input.readln** to the variable **k**. Of course, **k** has to be declared before a value can be assigned to it. The following program makes a decision based on the value returned by the **input.readln** procedure:

```
use t3x: t;
use console: con;
use input;

var UserName::50;

do var k;
    con.setup(0);
    con.writes("What's your name? ");
    k := input.readln(UserName, 50);
    ie (k > 0) do
        con.writes("Hello, ");
        con.writes(UserName);
    end
    else do
        con.writes("Goodbye");
    end
    con.writes("!\n\r");
    con.shutdown();
end
```

The program is very similar to the original example, but there are a few differences:

(1) The new program uses the value returned by **input.readln** and stores it in **k**, while the old program ignores it.

(2) The new program uses a statement of the form **ie**···**else**··· to select one of two alternative statements, depending on the value stored in **k**.

If the value of **k** is greater than zero, expressed by the term **k > 0**, the user has entered more than zero characters, and the program prints the usual greeting. If the user has not entered any name, the alternative branch is executed, and the program prints "Goodbye".

The general form of the **ie/else**[1] statement is

**ie (***expression***)**
    *consequent*
**else**
    *alternative*

The indentation is optional, as it is pretty much everywhere in T3X. The statement

**ie (***expression***)** *consequent* **else** *alternative*

would have exactly the same meaning, but would be harder to understand when reading the program. Therefore, the form consisting of multiple lines is preferable in the vast majority of cases.

The **ie/else** statements works by first computing — or evaluating — the *expression*, which yields a value. In the case of the greater-than operation $x > y$, the resulting value it either "true" or "false". When it is "true", the *consequent* branch is executed (and the *alternative* branch is skipped) and when the value is "false", the *alternative* branch is executed (and the *consequent* branch is skipped). So the statement

```
ie (1 < 2)
    con.writes("Yes");
else
    con.writes("No");
```

would display "Yes" on the console screen.

Note that there are *two* consequent statements in the example

[1] The name **ie** is itself short for if/else.

program, though, while the **ie/else** statements expects only one:

```
con.writes("Hello, ");
con.writes(UserName);
```

This works, because the two statements are packaged in a compound statement, which effectively makes them one single statement from **ie/else**'s point of view. Any number of consequent and alternative statements can be enclosed in a **do/end** block:

```
ie (expression) do
    consequent-1
    consequent-2
    ...
end
else do
    alternative-1
    alternative-2
    ...
end
```

It is even possible to have a compound statement in one of the branches and a single statement in the other, although this is widely considered to be poor style.

One final remark, in case you wonder why the variable **k** is declared inside of the main body, right after the **do** keyword: there are two kinds of variables[1] in T3X: local ones that are declared inside of compound statements, and global ones that are declared outside of compound statements, often right after the module dependencies.

In general it is good style to keep variables local to the point where they are used, but sometimes, variables are used in so many places that it is better to make them global. Variables that allocate larger amounts of storage, like

```
var UserName::50;
```

[1] three, actually, if you include module variables

in the original example, are sometimes declared globally, even if they are only used in one location.

The modified program is contained in the file **`hello1.t`** in the collection of example programs.

## Repeating Yourself

Now imagine that you wanted to ask for a name and print a greeting multiple times. Of course you could just repeat the program fragment

```
con.writes("What's your name? ");
input.readln(UserName, 50);
con.writes("Hello, ");
con.writes(UserName);
con.writes("!\n\r");
```

as often as you would like the program to ask and greet. However, what if you decided otherwise before reaching the end of the program? What if you reach the end of the program, but would like to see one more greeting? Of course, this is a silly example, but nonetheless it illustrates an important point: sometimes you want to repeat a task without knowing in advance how often you want to do it. And then:

| Repeating yourself in a program is very bad style! |
|---|

So, after making decisions, here is another fundamental concept in programming: repeating things without repeating yourself. The following program will ask for a name and print a greeting *as long as* the user enters another name. Not the entire program is repeated here, only the essential part is given:

```
con.writes("What's your name? ");
while (input.readln(UserName, 50) > 0) do
    con.writes("Hello, ");
    con.writes(UserName);
    con.writes("!\n0r");
    con.writes("What's your name? ");
end
con.writes("Goodbye!\n\r");
```

Where there was an **ie/else** statement in the previous version of the program, there is a **while** statement now. The **while** statements does exactly what its name suggests: it repeats a statement — a compound statement in this case — *while* its expression evaluates to truth. More generally:

**while (***expression***)** *statement*

repeats the *statement*, while the *expression* yields a true value. Should the expression already be false when the **while** statement is encountered, the statement will *never* execute.

So what does the new **hello** program do? It prints a prompt, reads a name, and *while* the name is longer than zero characters, it prints a greeting and repeats. Note that in this case the value returned by **input.readln** is not stored in a variable first, but it is compared to zero immediately. The program fragments

```
while (input.readln(UserName, 50) > 0) do
    ...
end
```

and

```
k := input.readln(UserName, 50);
while (k > 0) do
    ...
    k := input.readln(UserName, 50);
end
```

basically do the same thing. The only difference is that the second one keeps the return value of **input.readln** in the variable **k**, while the first one never stores the return value, but uses it in the comparison immediately. Note that the call to **input.readln** has to be duplicated in the second version, because the first call is outside of the **while** statement and will therefore not be repeated.

The final version of the **hello** program is contained in the file **hello.t** in the examples, in case you want to try it.

What the **while** statement does is called “iteration” or, more commonly, a “loop”. The statement to be repeated is called the

"body" of the loop, and the expression to be tested at the beginning of a loop is called its "test".

# Summary

**Declarations** announce either a dependency on a module or the existence of a variable. Example:

```
var UserName::50;
```

**Modules** are collections of procedures (and other things) that can be imported and used by programs. Example:

```
use t3x: t;
```

**Procedures** are small programs that can be called by programs or other procedures in order to perform some action. Example:

```
con.writes("Hello, World!");
```

**Statements** are fundamental parts of programs. The term covers both declarations and executable statements, like procedure calls, assignments, selection, and loops.

**Assignments** assign a value to a variable. After the assignment the variable will have the assigned value. Example:

```
number := 123;
```

**Expressions** are parts of a program that have a value. Procedures return values, numbers have values, and variables have values. Examples:

```
x > 25
input.readln(UserName, 50)
x
```

**Programs** execute from top to bottom. The following statements would first print "A", then "B", and then "C' (and never any other sequence):

```
con.writes("A");
con.writes("B");
con.writes("C");
```

**Selection** chooses one statement or another, depending on the value of an expression. The following statements would print “Greater” if **x** is greater than 100:

```
ie (x > 100)
    con.writes("Greater");
else
    con.writes("Nope");
```

**Loops** repeat statements while an expression is “true”. The following statements would print “Hello” 10 times:

```
do var i;
    i := 0;
    while (i < 10) do
        con.writes("Hello\n\r")
        i := i + 1;
    end
end
```


AVING
RIGHT IN

# Diving Right In

## Components of a Computer System

The four major components of every computer system are:

- the central processing unit (CPU)
- memory
- mass storage
- user interface

The user interface is what the user interacts with, typically a screen and a keyboard, and these days also a pointing device, like a mouse or a trackball. Sometimes keyboard and screen are one device, like the touchscreens on smartphones. In this text, the screen is a text device, i.e. it is only capable of displaying letters, numbers, and a small set of special characters, like punctuation signs, math signs, brackets, etc. This is pretty much what computing looked like up to the 1980's.

The mass storage is where programs and data are kept for long-term storage. Mass storage is non-volatile, so it retains its data when the computer system is powered down. Typical mass storage devices these days are hard disks, solid state disks (SSDs), USB thumb drives, and SD cards. Except for hard disk drives these are all electrical devices with no moving parts. Hard disks use magnetic fields to store data on rotating disks ("platters") covered with a ferromagnetic material. Because they are electro-mechanical devices, they are more easily damaged by mechanical shock than solid state devices, but when powered down, they retain data more reliably. From a user's perspective, mass storage is where your data goes when "saving" things in a program, like text in a text editor.

Memory is volatile, so its content gets lost when powering down the computer, but it is also *much* faster than mass storage. Programs execute in memory and they keep all or at least most of their data in memory. "Loading" data means to transfer them from

mass storage to memory. When a program allocates storage, like the $50$ characters for the **`UserName`** variable in the initial example program, this storage is allocated in memory. So memory plays an important role in programming, and we shall return to it soon.

The Central Processing Unit (CPU) is what executes programs. The CPU has its own programming language called “machine code”, and this is the only language that it can interpret and thereby execute programs. Different types of CPUs have different machine languages. There used to be lots of popular CPUs, but these days there are basically two families left: the x86 or x86-64 on desktop machines and servers, and the ARM family on mobile devices (and increasingly also stationary devices). The T3X/0 core module is written in “assembly language”, which is the human readable form of machine language. Assembly or machine language is very low-level and hard to program, which is why compilers for more high-level languages are typically used to program computers. T3X is such a high-level language.

For example, here is an assembly language program (in 8086 assembly language) that computes $z \leftarrow x^2 + y$ (the result is stored in $z$) on the left, and the corresponding T3X/0 statement on the right:

```
mov ax,x            z := x * x + y;
mov cx,ax
mul cx
add ax,y
mov z,ax
```

Most of the information presented in this section is only background information and not *really* necessary to understand how a program works. Remember, though, that curiosity is a good trait to have as a programmer, so being interested in such details and trivia can help you become better at writing good programs.

The only thing we will have to dive into a little bit more is...

# List of Figures

# List of Coatimundi

# Index