Acta Iccis

The ICCE Proceedings

Icmake

Frank B. Brokken and Karel Kubat

1993-9

Acta Iccis is published by the ICCE (Interdisciplinary Centre for the development of Computer Coaches and Expert systems).

All correspondence regarding copyrights, reprints or any other matter concerning this publication can be directed to:

ICCE
P.O. Box 335
9700 AH Groningen
The Netherlands
phone: +31 50 63 36 36

Contents

ICMAKE

The **IC**CE **MAKE** Utility
or:
the **I**mproved **C**-like **MAKE** Utility
Version 6.00

Frank B. Brokken and K. Kubat

1 Introduction to ICMAKE

Welcome to *icmake*! *Icmake* is a group of programs which constitute an automatic program maintenance utility, comparable to the *unix*-program *make*. However, *icmake* is, in our opinion, more 'manageable' than *make*: dependencies between files and actions to perform can be stated in way which is easily understood and quite 'natural' to **C** programmers. Besides being a tool for program maintentance, *icmake* has grown into a useful shell script language. System administrative tasks, such as the starting of backups, can be accomplished by *icmake* just as with an other language; but in our **C**-oriented opinion more easily.

This document is the User Guide to *icmake*. The programs constituting *icmake*, the syntax of a makefile and several examples are discussed. It is assumed that the reader is familiar with the concepts of a make utility, with the concept of dependencies between files and with the concept of automatic program maintenance.

Furthermore, it is assumed that the reader is familiar with concepts of programming (such as variable definitions, functions, arguments and return values) and with the syntax of the **C** programming language. This document does not study these themes in depth, but rather points out differences between **C** and *icmake*.

Icmake was developed as our answer to problems experienced with the make utility available in UNIX and MS-DOS operating system environments. Initially (that is, after the one and a half week it took us to develop *icmake*) the programs were available on MS-DOS systems, since the programs were developed there. The *icmake* programs were then successfully ported to UNIX platforms, such as, LINUX and SCO-UNIX. The *icmake* programs furthermore also are available for HP-UX.

1.1 Legal considerations

Icmake is shareware. This means that no fee is charged for it. As with everything that's free, there's no pay but also **absolutely no warranty**. Furthermore, you are allowed (and encouraged) to distribute *icmake*, provided that you include this information with each distribution and provided that you do **not** charge any amount for it. Not even the 5 dollars for shipping.

The source files and the documentation for *icmake* are copyrighted by us. The reason for this is (a) that we'd like to have always the last version of Icmake, and (b) that we'd like to have the last word in all modifications. If you have requests (or even better, "working code" to include in *icmake*) please mail us and we'll gladly oblige when we find the time.

1.2 Obtaining ICMAKE

The *icmake* package can be obtained by anonymous ftp from the site *beatrix.icce.rug.nl*, directory *pub/unix*. The package comes as an archive, usually in the form *icmake-X.XX.tar.gz*. This archive contains a tar'd and gzip'd directory structure, in which the source files for *icmake* and the executable programs for MS-DOS can be found. The same archive site also contains the file *icmake.doc*, which is a guide to the installation of *icmake*. This file is especially useful for UNIX installations.

The *icmake* package can also be found at the site *tsx-11.mit.edu* as part of the Linux Operating System. The files are usually located in *pub/linux/sources/usr.bin*.

2 The programs of ICMAKE

Icmake consists of five programs which may be necessary in the process of reading a makefile and executing the commands specified in this file. The programs can be divided in two categories: programs of which the casual user of *icmake* must be aware (the top-level programs) and programs which are called internally by *icmake*.

Each of the programs of the *icmake* family has its version number. The version numbers consist of a major and a minor number; e.g., in the version number 5.03, the major version is 5 and the minor version is 03. The programs can only communicate when the major version numbers of all *icmake* programs are equal: i.e., a working set of programs must have the same major version number. The minor number is used to indicate small changes in the separate programs.

2.1 The top-level programs

Two programs of the *icmake* group form the interface to the user: *icmake* itself and *icmun*. The 'dependent' programs are generally called by the top-level programs as child-processes or overlay-processes. However, every dependent program is a complete program by itself, and could as well be activated by users of *icmake*.

The ICMAKE program

The main program is *icmake*[1]. The *icmake* program acts as an interface between the user and other programs of the *icmake* group. Normally, the user will have no need to know the exact names or functions of the other *icmake* programs; only the 'top level' program will have to be activated to start the process of making.

The *icmake* program is normally invoked by one of the two following command lines:

icmake [flags] asciifile [binaryfile] [-- arguments]
icmake [flags] -i asciifile [arguments]

The two invocation modes differ in the fact that the first invocation, without the *-i* flag, allows the user to specify the name of a binary file. This file is an intermediate file in the making process. The second invocation allows *icmake* to choose a name for the binary file. When not specified, *icmake* uses the name of the asciifile but with the extension *.bim*.

In both command lines, the *flags,* or the two consecutive hyphens and the arguments are optional. The arguments are passed to the makefile and can be inspected there. The flags are used for requesting specific actions of *icmake*. The specification of the asciifile is in both invocations obligatory: this file is the make script, which is interpreted by *icmake* and according to which specific actions are taken.

When *icmake* is activated without any arguments, a help summary is displayed, showing among other things the flags which are recog-

[1] Under MS-DOS, all programs of the *icmake* group have extension *exe*. In this document the extension of executables is left out, since other operating systems than DOS may require other extensions or may require no extensions at all.

nized by *icmake*. When *icmake* is started without any arguments, something like the following output appears on the screen[2]:

```
ICCE Make Utility Version 6.00
Copyright (c) ICCE 1992, 1993.  All rights reserved.

Usage: ICMAKE [flags] source[.im] [dest[.bim]] [-- [args]]
where:
        flags:  optional flags:
                -b      : blunt execution of the destinationfile
                -c      : the destination file is compiled
                -o file: all icmake output is redirected to 'file'
                -i file: define input file, argument processing stops
                -p      : only the preprocessor is activated
                -q      : quiet mode: copyright banner not displayed
        source: make description source file
                (default extension: .im)
        dest:   binary make file
                (default filename:  source.bim)
        -- :   optional icmake-file arguments separator
        args:  optional arguments following -- received by
               the icmake file in its argv-list
```

From this help-summary it is seen that the following flags are recognized:

-b. When this flag is used, no *recency* test is taken. Rather, the binary make file is executed immediately 'as is'. When this flag is absent, *icmake* compiles the input file (*.im* file) into a binary file (*.bim* file) if the input file is more recent.

-c. When this flag is used, the ascii input file is compiled to a binary makefile, but not automatically executed. The compilation occurs irrespective to the recency of the input file and the binary file.

-i file. This option specifies *file* as the input file for *icmake* and stops the further processing of the arguments. The name of the binary makefile (*.bim* file) is then the name of the input file but with the extension *.bim*. The default input file extension, *.im*, is **not** supplied by *icmake* when this flag is given: the specified name is taken literally.

[2] The actual summary may vary with different versions of *icmake*.

-o file. With this option, an extra filename is required. When used, all *icmake* output is redirected to 'file', and will not appear on the screen[3]. This flag is only available in MS-DOS versions of *icmake*: under DOS, an 'ordinary' redirection statement on the command line fails to redirect child programs and therefore a separate redirection mechanism, activated by this flag, is needed. Under UNIX, redirection of *icmake* and of its child programs can be achieved using the > sign on the command line.

-p. When this option is used, only the *Icmake preprocessor* is activated. The output of the preprocessor is sent to a file having the same name as the input file, but extension *.pim*. When this flag is absent, *icmake*'s preprocessor still generates a *.pim* file; but this file is deleted when no longer required.

-q. When this option is used, *Icmake* operates *quiet*. I.e., the copyright banner is not displayed.

--. By default, the first non-flag argument on the command line is the input file, for which the extension *.im* is assumed. When a second non-flag argument is given, then this name is used for the binary file (*.bim* file). The processing of arguments stops only when two consecutive hyphens are encountered. All arguments which follow the -- flag can be inspected by the makefile.

Note that the -- flag is not required when *icmake* is activated with the *-i* flag present.

The *asciifile* specification is obligatory. This is the ascii makefile which will be compiled and tested by *icmake*. *Icmake* assumes a default extension *.im*. E.g., the command line *icmake test* will activate *icmake* to process the ASCII makefile *test.im*. Note however that the command line *icmake -i test* will start *icmake* to process the makefile *test*; the presence of the *-i* flag forces *icmake* not to supply an extension.

[3] The ICCE program *tee* is another useful tool here, allowing the display of output to the screen, and the *simultaneous* generation of a copy of the displayed information to a file.

The *binaryfile* specification is optional. When given, *icmake* uses this file as the binary intermediate file in the process of making. Extension *.bim* is the default. When not given, *icmake* uses the basename of the *asciifile* but with extension *.bim*.

Following the *binaryfile* specification, several arguments to the makefile itself may be given. However, before any extra arguments are specified two consecutive hyphens are needed. Following these hyphens extra arguments may follow which will be passed to the *icmake* dependent programs. As described above, the delimiting two hyphens are not necessary when the *-i* flag is used.

The *icmake* specification file is written as a **C** program, and contains a *main()* function which receives some of the arguments specified on the command line. The first argument is always the name of the binairy make file (normally having the *.bim* extension). Remaining arguments are the arguments that follow the two hyphens. The hyphens themselves are *not* included in the series of arguments which are passed to *main()*) (See also the paragraph below about *the user-defined function main()*).

The ICMUN program

The *icmun* program is mainly used in developing *icmake*. The *icmun* program is an unassembler for the binary make file created by *icmake*. *Icmun* expects a binary makefile as argument and produces an assembly-like listing of the instructions contained in the binary makefile. The remainder of this section can be skipped without loss of continuity. It is mainly included for those who are interested in the way the binary makefile is organized. Further information on this subject can be obtained from the authors of *icmake*

An example of a make-sourcefile is:

```
void main()
{
    printf("Hello world\n");
}
```

A hexdump of the binary makefile which is generated from this small *icmake* program is:

```
00000000:  32 2E 31 32 26 00 00 00 33 00 00 00 33 00 00 00  2.12&...3...3...
00000010:  22 00 00 00 06 00 00 05 00 00 05 02 00 1B 0D 1C  "...............
00000020:  03 23 21 14 00 1D 48 65 6C 6C 6F 20 77 6F 72 6C  .#!...Hello worl
00000030:  64 0A 00 74 65 73 74 2E 69 6D 0A                 d..test.im.
```

Another way to look at the binary makefile is to use *icmun* to unassemble the binary file. When *icmun* is called to unassemble this binary file, the following output is obtained:

```
ICCE ICMAKE Binary Make File Unassembler  Version 2.01
Copyright (c) ICCE 1992, 1993. All rights reserved.

Binary file statistics:
        strings           at offset     0026
        variables         at offset     0033
        filenames         at offset     0033
        first instruction at offset     0022

String constants dump:
        "Hello world."

Disassembled code:
        [0014] 06 00 00   push string "Hello world."
        [0017] 05 00 00   push int 0000
        [001a] 05 02 00   push int 0002
        [001d] 1b 0d      callrss 13 (print)
        [001f] 1c 03      add sp, 3
        [0021] 23         ret
        [0022] 21 14 00   call [0014]
        [0025] 1d         exit
```

This final output clearly shows the organization of the binary makefile: The strings, variables (in this example there are none), filenames, and first instruction offset is clearly marked, the strings are shown, and the instructions are displayed in an assembly-like way. The disassembled code consists of lines, one line for each disassembled instruction. Within each line three parts are recognized. The leftmost part is between square brackets, showing the byte-offset of the first byte of the instruction. The middle part shows the bytes forming the instruction, and the rightmost part is a legible translation of the binary code. For example, starting at offset 1f[4] there are tow bytes,

[4] *Icmun* displays values in hexadecimal form

having values of respectively 1c and 03. These values are interpreted as the instruction to add a value 3 to *sp,* the stack pointer.

2.2 The 'dependent' programs, used internally

The following programs are the 'dependent' program of *icmake*. Normally these programs are not started by the user of *icmake,* but are (eventually) started as child- or overlay-processes of *icmake*.

The preprocessor ICM-PP

Icm-pp is the preprocessor of the *icmake* compiler. This program scans the user-supplied makefile for preprocessor directives (e.g., *#include*, see section 3.1) and takes appropriate actions. An output file is written in which the preprocessor directives are 'expanded'. This (temporary) file is used by the next stage of *icmake*.

The compiler ICM-COMP

The following stage of *icmake* is the compiler *icm-comp*. This program translates an ASCII file, generated by *icm-pp*, to a binary format and performs error checking. The resulting binary file contains *opcodes*, much like the output file of a compiler of a programming language. When a binary makefile is generated, the intermediate output file of the preprocessor is no longer needed and is deleted.

The executor ICM-EXEC

Icm-exec is the executor of the binary makefile generated by *icm-comp*. This stage reads the binary makefile and interprets the opcodes: lists of files are built, files are compared by dates, actions are taken.

2.3 The calling order of programs

The *icmake* group of programs attempts to make use of the available memory as efficient as possible. Memory problems may arise during the execution phase of a program; i.e., during the run of *icm-exec*.

The main program *icmake* spawns the preprocessor and the compiler as child processes. However, when the executor is called the *icmake* program itself is overlayed, thereby making the maximum amount of memory available. Due to this setup, *icm-exec* may spawn child processes which are specified by the user with a maximum memory pool available.

2.4 Executable makefiles under Unix

Under the operating system UNIX it is possible to create makefiles for *icmake* which are themselves executable. E.g., a setup is feasible in which a makefile *backup* exists, which can be started by the command *backup*. In such a setup the same effect may be achieved by creating a makefile *backup.im* and by starting it with the command line *icmake backup*.
To create an executable makefile, the following steps may be taken.

- The makefile can be renamed to a suitable name. This name is later used to invoke the process which executes the makefile; for this reason, the name *backup* may be preferable to *backup.im*.
- The makefile is made executable using the UNIX *chmod* program. E.g., the command

  chmod +x backup

  labels the file *backup* as an executable program.

  The makefile may furthermore be placed in a directory pointed to by the *PATH* environment. E.g., the makefile may be placed in a user's private *bin* directory.
- The following line is added as the first line in the makefile:

  #!/usr/bin/icmake -qi

  This line informs the operating system that when executing the file, the program */usr/bin/icmake* should be started, supplying the flags *-qi* and supplying the name of the makefile. The flag *-q* suppresses *icmake*'s copyright header; the flag *-i* causes *icmake* to take the following file argument as the literal input file instead of supplying the default extension *.im*. See also section 2.1 for a description of these flags.

Note that this line must contain a full reference, including the path, to the program *icmake*. In this example this path is */usr/bin*.

The binary makefile (*.bim* file) is placed by *icmake* in the same directory as the ascii makefile. This directory must therefore be accessible to the user invoking the makefile.

3 The syntax of the makefile

The user-supplied makefile must follow a well-defined syntax, which resembles the **C** programming language. This section describes the syntax of the makefile.

3.1 Comment and preprocessor directives

One of the tasks of the preprocessor is to strip the makefile of comment. *Icmake* recognizes two types of comment: the standard **C**-like comment and end-of-line comment which is implemented in, e.g., the Microsoft 6.00a **C** compiler.

Standard comment must be preceded by /* and must be closed by */. This type of comment may stretch over more than one line. End-of-line comment is preceded by // and ends when a new line starts.

Lines which start with *#!* are skipped by the preprocessor. This feature is included to allow executable makefiles under UNIX, see section 2.4.

Apart from the *#!* directive, *icmake* recognizes two more preprocessor directives: *#include* and *#define*. Both directives must be preceded by a '#'-sign which must be located at the first column of a line in the makefile.

The #include directive

The *#include* directive must obey the following syntax[5]:

 # include *"filename"*
 # include <*filename* >

When *icmake* (or, more exactly the preprocessor *icm-pp*) encounters this directive, the *filename* which is stated following the *#include* directive is read. The filename may include a path specification[6]. When the filename is enclosed by double quotes, *icmake* attempts to access this file exactly as stated. When the filename is enclosed by < and >, *icmake* attempts to access this file relative to the directory pointed to by the system variable *IM*. This setup is analogous to the processing of include-files by the Microsoft **C** compiler under MS-DOS.

The #define directive

The *#define* directive is a means of incorporating constants in a makefile. The directive follows the following syntax:

 # define *identifier redefinition-of-identifier*

The defined name (the name of the defined constant) must be an identifier according to the **C** programming language: the first character must be an underscore or a character of the alphabet, subsequent characters may be underscores or alphanumerics.

The redefinition part of the *#define* directive consists of spaces, numbers, or whatever is appropriate. The preprocessor simply replaces all occurrences of the defined constant following the *#define* directive by the redefinition part. Note that redefinitions are not further expanded; i.e., an already defined name which occurs in the redefinition part is not processed but is left as is.

Also note that *icmake* considers the redefinition part to be *all* characters found on a line beyond the defined constant. This would also

[5] Syntactical rules are represented in this document by two fonts: the boldface parts must appear exactly in the makefile. The slanted parts may be substituted by the user.

[6] Under MS-DOS, the path specification may include slashes as well as backslashes. When backslashes are used, only one backslash must be used to separate (sub-)directories.

include comment, if found on the line. Consequently, it is normally *not* a good idea to use comment-to-end-of-line on lines containing *#define* directives.

3.2 Types, constants and variables

Icmake recognizes four types: *int, string, list* and *void*. The types serve the following purposes:

- The type *void* is used only with functions, to indicate that these functions do not return values.

- Constants may be used in the makefile to indicate a number or a string. The constants then obviously have type *int* or type *string*. *Int* constants are denoted by numeric characters; e.g., *13* is an *int* constant. *String* constants are denoted by text between double quote marks; e.g., *"a string"* is a piece of text. List constants are not allowed in *icmake*: lists of files always have to be built run-time.

  A second way to denote an *int* constant is by enclosing a character in single quotes. The numeric value of the constant is then the ASCII number of the character; e.g., the constant *'A'* has the value 65. The character between quotes may not be 'escaped', such as, e.g., *'\n'*. Only single characters are allowed in this notation of integer constants.

- The three types further serve the definition of variables and arguments. *Icmake* allows global variables and local variables. The declaration of a variable or of an argument must state the type of the variable: e.g., *int* for a counter variable or *list* for a variable holding, e.g., the names of all files having extension '.c'.

- Some of the built-in functions of *icmake* (see section 3.6) return a value of one of the types *int*, *string* or *list*. The returned value may be assigned to a variable of the same type or may be passed to another function. Additionally, all built-in functions require arguments of a certain type.

- Similarly to built-in functions, user-defined functions are assumed to return a value which is either *int*, *string* or *list*. The *int* type is the default.

The definition of variables follows a **C**-like syntax. Arguments are defined as in ANSI-**C**.
Some examples of the usage of types are given in the listing below. Note also the use of the constants *55* and *"main.c"*.

```
string myfun (int x, string y, list z)  // a user-defined function
{                                       // of type string, having 3
    int                                 // parameters
        counter,                        // local variables: 2 ints,
        i;                              // 1 string and 1 list
    string
        name;
    list
        cfiles;

    counter = 55;                       // counter is set to 55
    name = "main.c";                    // name is set to string main.c
    .
    .
    .
    return (name);                      // a string is returned to the
}                                       // caller
```

3.3 Strings and escape sequences

Strings in makefiles are used to represent both filenames and displayed text. *Icmake* allows a number of special formatting sequences in strings to facilitate the display of text. These sequences are called, in analogy to the **C** programming language, escape sequences.

Escape sequences in strings are identified by a backslash character (\), followed by a character which identifies the escape sequence. The escape sequences recognized by *icmake* (or, more accurately by the compiler *icm-comp*) are summarized in figure 1. E.g., when a string *"Starting.\n"* is displayed, the string *Starting.* is printed and the cursor is set to the start of the following line.

As a side effect, the backslash character itself must be represented as '\\'. This may be needed when, e.g., strings represent pathnames

Escape sequence	Action
\a	alert (bell)
\b	backspace character
\f	formfeed character
\n	newline
\r	carriage return character
\t	tab
\v	vertical tab
other	literal *other*

Figure 1: Escape sequences

under MS-DOS. The **only** exception to this rule is the *#include* preprocessor directive (see section 3.1): this directive takes strings literally as they appear in a makefile.

Like **C**, *Icmake* allows *string-concatenation*. Long strings, extending over several lines of text, can be built by separating string constants by white-space characters (blanks, tabs, newlines).

3.4 The code of a makefile

This section discusses the user-defined functions which may appear in a makefile and lists the most common statements.

Flow control statements

Icmake recognizes six control statements:

- *if* statements, including *if-else,*
- *while* statements
- *for* statements
- *return* statements
- *break* statements
- *exit* statements

The *exit()* statement, though a function in **C**, is part of the *icmake* language. Following the *exit* keyword, an expression yielding an *int* may follow. If an *int* expression follows, its value is returned as an *int* to the operating system; otherwise, the returned value is undefined. The other flow control statements are analogous to the corresponding ones in the **C** programming language.

User-defined functions

Icmake allows the construction of user-defined functions in a makefile. The definition of a function must follow an ANSI-**C**-like syntax, however, slight differences exist between an *icmake* function and a **C** function. These differences are highlighted in this section.
The definition of a function must follow the syntax:

- Optionally the return type of the function is specified. The type is *void, int, string* or *list*. The default return type is *int*.

  Note that when a function explicitly returns using a *return* statement, the returned value must match the return type. A function returns an undefined value if it does not use a *return* statement. Functions which are defined as *void* can also use the *return* statement, albeit without an expression.

- Following the optional return type, the function name must follow. The name must be an identifier; i.e., the first character must be an underscore or a character of the alphabet and optional following characters may be underscores or alphanumerics.

- Following the function name, a (is expected.

- A parameter list may follow, consisting of parameter specifications separated by , (this is referred to as an ANSI-**C** parameter list). Parameter specifications consist of the parameter type (*int*, *string* or *list*) and the parameter name (an identifier). In contrast to **C**, *icmake* does not allow functions having a variable number of parameters.

- Following the optional parameter list a) is expected.

- Next, the code of the function is expected: statements enclosed by { and }.

- Following the first { of the code block, local variables may be defined. Note that *icmake* allows the definition of local variables only after the outer curly brace of the function code block; the definition within any block of statements is not allowed (in contrast to **C**).

  The definition of local variables consists of the variable type, one or more variable names separated by commas and a semicolon. In contrast to **C**, *icmake* initializes all local variables to zero.

Icmake does not allow forward references. This means that a function may only be called *after* it has been defined. Furthermore, the statement which calls a function must supply the exact number of required arguments and each argument type must match the parameter list of the function. The built-in functions are predefined and may therefore be used anywhere within functions.

The user-defined function main()

The code section of a makefile must contain at least one user-defined function, called *main()*. The execution of a makefile starts at this function. The run-time support system of *icmake* (or, more exactly, the executor *icm-exec*) provides three arguments which the function *main()* may use. The arguments are used to hold the command line parameters of the *icmake* invocation and the environment setting.

The three arguments are most commonly referred to as *argc*, *argv* and *envp*. *Argc* is an *int* argument, holding the number of command line parameters. *Argv* is a *list*, holding the command line parameters themselves. *Envp* is a list holding the environment setting. A definition of the *main()* function which uses all arguments *argc*, *argv* and *envp* is given below:

int main (int argc, list argv, list envp)

Users may wish to define the *main()* function without arguments, when the command line parameters need not be examined. In this case, the *main()* function can be defined as:

int main ()

Also it is possible to define the *main()* function to use only the first two arguments *argc* and *argv*:

int main (int argc, list argv)

Following the same reasoning, it is possible to define a *main()* function using only its first argument *argc*. However, this does not seem very useful, since such a *main()* function could investigate the number of command line arguments but would not be able to investigate the arguments themselves.

A sample makefile which prints its command line arguments is given below[7]:

```
void main (int argc, list argv)
{
    int
        i;

    for (i = 0; i < argc; i++)
        printf ("Argument ", i, " is ", element (i, argv), "\n");
}
```

The arguments passed to the *main()* function of *icmake* as the list *argv* are:

- The first argument is always the name of the binary makefile which is interpreted by *icm-exec*.

- Remaining arguments are *only* those arguments which are explicitely supplied on the command line.

E.g., to supply the arguments *one*, *two* and *three* to a makefile called *test.im*, one of the following invocations can be used:

icmake test -- one two three
icmake -i test.im one two three

[7] The functions *printf()* and *element()* used in this example are discussed in section 3.6.

In both cases, the first *int* argument of the function *main ()* equals four. The first element of the list *argv* holds the name of the binary makefile (*test.bim*), the remaining elements of *argv* hold the arguments *one*, *two* and *three*.

The third argument of *main()*, *envp*, is a list holding the setting of the environment (the environment variables). An example of such a variable is the *PATH* setting under MS-DOS, which determines where the operating system searches for executable files. The *envp* list consists of pairs of elements, where each first element of the pair holds the variable name (e.g., the string "PATH") and where the second element of each pair holds the value of the variable (e.g,, a list of directories where executable files may be found).

An example of a makefile which prints the settings of environment variables is given below:

```
void main (int argc, list argv, list envp)
{
    int
        i;

    for (i = 0; i < sizeof (envp); i += 2)
        printf ("variable ", element (i, envp), " has value ",
               element (i + 1, envp));
}
```

3.5 Expressions and operators

Icmake allows a large number of operators to form or combine expressions. Each operator has its own characteristics, which are the following:

- The number of operands which an operator uses may be two or one. In analogy to **C**, *icmake* recognizes binary or unary operators.

- Each binary operator must be used with two variables or constants of the same type. E.g., the addition of an *int* and a *string* is not allowed: *icmake* performs no default type casting. The operand types for which operators are allowed is given in figure 2. The shown matrix shows which types may use which operators.

	int	string	list
int	*all but* newer older younger	—	—
string	—	== != > < <= >= + += = newer older younger	—
list	—	—	== != + - += -= =

Figure 2: The operand matrix for binary operators

- Some operators, unary and binary, may not be used with some types. E.g., string subtraction is not allowed, but string addition is.
- The operators have a certain priority: some operators are evaluated before others. The priority of operators is analogous to **C**.

The binary operators recognized by *icmake* are summarized in figure 3. All binary operators with the exception of the assignment operators are left-associative. The assignment operators are right-associative. The operators at the top of this figure have the lowest priority; the operators at the bottom of this figure have the highest priority. Operators with different priority are separated by lines.

The unary operators are summarized in table 4. The unary operators have higher priority than binary operators, and are right-associative. The exception is the expression-nesting operator, which surrounds an expression and does not associate.

Logical operators

Icmake recognizes three logical operators: the logical and (&&), the logical or (||) and the logical not (!). These operators can be used to combine or reverse logical expressions.

Operator	Operation	Allowed types
,	sequential evaluation	all (mixed)
=	assignment	all
−= += ∗= /= %= &= |= ^=	compound assignment	arithm. operators
||	logical or	all (mixed)
&&	logical and	all (mixed)
|	bitwise or	*int*
^	exclusive or	*int*
&	bitwise and	*int*
< > <= >=	comparison	*int, string*
newer, younger	file date	*string*
older	file date	*string*
<<	leftshift	*int*
>>	rightshift	
+	addition	all
−	subtraction	*int, list*
∗	multiplication	*int*
/	division	*int*
%	modulo	*int*

Figure 3: Overview of binary operators

Operator	Operation	Allowed types
!	logical not	all
~	bitwise not	*int*
+	unary plus	all
−	unary minus	*int*
++	increment	*int* (variable only)
−−	decrement	*int* (variable only)
(*type*)	type cast	see 3.5
( )	expression nesting	all

Figure 4: Overview of unary operators

The logical not operator reverses the logical outcome of an expression. The logical and operator and the logical or operator group conditions. *Icmake* evaluates a combined condition using these operators until the outcome of the condition is determined, in analogy to **C**:

- In the condition *c1 && c2*, *c2* is not evaluated if *c1* yields zero since when *c1* yields zero, the combined condition can only fail. Therefore, *c2* is only evaluated if *c1* yields not zero.
- In the condition *c1* || *c2*, *c2* is not evaluated if *c1* yields not zero since when *c1* yields not zero, the combined condition can only succeed. Therefore, *c2* is only evaluated if *c1* yields zero.

This is illustrated in the following code fragment:

```
if (exists ("main.c") && exists ("main.exe"))
    print (0, "main.c and main.exe both found.\n");
if (exists ("main.c") || exists ("main.exe"))
    print (0, "main.c exists,\n",
              "or main.c not found but main.exe found.\n");
```

Logical operators may be used with any type of expression. An *int* constant or variable yields its integer representation. A *string* constant or variable yields not zero when the length of the string is non-zero; e.g., string *"a"* yields not zero.

A *list* or variable yields not zero when the number of strings in the list is not zero. E.g., in the following code fragment the making process is stopped when no files with extension *".c"* are found:

```
list
    cfiles;

// cfiles is set to hold all filenames with extension .c
cfiles = makelist ("*.c");
// if no such files found... terminate!
if (! cfiles)
{
    print (0, "No files with extension .c found!!\n");
    exit (1);
}
```

Special operators

In addition to the operators of the **C** programming language, *icmake* recognizes a few 'special' operators. These are:

- The *younger* operator is used to compare two strings which represent filenames. An expression using *younger* is evaluated to non-zero or zero and may be used in a condition. The operator *newer* is an alias for *younger*.

  The expression using the *younger* operator yields non-zero if a file with the name represented by the left operand is more recent than the file represented by the right operand.

  E.g., the following code prints a message if file *main.c* is more recent than *main.exe*:

  ```
  if ("main.c" newer "main.exe")
      printf ("main.c is more recent than main.exe\n");
  ```

- The *older* operand compares two files and yields non-zero if the file represented by the left operand is older than the file represented by the right operand.

When the date of a file is compared using *older* or *younger* and when no file with such a name is present, then the age if the file is assumed to be infinite. A consequence of this implementation is that, in the following code example, a message is displayed if *"test.exe"* does not exist:

```
if ("test.c" younger "test.exe")
    printf ("test.c should be compiled!!\n");
```

Type casts

Though *icmake* does not allow the use of operators on different types, a possibility consists of the conversion of one type to another. The conversion of a type is referred to as 'type cast'.

Type casts are denoted by a type name in parentheses before the operand which should be converted. E.g., *(int)x* converts the operand

x to integer representation. Type casts are not allowed on all types; e.g., a list variable can not be converted to *int*.
Allowed type casts are the following:

- An integer may be cast to *string*. E.g., in the following listing *stringvar* is assigned to hold the text representation of the value *14*.

```
string
    stringvar;

stringvar = (string) 14;        // now, stringvar is "14"
```

- A string may be cast to *int*. This is the reverse action of the type cast shown in the listing above.

- A string may be cast to a *list*. This may be particularly useful when filenames should be added to or removed from a list. E.g., in the listing below the filename *"main.c"* (a *string*) is removed from the list *cfiles*:

```
list
    cfiles;

// cfiles is set to hold a list of
// all filenames with extension .c
cfiles = makelist ("*.c");
// filename main.c is removed from the list
cfiles -= (list) "main.c";
```

  Note that the string *"main.c"* **must** be converted to a *list* type to allow the subtraction from the list.

Other typecast, specifically from a string to an ASCII-representation, can be realized through specialized functions (see, e.g., the function *ascii()*.)

3.6 Built-in functions

Built into *icmake* is a number of functions which may be used to perform special operations, such as scanning a directory for files, displaying information, etc.. This section describes all built-in functions.

arghead

(int) arghead (string): This function sets the argument head to *string*. See *exec()* and *execute()*.

argtail

(int) argtail (string): This function sets the argument tail to *string*. See *exec()* and *execute()*.

ascii

(int) ascii (string): This function returns the ASCII-number of the first character in the string, supplied as argument. E.g., *ascii ("A")* returns 65.

ascii

(string) ascii (int): The overloaded function, which expects an *int* argument, returns a string representation of a numeric ASCII number. E.g., *ascii (65)* returns "A".

change_base

(string) change_base (string, string): This function changes the basename in the string which is supplied as its first argument to the basename which is supplied as second argument. The string with the changed basename is returned.

Example:

```
string
    name;

name = change_base ("main.c", "test");
// name now is "test.c"
```

change_ext

(string) change_ext (string, string): This function changes the extension in the string which is supplied as its first argument to the extension which is supplied as second argument. The modified string is returned.

The extension (the second argument) may be specified as an empty string (""); in this case *change_ext()* removes the extension. Also, the extension may be specified as one dot ("."); in this case *change_ext()* removes the extension but leaves the dot.

Example:

```
char
    name;

name = change_ext ("main.c", "obj");
// name now is "main.obj"
name = change_ext (name, "");
// name now is "main"
name = change_ext (name, ".");
// name now is "main."
```

change_path

(string) change_path (string, string): This function changes the path in the string which is supplied as its first argument to the path which is supplied as second argument.

Example:

```
string
    name;

name = change_path ("c:/prog/c/prog.c", "/bin");
// name now is "c:/bin/prog.c"
name = change_path (name, "");
// name now is "c:prog.c"
```

chdir

(string) chdir (int, string): This function changes the current working directory to the supplied name. The first *int* argument may be

P_CHECK or *P_NOCHECK*. This argument is optional; when absent, *P_CHECK* is assumed. Failure to change the working directory with the presence of *P_CHECK* leads to the termination of the making process.

When the processing of the makefile terminates the original directory is restored.

A string containing the new working directory, always ending in a directory separator[8] is returned. The *string* argument may terminate in a final (back)slash. The returned string can be used to inspect whether the requested directory is reached, given that the modifier *P_NOCHECK* is supplied as first argument.
Two special string arguments are recognized by *chdir ()*:

- A directory argument which consists of one dot (i.e., the string ".") causes *icmake* to change to the current directory. The return value is then a string holding the current directory.
- A directory argument which is an empty string (i.e., the string "") causes *icmake* not to change the current directory but to return the startup-directory, from which *icmake* was started.

Example:

```
// print the current working directory
printf("Current dir: ", chdir ("."), "\n");
// change to directory /c/prog
chdir ("/c/prog");
// print startup directory
printf ("Startup dir: ", chdir (""), "\n");
```

cmdhead

(int) cmdhead (string): This function sets the command head to *string*. See *exec()* and *execute()*.

cmdtail

(int) cmdtail (string): This function sets the command tail to *string*. See *exec()* and *execute()*.

[8] The directory separator is a backslash under MS-DOS and a slash under UNIX.

echo

(int) echo (int): This function determines whether before the execution of a command the command will be displayed. The argument of the function determines the displaying mode: when zero, displaying is suppressed; else, commands are displayed before execution. Two predefined constants are available for use as an argument to *echo()*: the constants *ON* and *OFF*. The values of these constants are, respectively, 1 and 0. Initially, echoing is on.

Example:

```
echo (ON);    // commands will be displayed
.
.
echo (OFF);   // commands will not be displayed
```

element(int, list)

(string) element (int, list): This function retrieves a string from a list. The order number of the name in the list is given by the first argument. Note that this index is zero-based; i.e., the first element in the list has index 0. The last element in the list has index *sizeoflist(list) – 1*.

Example:

```
list
    l;
string
    n;
int
    i;

l = makelist ("*.c");
for (i = 0; i < sizeoflist (l); i++)
    if (element (i, l) newer "main.exe")
        printf ("Source file ", element (i, l),
            " is more recent than main.exe\n");
```

element(int, string)

(string) element (int, string): This function retrieves a substring of

one character from the string given as its second argument.

The character which is returned is found in the second (string) argument at the offset position specified in the first (int) argument. This index is zero-based; i.e., the first character of the string has index 0.

Example:

```
string
    s;
int
    count;
    i;

count = 0;
s = "Hello world";
for (i = 0; element(i, s); i++)
    count++;

printf("String '", s, "' contains ",
       count, " characters.\n");
```

exec

(int) exec (int, string, ...): This function executes a command by spawning a child process. The arguments are:

- The first argument is an *optional* mode (an *int*). It may be *P_CHECK* (0) or *P_NOCHECK* (2). These predefined constants determine whether the exit status of the command should be checked or not. If the exit status should be checked, and a non-zero value is returned by the called program, the processing of the icmake file is aborted.

  If the first argument is omitted (i.e., if the first argument is *not* an int), *P_CHECK* is assumed.

- The second argument is the command to run (a *string*). This is the name of the program to be activated.

- The following arguments are the arguments which should be passed to the called program. These arguments may be *int*s, *list*s or *string*s.

Each command is composed of the program name (the second argument), followed by the current setting of the command head (see *cmdhead()*, followed by all arguments and terminated with the command tail (see *cmdtail()*). Each argument to the command is prefixed with the argument head (see *arghead()*) and postfixed by the argument tail (see *argtail()*).

Example: see *execute()* below.

execute

(int) execute (int, string, string, string, ..., string, string): This function executes a command by spawning a child process. The arguments are:

- The first argument is an *optional* mode (an *int*). It may be *P_CHECK* (0) or *P_NOCHECK* (2). These predefined constants determine whether the exit status of the command should be checked or not. If the exit status should be checked, and a nonzero value is returned by the called program, the processing of the icmake file is aborted.

  If the first argument is omitted (i.e., if the first argument is *not* an int), *P_CHECK* is assumed.
- The second argument is the command to run (a *string*). This is the name of the program to be activated.
- The third argument is the command head (a *string*). This string is used as first argument to the program name. The string may be empty (i.e., ""), in which case no command head is used.
- The fourth argument is the argument head (a *string*). This string is prefixed to all following arguments. The string may be empty (i.e., ""), in which case no argument head is used.
- The following arguments are the arguments which should be passed to the called program. These arguments may be *int*s, *list*s or *string*s.
- The one but last argument is the argument tail (a *string*). This string is postfixed to each argument passed to the called program.

The string may be empty (i.e., ""), in which case no argument tail is used.

- The last argument is the command tail (a *string*). The command run by the *execute()* function is postfixed with this string. The string may be empty (i.e., ""), in which case no command tail is used.

After execution, *execute()* resets the command head, command tail, argument head and argument tail to empty strings.

The following listing shows an example of the use of *execute()*. It involves the Microsoft Library Manager. This program, which is called *"lib"*, has always the library name to process as first argument (the command head). Following this argument, all object files which should be processed are stated preceded by a string which specifies the type of operation. The type of operation is, e.g., *"−+"* which specifies that an object should be replaced in the library. Following this, a semicolon must close the command line to prevent the library manager of asking for the new library name etc..

```
list
    objfiles;

// objfiles is set to a list of all objects
objfiles = makelist ("*.obj");

execute (0,           // P_CHECK
         "lib",       // the program name
         "prog.lib",  // the command head
         "-+",        // the argument head: replacement
         objfiles,    //   of all object files
         "",          // no argument tail
         ";");        // command tail: semicolon

// this may lead to e.g.:
// "lib prog.lib -+a.obj -+b.obj -+c.obj ;"
```

The same effect using function *exec()* could be accomplished in the following manner:

```
list
    objfiles;
```

```
objfiles = makelist ("*.obj");

cmdhead ("prog.lib"); // set command head
cmdtail (";");        //  and tail
arghead ("-+");       // set argument head
argtail ("");         //  and tail
exec (0, "lib", objfiles);
cmdhead ("");         // clear command head
cmdtail ("");         //  and tail
arghead ("");         // clear argument head
argtail ("");         //  and tail

// this may lead to e.g.:
// "lib prog.lib -+a.obj -+b.obj -+c.obj ;"
```

Both the *exec()* and *execute()* function terminate the making process if error checking is turned on (mode flag *P_CHECK*) and if the run command exits with a non-zero exit value. If error checking is off, the exit status of the child process is returned.

exists

(int) exists (string): This function tests if a file exists. The file name is supplied as argument. A non-zero value is returned when the file exists, else, zero is returned.

Example:

```
if (exists ("main.c"))
    printf ("file main.c found\n");
else
    printf ("file main.c not found\n");
```

fgets

(list) fgets (string, int): This function reads a line of text from the file whose name is given as its first (string) argument. Reading starts at the offsetposition specified in the second (int) argument. A list is returned, containing as its first element the string which was read, including the final newline character (as it is returned by the **C++** function *fgets()*). The second element of the returned list contains

the string representation of the offset of the file *after* the line was read. This string can be cast to an int.

Example:

```
// showing the file info.doc on the screen:
int
    offset;
list
    l;

for
(
    offset = 0;
        l = fgets("info.doc", offset);
            offset = (int)element(1, l)
)
    printf(element(0, l));
```

fprintf

(int) fprintf (string, ...): This function appends information to the file whose name is given as its first string argument. The remaining arguments define the information which is written to the file. The information is always appended to an existing file, which is opened in textmode.

fprintf() acts analogously to *printf()* (see below), but the information is written to file, rather than to screen.

The arguments beyond the first argument of *fprintf()* define the information to print and may be *int*s, *list*s or *string*s.

Example:

```
fprintf ("file.txt", 1,
         " line written to file.txt\n");
```

get_base

(string) get_base (string): This function returns the basename of the filename stored in the string argument. The empty string is returned if the argument contains no basename. This happens when a disk or

a root directory is specified in *string*. It may also happen if the syntax rules for a filename specification are violated.

Example:

```
printf(get_base ("c:/path/main.c"));
// prints 'main'

printf(get_base ("c:main.c"));
// prints 'main'

printf("No basename: ", get_base ("c:/"));
// prints 'No basename: '
```

getch

(string) getch (): This function returns one character as a ministring. The character is read from the standard input stream (usually the keyboard).

Under UNIX, this function waits until a key and the enter key is pressed. Under DOS, only one key needs to be pressed.

Example:

```
printf(getch());  // prints a character
                  // (or an empty string)
```

get_ext

(string) get_ext (string): This function returns the extension in the string argument of the function.

The empty string is returned if the argument contains no extension.

Example:

```
printf(get_ext ("c:/path/main.c"));
// prints 'c'

printf(get_ext ("c:main"));
// returns empty string
```

get_path

(string) get_path (string): This function returns the path stored in the string argument of the function. An empty string is returned if *string* contains neither a disk specifier nor a directory separator (a forward- or backward slash). The function returns the longest possible path-name which can be derived from the string argument.

Example:

```
printf(get_path ("c:/path/main.c"));
// prints 'c:/path/'

printf(get_path ("c:main.c"));
// prints 'c:'

printf(get_path ("c:/"));
// prints 'c:/'
```

getpid

(int) getpid () This function returns under UNIX the process number of the currently executing *icmake* program. Under MS-DOS, where process ID numbers have no special meaning since no parallel program may be run, *getpid ()* returns the segment of the memory location where the executor *icm-exec* is loaded (this value is the PSP address).

Example:

```
// this function kills the current process..
// analogous to exit()
// works only under UNIX
void harakiri ()
{
    exec ("kill", "-9", getpid ());
}

// this function returns, under UNIX,
// a name for a temporary file
// based on the process ID number
// file names are, eg: "/tmp/_TMPFILE.1256"
string tempfilename ()
{
    return ("/tmp/" + "_TMPFILE." + (string) getpid ());
}
```

gets

(string) gets (): This function returns the returnvalue of the **C** function *gets()* as a string. The function accepts character, including backspaces allowing corrections, until the *enter-key* is pressed. The entered characters are returned in a string.

Example:

```
printf(gets());  // prints a string
                 // (or an empty string)
```

makelist

(list) makelist (int, string): This function makes a list of strings, representing filenames which match the expanded form of its argument. The arguments are an optional *int*, specifying the type of directory entries to search for, and a *string*, specifying the file mask. The returned list may hold zero or more names.

The first *int* argument specifies the type of entries to search for. It may be *O_FILE* (when searching for files), *O_DIR* (when searching for directories) or *O_SUBDIR* (when searching for subdirectories). The difference between the searching for directories and the searching for subdirectories lies in the fact that the current directory, denoted by ".", and the parent directory, denoted by "..", are not considered subdirectories but are considered directories. This argument may be absent, in which case *O_FILE* is assumed.

A fourth entry type is *O_ALL*. When this type is given, *makelist ()* searches for all directory entries irrespective of their type. E.g., under DOS, hidden and system files are matched as well as normal files or (sub)directories.

The behavior of *makelist()* is dependent on the used platform. E.g., to search for all files or (sub)directories under DOS, the file mask "*.*" must be given. The file mask "*" will fail to find files or (sub)directories with an extension.

In a similar vein, the filemask "*" under UNIX will fail to find files or (sub)directories starting with a dot.

Example:

```
list
```

```
    l;

l = makelist ("*.c");
printf ("All found *.c files are: ",
        l, "\n");
l = makelist (O_SUBDIR, "*.*");
printf ("All found subdirectories are: ",
        l, "\n");
```

makelist

The function *makelist()* furthermore has overloaded versions, which, apart from a first optional *int* indicating the type of entries to search for, expect three arguments[9]. The arguments of these versions are a *string mask*; a comparison operator which must be *younger*, *newer* or *older*; and a *string referencefile*. The function returns a list of files matching the mask which are older or newer than the referencefile[10]. An optional first *int* argument, which specifies the type of directory entry (*O_FILE, O_DIR* or *O_SUBDIR*) may be present.

Example:

```
list
    l;

l = makelist ("*.c", newer, "myprog.lib");
printf ("All .c files newer than "
       "myprog.lib are: ", l, "\n");
```

printf

(int) printf (int, ...): This function displays information. The arguments define the information to print and may be *int*s, *list*s or *string*s.

A list is printed as a series of all its elements with spaces in between.

[9] The overloaded versions are in fact macros, expanded by the compiler.

[10] The overloaded functions are handled by the compiler *icm-comp*, which generates appropriate code to make a list of files and to weed out a part of this list.

putenv

putenv (string): This function may be used to set environment variables during the execution of *icmake* programs. The environment variables remain active during the complete *icmake* run.

Example:

```
main()
{
    putenv("CL=/c");  // set variable
    system("set");    // show settings
}
```

sizeof

(int) sizeof (list): This function performs the same action as *sizeoflist()*.

sizeoflist

(int) sizeoflist (list): This function determines the number of names held in a list.

Example:

```
list
    l;
int
    i;

list = makelist ("*.c");
i = sizeoflist (l);
printf ("There are ", i,
        " names in the list.\n");
```

stat

(list) stat (int, string): This function attempts to retrieve file attributes of the file specified by the second *string* argument. The first *int* argument may be *P_CHECK* or *P_NOCHECK*. When absent, *P_CHECK* is assumed. The making process is aborted when *stat ()* fails to retrieve file attributes and when *P_CHECK* is given for the first argument.

The returned list holds the following information:

- The first element is a *string* representation of the mode of the file or directory. This string can be converted to an *int* where the following bits represent the modes:
  - The bit *S_IFDIR* is set when the entry is a directory.
  - The bit *S_IFCHR* is set when the entry is a character-special.
  - The bit *S_IFREG* is set when the entry is a regular file.
  - The bit *S_IREAD* is set when the entry is readable.
  - The bit *S_IWRITE* is set when the entry is writeable.
  - The bit *S_IEXEC* is set when the entry is executable.
- The second element is the file size, also represented as a *string*.

strlen

(int) strlen (string): This function returns the number of characters in the string which is supplied as its argument. Its working is analogous to the function *strlen()* in **C**[11].

strlwr

(string) strlwr (string): This function returns a copy of the string which is supplied as argument in lower case[12].

strupr

(string) strupr (string): This function returns a copy of the string which is supplied as argument in upper case.

[11] This function is a macro, expanded by the compiler.

[12] The string functions *strlwr()* and *strupr()* are in fact macro's, expanded by the compiler.

strtok

(list) strtok (string, string): This function parses the first string in substrings, separated from each other by the separators specified in the second string. Each substring is an element of the returned list. If no separators are specified (i.e., the second string is empty), the individual characters of the first string become the elements of the returned list.

Example:

```
list
    l;
int
    i;

l = strtok("Hello-world\n", "-");
printf("Two elements: ", l, "\n");

l = strtok("Hello-world\n", "");
printf("A string of ", sizeof(l),
       " characters\n");
```

substr

(int) substr (string, string): This function searches for the string which is given as the second argument in the string given as first argument. The position where the second string occurs in the first string is returned. The value -1 is returned when the second string does not occur in the first string[13].

system

(int) system (int, string): This function activates the operating system's command interpreter to run the command defined by the argument. The string holds the command to execute and, if needed, its arguments.

The first argument specifies whether a failure of the *system ()* function should terminate the making process. The value of this *int* may

[13]This function is a macro, which is expanded by the compiler.

be *P_CHECK* or *P_NOCHECK*. This argument may be absent, in which case *P_CHECK* is assumed. The return value of *system ()* is zero when the command could successfully be executed. A return value which is not zero can only be received by the makefile when *P_NOCHECK* is given as first argument.

The definition of a success or failure of *system ()* depends on the used platform. Under MS-DOS, *system ()* succeeds if the command could be executed, but no checks on the return value of the command itself can be made. Under UNIX, *system ()* succeeds if the command could be executed and if the return value of the command itself is zero.

4 Summary

A Keywords

The following table shows the keywords of the *icmake* language.

arghead	argtail	ascii	break	chdir
cmdhead	cmdtail	change_base	change_ext	change_path
echo	element	else	exec	execute
exists	fgets	for	fprintf	get_base
get_ext	getch	get_path	gets	if
int	list	makelist	newer	older
printf	putenv	return	sizeof	sizeoflist
stat	string	strlwr	strtok	strupr
system	while	younger		

B Symbolic constants

The following symbolic constants are defined:

Name	value	Name	value
OFF	0	ON	1
P_CHECK	0	P_NOCHECK	2
O_FILE	1	O_DIR	2
O_SUBDIR	4	O_ALL	8
S_IFDIR	1	S_IFCHR	2
S_IFREG	4	S_IREAD	8
S_IWRITE	16	S_IEXEC	32

C Binary operators

In the following table, 'all' means that the type of the operand left of the operator must be equal to the type of the operand right of the operator. The operators at the top of the table have the lowest priority. The assignment operators are right-associative, the other binary operators are left-associative.

Operator	**Operation**	**Allowed types**
,	sequential evaluation	all (mixed)
=	assignment	all
-= += *= /= %= &= |= ^=	compound assignment	arithm. operators
||	logical or	all (mixed)
&&	logical and	all (mixed)
|	bitwise or	*int*
^	exclusive or	*int*
&	bitwise and	*int*
< > <= >=	comparison	*int, string*
newer, younger	file date	*string*
older	file date	*string*
<<	leftshift	*int*
>>	rightshift	
+	addition	all
-	subtraction	*int, list*
*	multiplication	*int*
/	division	*int*
%	modulo	*int*

D Unary operators

All unary operators are right-associative. The exception is the expression-nesting operator, which surrounds an expression and does not associate.

Operator	Operation	Allowed types
!	logical not	all
~	bitwise not	*int*
+	unary plus	all
−	unary minus	*int*
++	increment	*int* (variable only)
−−	decrement	*int* (variable only)
(*type*)	type cast	see 3.5
( )	expression nesting	all

E Type casts

The following table shows the allowed type casts. (Ok) is used in the table to indicate that no special action is needed. The conversions *itoa*, *atoi* and *atol* specify respectively *int to ascii*, *ascii to int* and *ascii to list*.

	cast result:		
	(int)	(string)	(list)
cast operand:			
int	(ok)	itoa	—
string	atoi	(ok)	atol

(ok) means: no special casting performed

F Operand matrix

	int	string	list
int	*all but* newer older younger	—	—
string	—	== != > < <= >= + += = newer older younger	—
list	—	—	== != + − += −= =

G Numbers and identifiers

Numbers are integral values of at most 32767 (negative numbers are obtained using e.g., the unary minus.

Identifiers are C-like identifiers: they start with an underscore or letter, and consist furthermore of alphanumeric characters or underscores. The *icm-comp* compiler interprets identifiers case-sensitively.

H Escape sequences

Escape sequence	Action
\a	alert (bell)
\b	backspace character
\f	formfeed character
\n	newline
\r	carriage return character
\t	tab
\v	vertical tab
other	literal *other*