T3X Corp.

Delivering Fine Software Since 1895

Rules of the Game

A *compiler* is a program that reads the source code of a program and outputs an executable form of the same program. The most important aspect of a compiler the generation of *correct* code, i.e. the executable program must perform exactly those actions which the source program describes.

Because a compiler is a program and translates *programs* from source to executable form, it may under some circumstances compile itself. A compiler that compiles itself is called a *self-hosting* compiler. The prerequisite for this to work is that the source language and the implementation language of the compiler are the same.

Generally, three languages are involved when talking about compilers:

- the source language
- the target language
- the implementation language

The source language S is the language which the compiler "understands", i.e. the form of the programs it *reads*. This is typically a *programming language*, such as C, Pascal, or LISP.

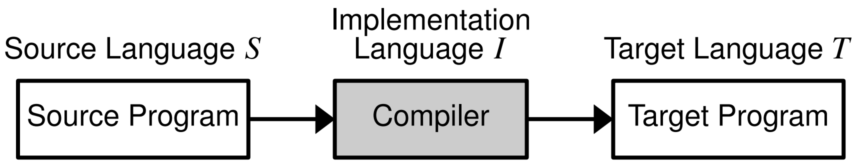

Figure 1: Compilation

The target language T is the machine language that the compiler outputs, for example: machine code for a Z80 CPU or bytecode for the Tcode Machine or the Java Virtual Machine. Executables are typically packaged in some executable file format, like JAR (Java Archive), ELF (Executable and Linkable Format), or COFF (Common Object File Format), Since this book is about *retro* computing, though, more typical output formats would be the DOS or CP/M COM file, the DOS EXE file, the REL (relocatable object) file, the OMF (Object Module Format) file (also known simply as an OBJ file), or the loadable HEX file.

While the term “compilation” is not necessarily limited to the transformation of a source program to an object program, as illustrated in figure 1, the discussion in this book will only cover this case.

Executable code is also called *object code*. Many compilers generate linkable object code instead of executable object code. In this case an additional program, a linker or linkage editor, is required to construct an executable program.

This is mostly done in order to support concepts such as separate compilation or add-on libraries. In separate compilation, chunks of a large program are compiled separately and then glued together by the linker.

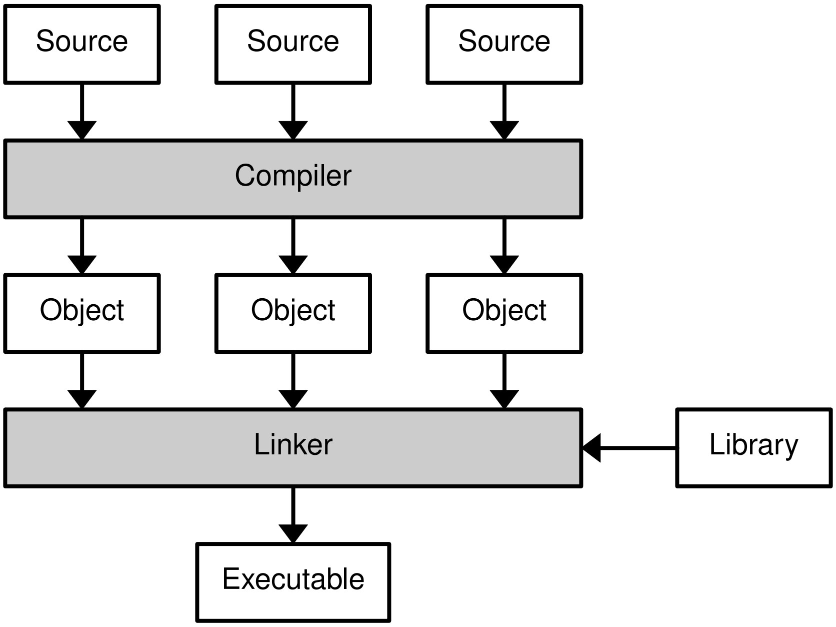

Figure 2: Separate Compilation with Linking

Run time libraries are a part of most compiler infrastructures. They provide pre-defined functions that can be used in a source program. Separate libraries are very common, but in some simple cases, the compiler may generate the collection of pre-defined functions directly instead of referring to an external library. In this case this collection may still be referred to as a “library”, though, as will be done in this book. Figure 2 summarizes the process of separate compilation, libraries, and linking.

The compiler described in this text uses the simplified model introduced initially. It reads a single source program and translates it directly to an executable file. All pre-defined functions are generated by the compiler, there is no linker and no support for additional run time libraries.

The Language

The source language *and* implementation language used in this book is a subset of an obscure, little, procedural language called T3X. It is a tiny language that once had a tiny community, and it was even used to write some real-life software, like its own compiler and linker, an integrated development environment, a database system, and a few simple games. It was also subject of a few college courses, most probably because (1) it was reasonably well defined and documented and (2) due to the size of its community, nobody could be bothered to do your homework assignments for you.

T3X looks like a mixture of Pascal, C, and BCPL. It has untyped data and typed operators, which simplifies the compiler *a lot*, but also leaves all the type checking to the programmer, which requires some discipline on the side of the user — a perspective that is not in vogue these days, where computing seems to be only about productivity, safety, and security.

But this text is not about creating a product and making a shiny web page about it. This is about diving right into the depths of the matter and having some *fun*. And a fun language T3X is. It is interesting to see how little you need to be able to write quite comprehensible and expressive programs.

Syntax

Syntax is what a language looks like. T3X is a *block-structured*, *procedural* language, which means that its programs describe *procedures* for manipulating data, i.e. “what to do with data”. It is called *block-structured*, because it is *structured* language that divides programs into blocks. A block is a chunk of source code that describes a part of a procedure.

A *structured* language uses certain constructs to describe the flow of control while a program executes, typically *selection* and *loops* (repetition).

Source code of procedural languages is mostly organized in the form a hierarchy consisting of a programs, declarations (including procedures and functions), statements, and expressions. The most abstract view is the program, the least abstract one the expression. See figure 3 for an illustration.

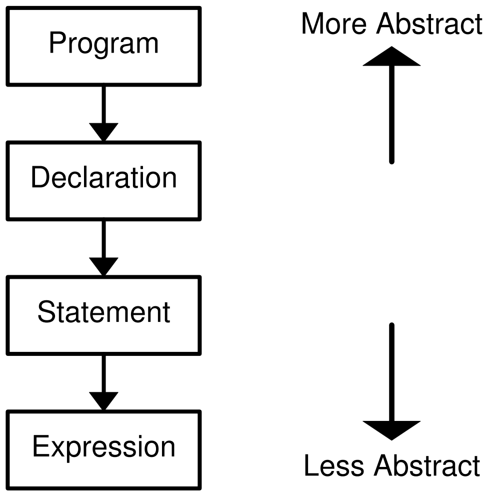

Figure 3: Elements of Block-Structured Languages

In procedural languages:

- programs contain declarations, statements, and expressions
- declarations contain statements and expressions
- statements contain expressions

If you are familiar with C or Pascal or BCPL, the T3X syntax will look quite familiar. Exhibit 4 displays the infamous bubblesort algorithm in T3X.

The keywords are highlighted by using upper case in this example, but this is not necessary and not usually done in actual code.

`bubblesort(n,v)` starts the declaration of the procedure *bubblesort* with the formal arguments *n* and *v*. The *body* of the procedure is a block statement (or *compound statement*) enclosed in the keywords **`DO`** and **`END`**. The compound statement declares the local variables *i*, *swapped*, and *tmp*.

Assignment is done by the **:=** operator (and equality is expressed with **=**). The statement **FOR (i=0, n-1)** counts from 0 to $n-2$. The i^{th} element of a *vector* (or array) v is accessed using **v[i]**. Elements are numbered $v[0] \cdots v[n-1]$.

```
! This is a comment
bubblesort(n, v) DO VAR i, swapped, tmp;
    swapped := %1;
    WHILE (swapped) DO
        swapped := 0;
        FOR (i=0, n-1) DO
            IF (v[i] > v[i+1]) DO
                tmp := v[i];
                v[i] := v[i+1];
                v[i+1] := tmp;
                swapped := %1;
            END
        END
    END
END
```

Figure 4: Bubblesort Procedure in T3X

The lexeme **%1** denotes the number -1. You could also write **-1**, but there is a subtle difference: the former is a value and the latter is an operator applied to a value, which will not work in contexts where a constant is expected.

Furthermore, **/\** and **\\/** denote logical (short-circuit) AND and OR, and **X->Y:Z** means “if x then y else z”, just like **x?y:z** in C.

IF with an **ELSE** is called **IE** (If/Else).

You will pick up the rest of the T3X syntax as we walk through the compiler source code. If you are interested, there is a brief introduction to T3X in the appendix.

Semantics

Semantics is how the syntax is interpreted. Note that “interpreted” does not imply the use of interpreting software here. Interpretation can be done at various levels, and in the case of the T3X compiler presented here, the code will eventually be interpreted by a Z80 CPU.

Interpretation in this case is a question of meaning. What does a statement like

```
WHILE (swapped) DO ... END
```

mean? To you, it is probably obvious that it means: “while the value of *swapped* is a ‘true’ value, repeat everything between `DO` and `END`”.

But now we need to know what a “true” value is and what “repetition” means. This is what the semantics of a language describes.

For example, the expressions `v[i]` and `s::i` both denote the i^{th} element of a vector. However, the first variant describes the i^{th} machine word in a vector of machine words, and the second one describes the i^{th} *byte* in a *byte vector*.

In this book, semantics will be specified in three different ways:

- by diagrams describing program flow;
- by short machine code sequences that resemble the meaning of a language construct;
- by simple mathematical formulae.

For instance, the meaning of the `[]` operator in the expression `v[i]` would be specified as follows, assuming that the value of i is stored in the *hl* register and the address of v is on top of the stack.

```
add hl,hl      hl = 2 · i
pop de         de = v
add hl,de      hl = address of v[i]
ld  a,(hl)     hl = (hl)
inc hl
```

```
ld  h,(hl)
ld  l,a
```

The exact semantics of the T3X language will be explained in the subsequent section and during the tour through the compiler source code.

Compiling the Compiler

Because the T3X/0 compiler is *self-hosting* (i.e. it is written in its own source language), it can compile its own source code. However, when a new language is created, how do we get the process started *without* an existing compiler?

This problem is widely known as the *bootstrapping* problem, because it seems to be as hard as pulling oneself out of a swamp by one's own bootstraps (bootstraps being the small loops that are attached to some boots to facilitate putting them on.)

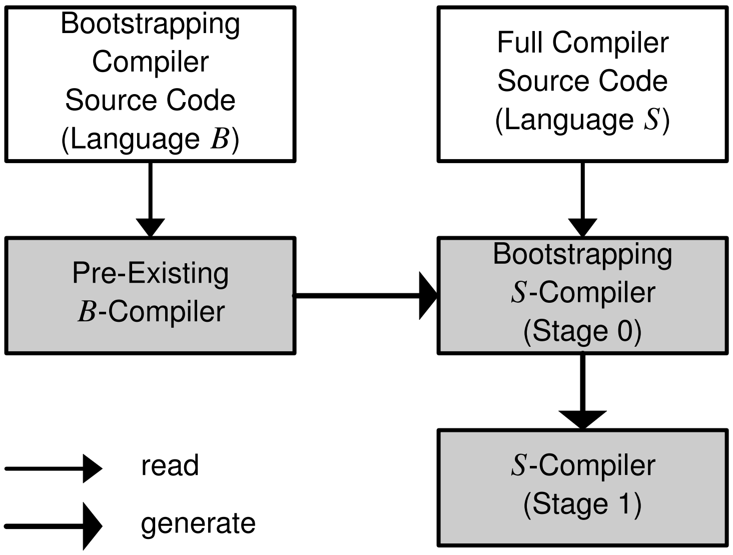
Bootstrapping Compiler Source Code (Language B)
Full Compiler Source Code (Language S)
Pre-Existing B-Compiler
Bootstrapping S-Compiler (Stage 0)
S-Compiler (Stage 1)
read
generate

Figure 9: Bootstrapping (gray boxes indicate binaries)

The process of bootstrapping is illustrated in figure 9. The goal of the process is to create a compiler for language S without access to a compiler for language S. The most common approach is to write a *bootstrapping compiler* for S in a different, pre-existing language B, and use that compiler to create the stage-0 compiler for S.

The *stage-0 compiler* is the result of bootstrapping a language S. The bootstrapping language which the stage-0 compiler accepts is often a subset S_0 of S that is exactly sufficient to compile the initial full S compiler.

Even if the stage-0 compiler covers the complete language S, it is often very simple. For instance, it may not perform any optimizations, have rudimentary error reporting, or have limited performance. It may even be implemented as a simple interpreter, because it only has to be run for one single time during the entire bootstrapping process.

Once the stage-0 compiler exists, it can immediately be used to compile the full compiler source code again, resulting in the *stage-1 compiler*. If the stage-0 compiler implements a subset S_0, this will be the first compiler implementing the full source language S. At this point, the bootstrapping problem is solved.

Bootstrapping the T3X/0 Compiler

The T3X/0 compiler package can be found at **`http://t3x.org`**. It provides two ways to solve the bootstrapping problem by including the following files:

- a bootstrapping compiler written in T3Xr7;
- a pre-compiled TCVM binary of the compiler and the C source code for the TCVM.

After compiling the TCVM, which consists of about 300 lines of portable C89 code, the TCVM binary of the compiler can immediately be used to re-compile the compiler.

Otherwise, a full T3X Release 7 compiler has to be installed first, which can then be used to bootstrap T3X/0. This method was used to bootstrap the initial stage-0 binary of the T3X/0 compiler.

Testing the Compiler

A simple method for verifying that a self-hosting compiler is performing properly is the so-called *triple test*. This test re-compiles the compiler with the stage-1 compiler, resulting in a stage-2 compiler, and then re-iterates the process to generate a stage-3 compiler. The process is depicted in figure 10. It is called the “triple test”, because it compiles the full compiler three times.

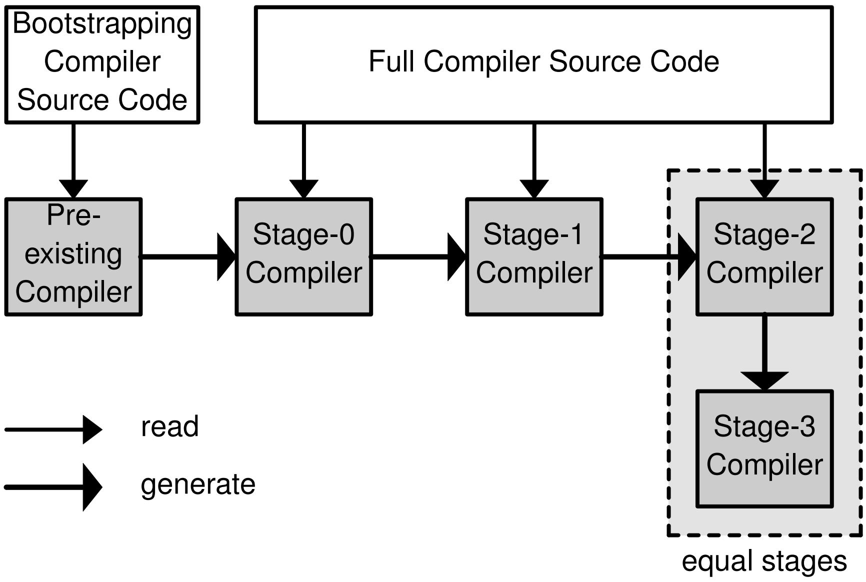

Figure 10: Triple Test (light gray boxes indicate binaries)

As can be seen in the figure, the stage-2 compiler is the first compiler that satisfies the following two conditions:

- it is a full compiler, i.e. it has been compiled from the full compiler sources, so its source language is the full language S and not a subset language S_0;
- it has been compiled by a full compiler.

Stage 0 is the bootstrapping compiler, which is generated from a completely different source code and probably only accepts a subset language S_0. Stage 1 is a full compiler, but has not been compiled by a full compiler, Stage 2 is the first entirely self-compiled compiler.

It can easily be seen in the diagram that the same conditions hold for the stage-3 compiler. They would also hold for any subsequent stages, so starting at stages 2 and 3, we expect the compilers to be identical, because they have been generated by identical compilers. If this is the case, the triple test has been passed.

Note that the triple test is only a simple test and probably does not cover the entire compiler source code. Passing the triple test is an essential step in the testing of the compiler, though. When this test fails, the compiler cannot be assumed to generate correct code.

Some Bootstrapping Sessions

This section describes the process of bootstrapping the T3X/0 compiler with Unix and DOS as host systems and CP/M as the target. It serves both as an example as well as instructions for getting the compiler up and running on CP/M.

Building T3X/0 on Unix the Lazy Way

Note that what is described in this subsection is not really a bootstrapping process, but merely the process of building the T3X/0 compiler with a pre-existing T3X/0 compiler. The existing compiler has been created using the process described in a later subsection, which is currently the only way to bootstrap T3X/0 “from nothing”, i.e. without a pre-existing T3X/0 compiler.

However, the lazy way is sufficient for those who merely want to have a T3X/0 compiler on CP/M (or a cross-compiler targeting CP/M on Unix) for further experiments.

Even if you decide to go the lazy route, it is still recommended to read the following subsections, because they explain a lot of details of the bootstrapping process, the components of the T3X/0 compiler, and its installation on CP/M.

The process of generating compilers for various host and target platforms is automated in the **bin/build.sh** script, which is contained in the T3X/0 distribution. Using this script, the cross-compilation of the compiler for CP/M is as simple as unpacking the T3X/0 source code archive and typing

```
make && bin/build.sh cpm cpm
```

This command will build

- the TCVM compiler emitting Tcode executables (**txtrn.tc**);
- the Tcode Virtual Machine (**tcvm**);
- a native T3X/0 compiler for CP/M (**tx-cpm.com**).

CP/M Example Program

Some functions of CP/M are not available in T3X/0, and the BDOS interface of CP/M has to be used directly in order to access them. For example, finding the names of all files on a specific drive is such a function.

The program shown in this chapter implements an extended CP/M "dir" command that sorts the files found on a drive alphabetically and also displays their sizes.

The program uses the **CPM** module, which defines the FCB ("file control block") structure, provides access to the internal file name conversion function **expandfn**, and also offers mnemonic names for all BDOS services.

```
! T3X/0 module: CP/M 2.2 BDOS Functions
! Nils M Holm, 2023
! In the public domain / 0BSD License

module cpm;
```

The FCB structure should be an old acquaintance at this point. Note that it is a byte-field structure, so the proper way to define an FCB is

```
VAR Fcb::CPM.FCB;
```

and its fields must be accessed with the byte-subscript operator **x::y**.

```
 public const   FCB         = 36,
                FCB_DISK    =  0,
                FCB_NAME    =  1,
                FCB_TYPE    =  9,
                FCB_ROBIT   =  9,
                FCB_SYSBIT  = 10,
                FCB_CHANGED = 11,
                FCB_EXTENT  = 12,
                FCB_RECORDS = 15,
                FCB_BLOCKS  = 16,
                FCB_SEQREC  = 32,
                FCB_RANRECL = 33,
                FCB_RANRECH = 34,
                RCB_RANOVFL = 35;
```

The **`cpm.expandfn`** inline function jumps to the internal **`t_expandfn`** function of the CP/M version of the T3X/0 core module via its jump table entry.

```
                           ! jp 0x014d
public inline expandfn(2) = [ 0xc3, 0x4d, 0x01 ];
```

The following functions provide mnemonic names for the CP/M 2.2 BDOS functions. For example,

`cpm.prints("Hello, World\r\n$")`

would print the well-known message on the CP/M console via the BDOS function **`BDPRINTS`**.

```
public sysreset()      return t3x.bdos(0, 0);
public conin()         return t3x.bdos(1, 0);
public conout(c)       return t3x.bdos(2, c);
public readin()        return t3x.bdos(3, 0);
public punout(c)       return t3x.bdos(4, c);
public lstout(c)       return t3x.bdos(5, c);
public conio(c)        return t3x.bdos(6, c);
public getiob()        return t3x.bdos(7, 0);
public setiob(iob)     return t3x.bdos(8, iob);
public prints(s)       return t3x.bdos(9, s);
public readcons(buf)   return t3x.bdos(10, buf);
public constat()       return t3x.bdos(11, 0);
public getver()        return t3x.bdos(12, 0);
public dskreset()      return t3x.bdos(13, 0);
public seldsk(dsk)     return t3x.bdos(14, dsk);
public open(fcb_)      return t3x.bdos(15, fcb_);
public close(fcb_)     return t3x.bdos(16, fcb_);
public search(fcb_)    return t3x.bdos(17, fcb_);
```

```
 public searchnext()    return t3x.bdos(18, 0);
 public erase(fcb_)     return t3x.bdos(19, fcb_);
 public readseq(fcb_)   return t3x.bdos(20, fcb_);
 public writeseq(fcb_)  return t3x.bdos(21, fcb_);
 public create(fcb_)    return t3x.bdos(22, fcb_);
 public rename(fcb_)    return t3x.bdos(23, fcb_);
 public getlogvec()     return t3x.bdoshl(24, 0);
 public getcurdsk()     return t3x.bdos(25, 0);
 public setdma(dma)     return t3x.bdos(26, dma);
 public getalvec()      return t3x.bdoshl(27, 0);
 public setdskro()      return t3x.bdos(28, 0);
 public getrodsks()     return t3x.bdoshl(29, 0);
 public setfat(fcb_)    return t3x.bdos(30, fcb_);
 public getdpb()        return t3x.bdoshl(31, 0);
 public getsetusr(n)    return t3x.bdos(32, n);
 public readran(fcb_)   return t3x.bdos(33, fcb_);
 public writeran(fcb_)  return t3x.bdos(34, fcb_);
 public getfsiz(fcb_)   return t3x.bdos(35, fcb_);
 public setranrec(fcb_) return t3x.bdos(36, fcb_);
 public resetdsks(map)  return t3x.bdos(37, map);
 public writeranz(fcb_) return t3x.bdos(40, fcb_);

end
```

Here follows the code of the “cpmdir” program, which can be found in the file “programs/cpmdir.t” in the T3X/0 source code archive.

```
! List a CP/M directory with file sizes
! Nils M Holm, 2023
! Public Domain / 0BSD License

use t3x: t;
use cpm;
```

The maximum number of directory entries in a standard CP/M floppy disk directory. Increase this for use on hard disks.

```
const   MAXFILES = 64;
```

FCB and DMA buffer for file I/O.

```
var Fcb::CPM.FCB;
var Buf::128;
```

A buffer for file names and a vector of pointers to the file names in the buffer. Each file name in an FCB has a length of 11 characters.

```
var Files::MAXFILES*11;
var Ptrs[MAXFILES];
```

Compute the length of a string.

```
length(s) return t.memscan(s, 0, 32767);
```

Write a NUL-terminated string to the console.

```
writes(s) t.write(T3X.SYSOUT, s, length(s));
```

Write a newline sequence to the console.

```
nl() do var b::3;
    writes(t.newline(b));
end
```

Convert an unsigned number to ASCII and return a pointer to the ASCII representation.

```
var ntoa_buf::10;
ntoa(x) do var i;
    if (x = 0) return "0";
    ntoa_buf::9 := 0;
    i := 9;
    while (x) do
        i := i-1;
        ntoa_buf::i := x mod 10 + '0';
        x := x / 10;
    end
    return @ntoa_buf::i;
end
```

Sort the pointers to the files names in **Ptrs** so that they point to names in lexicographic order. The **sortdir** function uses a slightly optimized Bubblesort algorithm, which should be good enough for sorting rather small directories.

```
sortdir(k) do var i, j, tmp, sw;
    for (i=1, k) do
        sw := 0;
        for (j=0, k-i) do
            if (t.memcomp(Ptrs[j],
                          Ptrs[j+1], 11) > 0)
            do
                tmp := Ptrs[j];
                Ptrs[j] := Ptrs[j+1];
                Ptrs[j+1] := tmp;
                sw := 1;
            end
        end
        if (\sw) leave;
    end
end
```

Read all file names on the given disk and store them in the **File** buffer and pointers to the names in the **Ptrs** vector. When a drive letter is given insert it into the match string, otherwise use a match string without a drive letter. In a *file match* every question mark will match any character in a file name, so a file match of the form **????????.???** will match any file name. Note that the dot has a special meaning in **cpm.expandfn** and must not be replaced with a question mark.

The **cpm.search** and **cpm.searchnext** functions both return a *directory code*, which is an index into the DMA buffer. The search functions read the directory one record at a time, and four directory entries fit in a 128-byte record, so the functions return a value in the range $0..3$ upon success. To locate the directory entry in the buffer, the return value has to be multiplied by 32. The layout of the name in the directory entry is the same as in an FCB.

The **readdir** function returns the number of file names stored in the **Files** buffer.

```
readdir(d) do var k, n, match;
    match := "A:????????.???";
    ie (d)
        match::0 := d;
    else
        match := @match::2;
    cpm.expandfn(match, Fcb);
    cpm.setdma(Buf);
    k := 0;
    n := cpm.search(Fcb);
    while (k < MAXFILES /\ n \= 255) do
        Ptrs[k] := @Files::(k*11);
        t.memcopy(Ptrs[k], @Buf::(1+n*32), 11);
        k := k+1;
        n := cpm.searchnext();
    end
    return k;
end
```

Return the size of the i^{th} file in the buffer in kilobytes. The **cpm.getfsiz** function returns the size of a file in records in the random record fields of the FCB passed to it.

```
filesize(i) do
    t.memcopy(@Fcb::CPM.FCB_NAME, Ptrs[i], 11);
    cpm.getfsiz(Fcb);
    ! ignoring FCB_RANOVF
    return (Fcb::CPM.FCB_RANRECL +
            Fcb::CPM.FCB_RANRECH * 256 + 7) / 8;
end
```

Print the size in kilobytes and the name of a file on the console.

```
printfile(i) do var n, j, s;
    n := filesize(i);
    s := ntoa(n);
    for (j = length(s), 4) writes("\s");
    writes(s);
    writes("K\s\s");
    t.write(T3X.SYSOUT, Ptrs[i], 8);
    writes("\s");
    t.write(T3X.SYSOUT, Ptrs[i]+8, 3);
end
```

Print all files in the directory in three sorted columns, separated by colon characters.

```
const COLS = 3;

printdir(k) do var i, j, m, n;
    ie (k mod COLS)
        n := k / COLS + 1;
    else
        n := k / COLS;
    i := 0;
    for (i=0, n) do
        m := i;
        for (j=0, COLS) do
            if (m < k) do
                printfile(m);
                if (m+n < k) writes("  : ");
            end
            m := m + n;
        end
        nl();
    end
end
```

Return the given character converted to upper case.

```
upcase(c) return 'a' <= c /\ c <= 'z'-> c-32: c;
```

The main program accepts one command line argument, extracts only the first character from it, and uses it as a drive letter. This means that the commands

```
B>cpmdir a
B>cpmdir a:
B>cpmdir abracadabra
```

would all list the files on drive `A:`. When no drive letter is given, the default drive is used. The program then reads the file names on that drive into a buffer, sorts them, and prints them.

```
do var d, k, b::2;
    d := 0;
    if (t.getarg(1, b, 2) >= 0)
        d := upcase(b::0);
    k := readdir(d);
    sortdir(k);
    printdir(k);
end
```

This is what the output of CPMDIR looks like:

```
 17K  CG        T    :    2K  T3X       T    :   34K  TXTRN     T
  6K  CPMDIR    COM  :   34K  TX0       COM
  3K  CPMDIR    T    :    2K  TXEMIT    T
```

Some Random Facts

The source code of the T3X/0 compiler for CP/M on the Z80 has a total size of about 2330 lines or 50K bytes. The parser is the largest part by far, which is typical for a simple compiler without sophisticated code generation. The sizes of the individual components are listed in figure 81, but, since the code of each component is spread out over the entire file, these sizes are not very accurate.

Component	**Size (Lines)**
Symbol Table Management	100
Emitter	200
Scanner	290
Parser	1080
• Declarations	360
• Statements	250
• Expressions	400
Code Generator	190
Runtime Library Dump	250
Core Module	40
Miscellanea	180
Total	2330

Figure 81: Sizes of the Components of the Compiler

The runtime library dump alone has a size of about 10K bytes. The Z80 assembly language code from which the run time library is compiled has a size of 1200 lines and compiles to a binary of 1670 bytes.

The stage-3 executable of the compiler has a size of 34834 bytes and self-compiles in less than 10 minutes on an 4MHz Z80 system with its file system on an SRAM card. On a floppy-based system, the compile time could be expected to be much longer, of course.

Figures 82 and 83 list some more compiler executable sizes and self-compilation times for various systems. The sizes depend much on the code density of the compiler output. Of course the TCVM has the best density, because it has been designed for the T3X/0 compiler. The Z80 back end typically emits more native instructions per VSM instruction than the 8086 and 386 back ends.

Target	**Word Size**	**Size (Bytes)**
TCVM	16-bit	21003
CP/M Z80	16-bit	34834
DOS 8086	16-bit	29181
FreeBSD 386	32-bit	39140

Figure 82: Compiler Executable Sizes

System	**Speed (MHz)**	**File System**	**Time**	**LPM**	**LPM/MHz**
CP/M Z80	4.00	SRAM[1]	9m20s	251	63
DOS V20[2]	4.77	CF[3]	8m31s	274	58
CP/M Z80	7.40	CF	5m37s	434	59
FreeBSD TCVM[4]	1000.00	SSD[5]	0.80s	164,000	164
FreeBSD x86-64	1000.00	SSD	0.15s	933,000	933

Figure 83: Self-Compilation Speed, LPM = lines/minute

1 Static RAM card
2 8088-compatible
3 Compact Flash card
4 TCVM running on an x86-64
5 Solid State Disk