Ack Description File
Reference Manual

Ed Keizer

Vakgroep Informatica
Vrije Universiteit
Amsterdam

1. Introduction

The program *ack*(I) internally maintains a table of possible transformations and a table of string variables. The transformation table contains one entry for each possible transformation of a file. Which transformations are used depends on the suffix of the source file. Each transformation table entry tells which input suffixes are allowed and what suffix/name the output file has. When the output file does not already satisfy the request of the user (indicated with the flag **−c.suffix**), the table is scanned starting with the next transformation in the table for another transformation that has as input suffix the output suffix of the previous transformation. A few special transformations are recognized, among them is the combiner, which is a program combining several files into one. When no stop suffix was specified (flag **−c.suffix**) *ack* stops after executing the combiner with as arguments the − possibly transformed − input files and libraries. *Ack* will only perform the transformations in the order in which they are presented in the table.

The string variables are used while creating the argument list and program call name for a particular transformation.

2. Which descriptions are used

Ack always uses two description files: one to define the front-end transformations and one for the machine dependent back-end transformations. Each description has a name. First the way of determining the name of the descriptions needed is described.

When the shell environment variable ACKFE is set *ack* uses that to determine the front-end table name, otherwise it uses **fe**.

The way the backend table name is determined is more convoluted.
First, when the last filename in the program call name is not one of *ack* or the front-end call-names, this filename is used as the backend description name. Second, when the **−m** is present the **−m** is chopped of this flag and the rest is used as the backend description name. Third, when both failed the shell environment variable ACKM is used. Last, when also ACKM was not present the default backend is used, determined by the definition of ACKM in h/local.h. The presence and value of the definition of ACKM is determined at compile time of *ack*.

Now, we have the names, but that is only the first step. *Ack* stores a few descriptions at compile time. This descriptions are simply files read in at compile time. At the moment of writing this document, the descriptions included are: pdp, fe, i86, m68k2, vax2 and int. The name of a description is first searched for internally, then in lib/descr/*name*, then in lib/*name*/descr, and finally in the current directory of the user.

3. Using the description file

Before starting on a narrative of the description file, the introduction of a few terms is necessary. All these terms are used to describe the scanning of zero terminated strings, thereby producing another string or sequence of strings.

Backslashing

All characters preceded by \ are modified to prevent recognition at further scanning. This modification is undone before a string is passed to the outside world as argument or message. When reading the description files the sequences \\, \# and \<newline> have a special meaning. \\ translates to a single \, \# translates to a single # that is not recognized as the start of comment, but can be used in recognition and finally, \<newline> translates to nothing at all, thereby allowing continuation lines.

Variable replacement

The scan recognizes the sequences {{, {NAME} and {NAME?text} Where NAME can be any combination if characters excluding ? and } and text may be anything excluding }. (\} is allowed of course) The first sequence produces an unescaped single {. The second produces the contents of the NAME, definitions are done by *ack* and in description files. When the NAME is not defined an error message is produced on the diagnostic output. The last sequence produces the contents of NAME if it is defined and text otherwise.

Expression replacement

Syntax: (*suffix sequence*:*suffix sequence=text*)

Example: (.c.p.e:.e=tail_em)

If the two suffix sequences have a common member – .e in this case – the text is produced. When no common member is present the empty string is produced. Thus the example given is a constant expression. Normally, one of the suffix sequences is produced by variable replacement. *Ack* sets three variables while performing the diverse transformations: HEAD, TAIL and RTS. All three variables depend on the properties *rts* and *need* from the transformations used. Whenever a transformation is used for the first time, the text following the *need* is appended to both the HEAD and TAIL variable. The value of the variable RTS is determined by the first transformation used with a *rts* property.

Two runtime flags have effect on the value of one or more of these variables. The flag **–.suffix** has the same effect on these three variables as if a file with that **suffix** was included in the argument list and had to be translated. The flag **–r.suffix** only has that effect on the TAIL variable. The program call names *acc* and *cc* have the effect of an automatic **–.c** flag. *Apc* and *pc* have the effect of an automatic **–.p** flag.

Line splitting

The string is transformed into a sequence of strings by replacing the blank space by string separators (nulls).

IO replacement

The > in the string is replaced by the output file name. The < in the string is replaced by the input file name. When multiple input files are present the string is duplicated for each input file name.

Each description is a sequence of variable definitions followed by a sequence of transformation definitions. Variable definitions use a line each, transformations definitions consist of a sequence of lines. Empty lines are discarded, as are lines with nothing but comment. Comment is started by a # character, and continues to the end of the line. Three special two-characters sequences exist: \#, \\ and \<newline>. Their effect is described under 'backslashing' above. Each – nonempty – line starts with a keyword, possibly preceded by blank space. The keyword can be followed by a further specification. The two are separated by blank space.

Variable definitions use the keyword *var* and look like this:

```
var NAME=text
```

The name can be any identifier, the text may contain any character. Blank space before the equal sign is not

part of the NAME. Blank space after the equal is considered as part of the text. The text is scanned for variable replacement before it is associated with the variable name.

The start of a transformation definition is indicated by the keyword *name*. The last line of such a definition contains the keyword *end*. The lines in between associate properties to a transformation and may be presented in any order. The identifier after the *name* keyword determines the name of the transformation. This name is used for debugging and by the **−R** flag. The keywords are used to specify which input suffices are recognized by that transformation, the program to run, the arguments to be handed to that program and the name or suffix of the resulting output file. Two keywords are used to indicate which run-time startoffs and libraries are needed. The possible keywords are:

from
: followed by a sequence of suffices. Each file with one of these suffices is allowed as input file. Preprocessor transformations do not need the *from* keyword. All other transformations do.

to
: followed by the suffix of the output file name or in the case of a linker the output file name.

program
: followed by name of the load file of the program, a pathname most likely starts with either a / or {EM}. This keyword must be present, the remainder of the line is subject to backslashing and variable replacement.

mapflag
: The mapflags are used to grab flags given to *ack* and pass them on to a specific transformation. This feature uses a few simple pattern matching and replacement facilities. Multiple occurrences of this keyword are allowed. This text following the keyword is subjected to backslashing. The keyword is followed by a match expression and a variable assignment separated by blank space. As soon as both description files are read, *ack* looks at all transformations in these files to find a match for the flags given to *ack*. The flags **−m**, **−o**, **−O**, **−r**, **−v**, **−g**, **−−c**, **−t**, **−k**, **−R** and **−−.** are specific to *ack* and not handed down to any transformation. The matching is performed in the order in which the entries appear in the definition. The scanning stops after first match is found. When a match is found, the variable assignment is executed. A * in the match expression matches any sequence of characters, a * in the right hand part of the assignment is replaced by the characters matched by the * in the expression. The right hand part is also subject to variable replacement. The variable will probably be used in the program arguments. The **−l** flags are special, the order in which they are presented to *ack* must be preserved. The identifier LNAME is used in conjunction with the scanning of **−l** flags. The value assigned to LNAME is used to replace the flag. The example further on shows the use of all this.

args
: The keyword is followed by the program call arguments. It is subject to backslashing, variable replacement, expression replacement, line splitting and IO replacement. The variables assigned to by *mapflags* will probably be used here. The flags not recognized by *ack* or any of the transformations are passed to the linker and inserted before all other arguments.

stdin
: This keyword indicates that the transformation reads from standard input.

stdout
: This keyword indicates that the transformation writes on standard output.

optimizer
: The presence of this keyword indicates that this transformation is an optimizer. It can be followed by a number, indicating the "level" of the optimizer (see description of the -O option in the ack(1ACK) manual page).

priority
: This − optional − keyword is followed by a number. Positive priority means that the transformation is likely to be used, negative priority means that the transformation is unlikely to be used. Priorities can also be set with a ack(1ACK) command line option. Priorities come in handy when there are several

implementations of a certain transformation. They can then be used to select a default one.

linker

This keyword indicates that this transformation is the linker.

combiner

This keyword indicates that this transformation is a combiner. A combiner is a program combining several files into one, but is not a linker. An example of a combiner is the global optimizer.

prep

This – optional – keyword is followed an option indicating its relation to the preprocessor. The possible options are:

always	the input files must be preprocessed
cond	the input files must be preprocessed when starting with #
is	this transformation is the preprocessor

rts

This – optional – keyword indicates that the rest of the line must be used to set the variable RTS, if it was not already set. Thus the variable RTS is set by the first transformation executed which such a property or as a result from *ack*'s program call name (acc, cc, apc or pc) or by the **–.suffix** flag.

need

This – optional – keyword indicates that the rest of the line must be concatenated to the HEAD and TAIL variables. This is done once for every transformation used or indicated by one of the program call names mentioned above or indicated by the **–.suffix** flag.

4. Conventions used in description files

Ack reads two description files. A few of the variables defined in the machine specific file are used by the descriptions of the front-ends. Other variables, set by *ack*, are of use to all transformations.

Ack sets the variable EM to the home directory of the Amsterdam Compiler Kit. The variable SOURCE is set to the name of the argument that is currently being massaged, this is useful for debugging. The variable SUFFIX is set to the suffix of the argument that is currently being massaged.
The variable M indicates the directory in lib/{M}/tail_..... and NAME is the string to be defined by the preprocessor with –D{NAME}. The definitions of {w}, {s}, {l}, {d}, {f} and {p} indicate EM_WSIZE, EM_SSIZE, EM_LSIZE, EM_DSIZE, EM_FSIZE and EM_PSIZE respectively.
The variable INCLUDES is used as the last argument to *cpp*. It is used to add directories to the list of directories containing #include files.

The variables HEAD, TAIL and RTS are set by *ack* and used to compose the arguments for the linker.

5. Example

Description for front-end

```
name cpp                                    # the C-preprocessor
                                            # no from, it's governed by the P property
    to .i                                   # result files have suffix i
    program {EM}/lib/cpp                    # pathname of loadfile
    mapflag -I* CPP_F={CPP_F?} -I*          # grab -I.. -U.. and
    mapflag -U* CPP_F={CPP_F?} -U*          # -D.. to use as arguments
    mapflag -D* CPP_F={CPP_F?} -D*          # in the variable CPP_F
    args {CPP_F?} {INCLUDES?} -D{NAME} -DEM_WSIZE={w} -DEM_PSIZE={p} \
      -DEM_SSIZE={s} -DEM_LSIZE={l} -DEM_FSIZE={f} -DEM_DSIZE={d} <
                                            # The arguments are: first the -[IUD]...
                                            #  then the include dir's for this machine
                                            #  then the NAME and size values finally
                                            #  followed by the input file name
    stdout                                  # Output on stdout
    prep is                                 # Is preprocessor
end
name cem                                    # the C-compiler proper
    from .c                                 # used for files with suffix .c
    to .k                                   # produces compact code files
    program {EM}/lib/em_cem                 # pathname of loadfile
    mapflag -p CEM_F={CEM_F?} -Xp           # pass -p as -Xp to cem
    mapflag -L CEM_F={CEM_F?} -l            # pass -L as -l to cem
    args -Vw{w}i{w}p{p}f{f}s{s}l{l}d{d} {CEM_F?}
                                            # the arguments are the object sizes in
                                            # the -V... flag and possibly -l and -Xp
    stdin                                   # input from stdin
    stdout                                  # output on stdout
    prep always                             # use cpp
    rts .c                                  # use the C run-time system
    need .c                                 # use the C libraries
end
name decode                                 # make human readable files from compact code
    from .k.m                               # accept files with suffix .k or .m
    to .e                                   # produce .e files
    program {EM}/lib/em_decode              # pathname of loadfile
    args <                                  # the input file name is the only argument
    stdout                                  # the output comes on stdout
end
```

Example of a backend, in this case the EM assembler/loader.

```
var w=2                                 # wordsize 2
var p=2                                 # pointersize 2
var s=2                                 # short size 2
var l=4                                 # long size 4
var f=4                                 # float size 4
var d=8                                 # double size 8
var M=em22
var NAME=em22                           # for cpp (NAME=em22 results in #define em22 1)
var LIB=lib/{M}/tail_                   # part of file name for libraries
var RT=lib/{M}/head_                    # part of file name for run-time startoff
var SIZE_FLAG=−sm                       # default internal table size flag
var INCLUDES=−I{EM}/include             # use {EM}/include for #include files
name asld                               # Assembler/loader
    from .k.m.a                         # accepts compact code and archives
    to e.out                            # output file name
    program {EM}/lib/em_ass             # load file pathname
    mapflag −l* LNAME={EM}/{LIB}*       # e.g. −ly becomes
                                        #{EM}/mach/int/lib/tail_y
    mapflag −+* ASS_F={ASS_F?} −+*  # recognize −+ and −−
    mapflag −−* ASS_F={ASS_F?} −−*
    mapflag −s* SIZE_FLAG=−s*           # overwrite old value of SIZE_FLAG
    args {SIZE_FLAG} \
      ({RTS}:.c={EM}/{RT}cc) ({RTS}:.p={EM}/{RT}pc) −o > < \
      (.p:{TAIL}={EM}/{LIB}pc) \
      (.c:{TAIL}={EM}/{LIB}cc.1s {EM}/{LIB}cc.2g) \
      (.c.p:{TAIL}={EM}/{LIB}mon)
                                        # −s[sml] must be first argument
                                        # the next line contains the choice for head_cc or head_pc
                                        # and the specification of in- and output.
                                        # the last three args lines choose libraries
    linker
end
```

The command *ack −mem22 −v −v −I../h −L −ly prog.c* would result in the following calls (with exec(II)):

1) /lib/cpp −I../h −I/usr/em/include −Dem22 −DEM_WSIZE=2 −DEM_PSIZE=2 \
   −DEM_SSIZE=2 −DEM_LSIZE=4 −DEM_FSIZE=4 −DEM_DSIZE=8 prog.c
2) /usr/em/lib/em_cem −Vw2i2p2f4s2l4d8 −l
3) /usr/em/lib/em_ass −sm /usr/em/lib/em22/head_cc −o e.out prog.k
   /usr/em/lib/em22/tail_y /usr/em/lib/em22/tail_cc.1s
   /usr/em/lib/em22/tail_cc.2g /usr/em/lib/em22/tail_mon