`/home/ted/awk_stuff/awk.language.summary`
Prepared, on the basis of *The AWK Programming Language* by Aho, Kernighan & Weinberger,
by Ted Harding (original 01/12/2001; mods: 20/08/2002, 07/10/2004, 24/08/2006, 07/06/2007, 17/07/2007, 19/04/2008, 09/09/2010)

The *AWK* Crib

Basic statement

pattern **{** *action* **}**

or *function definition*

Pattern Summary

Summary of Patterns

1. **`BEGIN {`** *statements* **`}`**
   The *statements* are executed once, before any input has been read.
   `BEGIN` does not combine with any other pattern, and **does** require an action.
2. **`END {`** *statements* **`}`**
   The *statements* are executed once, after all input has been read.
   `END` does not combine with any other pattern, and **does** require an action.
3. *expression* **{** *statements* **}**
   The *statements* are executed at each input line where the *expression* is true (non-zero or non-null).
4. **`/`***regular expression***`/ {`** *statements* **`}`**
   The *statements* are executed at each input line matched by the *regular expression*.
   `!/`*regular expression***`/ {`** *statements* **`}`**
   The *statements* are executed at each input line **not** matched by the *regular expression*.
5. *compound pattern* **{** *statements* **}**
   A *compound pattern* combines expressions with **`&&`** (AND), **`||`** (OR), **`!`** (NOT) and parentheses; the *statements* are executed at each input line where the *compound pattern* is true.
6. $pattern_1$ **`,`** $pattern_2$ **{** *statements* **}**
   A **range pattern** matches each input line, starting from a line matched by $pattern_1$ and ending with the next line matched by $pattern_2$ (both inclusive); the *statements* are executed at each such line.
   A **range pattern** can not be part of any other pattern.
7. **{** *statements* **}**
   The **missing pattern** matches every input line (including blank lines), so the *statements* will be executed for every line.
8. *pattern*
   A pattern with no explicit **{** *action* **}** has an implicit action which simply outputs the whole of any input line matched by *pattern*.

Expressions as Patterns

Any *expression* can be used as a *pattern*. If it evaluates to **non-zero** or **non-null** for a line, then the line matches and any actions are executed.

Expressions and Operators

Any *expression* can be used as **operand** for any **operator**.

Expressions and Operators

Expressions

1. The primary expressions are
   numeric and string constants;
   variables;
   fields;
   function calls;
   array elements.

Operators

2. The following operators combine expressions

the assignment operators	`=` `+=` `-=` `*=` `/=` `%=` `^=`
the conditional expressions operator `? :`	*cond* `?` $expr_1$ `:` $expr_2$
the logical operators	`||` `&&` `!`
the substring matching operators	`~` `!~`
the relational operators	`<` `<=` `==` `!=` `>=` `>`
the string concatenation operator	`blank`
the arithmetic operators	`+` `-` `*` `/` `%` `^`
unary arithmetic	`+` `-`
prefix and suffix increment and decrement operators	`++` `--`
parentheses for grouping	

String-Matching Patterns

String-Matching Patterns

1. `/`*regexpr*`/`
   Matches whenever the current line contains a **substring** matched by *regexpr*.
2. `!/`*regexpr*`/`
   Matches whenever the current line **does not** contain a **substring** matched by *regexpr*.
3. *expression* `~` *regexpr*
   expression `~` $expression_2$
   Matches whenever the string value of *expression* contains a substring matched by *regexpr*, or equal to the string value of $expression_2$.
4. *expression* `!~` *regexpr*
   expression `!~` $expression_2$
   Matches whenever the string value of *expression* does not contain a substring matched by *regexpr*, or equal to the string value of $expression_2$.

Regular Expressions

Metacharacters
\ ^ . [ ] | ( ) * + ?

Expression	*Matches*
c	the non-metacharacter *c*
c	escape sequence, or literal *c* **\b** backspace **\f** formfeed **\n** newline **\r** carriage return **\t** tab *ddd* octal value *ddd* *c* any other character *c* literally
^	beginning of string
$	end of string
.	any character
[*c*$_1$*c*$_2$…**]**	any character in *c*$_1$*c*$_2$…
[^*c*$_1$*c*$_2$…**]**	any character not in *c*$_1$*c*$_2$…
[*c*$_1$ – *c*$_2$**]**	any character in the range *c*$_1$ – *c*$_2$
[^*c*$_1$ – *c*$_2$**]**	any character not in the range *c*$_1$ – *c*$_2$
r$_1$**|***r*$_2$	any string matched by *r*$_1$ or by *r*$_2$
(*r*$_1$**)(***r*$_2$**)**	any string *xy* where *r*$_1$ matches *x* and *r*$_2$ matches *y*
(*r***)***	zero or more consecutive strings matched by *r*
(*r***)+**	one or more consecutive strings matched by *r*
(*r***)?**	zero or one string matched by *r*
(*r***)**	any string matched by *r*

Actions

An *action* consists of one or more *statements* separated by newlines or semicolons.

Action Statements

1. *expression*
2. **print** *expression-list*
   print(*expression-list***)**
3. **printf** *format***,** *expression-list*
   printf(*format***,** *expression-list***)**
4. **if (***expression***)** *statement*
5. **if (***expression***)** *statement* **else** *statement* [**else** associated with most recent unmatched **if**]
6. **while (***expression***)** *statement*
7. **for (***expression***;** *expression***;** *expression***)** *statement*
8. **for (***variable* **in** *array***)** *statement*
9. **do** *statement* **while (***expression***)**
10. **break**
11. **continue**
12. **next**
13. **exit**
14. **exit** *expression*
15. **{** *statements* **}**

User-defined Variables

Named user-defined variables come into existence on being mentioned for the first time. They have default null initial values of **""** (in a string context) or **0** (in a numerical context). Variables can also be defined in the command-line.

Variables acquire values by assignment. The value of a string variable is automatically converted to numeric type (if this makes sense†) if the variable is used in a numerical context. The value of a numeric variable is automatically converted to string type‡ if the variable is used in a string context.

† If not, the numeric value used is **0**, the default numeric null initial value.
E.g. **{i="hello";printf("%5.5f ",i); print i}** → **0.00000 hello**
The numeric value of a string is the value of its longest initial numerically compatible sequence,
e.g. **"0.12abc"** → **"0.12"** → **0.12**, **"abc"** → **""** → **0**, **1E2G3H4"** → **"1E2"** → **100**

‡ The string value is its representation according to **OFMT** (default **"%.6g"**),
e.g. **100/3** → **"33.3333", 100000000/3** → **"3.33333e+07".**

Functions use **local** copies of variables named in their argument lists. All other variables (including variables defined within functions) are **global**; variables arising within a function which are intended to be local can be so coerced by being named (with default null values) in the argument list of the function definition.

Built-in Variables

Variable	*Meaning*	*Default*
ARGC	number of command-line arguments	
ARGV	array of command-line arguments	
ARGV[*i***]**	*i*th command-line argument	
FILENAME	name of current input file	
FNR	current record number in current file	
FS	field separator (character or string)	**" "**‡
NF	number of fields in current record	
NR	number of records read so far	
OFMT	output format for numbers	**"%.6g"** (i.e. 6 significant figures)
OFS	output field separator	**" "**
ORS	output record separator	**"\n"**
RLENGTH	length of string matched by **match( )**	
RS	input record separator	**"\n"**
RSTART	start of string matched by **match( )**	
SUBSEP	subscript separator (**ctrl-** = **^**)†	**"\034"**

† For the **awk** 'hack' of multidimensional arrays (See the A-K-W book pp. 52–3, and below)

‡ Multiple **FS="** …**"** treated as one **" "**; for **FS="[ ][ ]**…**"**, each **" "** is significant. String **FS** is treated as regular expression (leftmost longest non-overlapping match). **FS="[ ][ ]*|[\t]|[;]|[,]"** will cause one or more spaces, or **TAB**, or **";"**, or **","**, to be recognised as a field separator. **FS** can be a string, e.g. **")("** is set up by **FS="[)][(]"**

Field Variables

Variable	*Meaning*	*Default*
$0	The whole line	
$1, **$2**, …	The fields of the line	
$(*i***)**	Field *i* (dynamic index)	
$*var*	E.g. **{Fno=25 ; print($Fno)}** (dynamic index)	
$(NF+1) = ...	Defines an additional field; increments **NF**	

Functions

There are several **built-in** functions. The user may define any number of **user-defined** functions.

User-defined Functions

Such a function is defined by a *function definition* statement (declaration) of the form

`function` *name* **`(`** *parameter-list* **`) {`** *statements* **`}`**

which may occur anywhere a *pattern-action* statement can. See above for the distinction between **local** and **global** variables. Example:

```
function   log10(x) { return log(x)/log(10) }
```

In calling a function there must be no space between the name and the opening parenthesis of the parameter-list (a space is permissible in the definition).

Functions may be defined recursively. Internal variables used in recursive functions need to appear in the parameter list of the function declaration (so that they will be *local* at every depth; otherwise they are *global* and the function will misbehave).

Built-in Arithmetic Functions

FUNCTION	VALUE RETURNED
`atan2(`*y*`,`*x*`)`**	arctangent of y/x in the range $-\pi$ to π (radians)
`cos(`*x*`)`**	cosine of x (x in radians)
`exp(`*x*`)`**	exponential function of x, e^x
`int(`*x*`)`**	integer part of x; truncated towards 0 when $x > 0$
`log(`*x*`)`**	natural (base e) logarithm of x
`rand()`	random number† r, uniform real on $0 \le r < 1$
`sin(`*x*`)`**	sine of x (x in radians)
`sqrt(`*x*`)`**	square root of x
`srand(`*x*`)`**	x (unsigned integer) is new seed† for **`rand()`**

† **`rand()`** starts, by default, from the same value (**`1`**) every time **`awk`** is run.
`srand(`*x*`)`** starts **`rand()`** from the value x; **`srand()`** sets x from the system clock.
`srand` returns the *current* (**pre-call**) seed; to get (for re-use) the seed set by **`srand()`** do
`srand(); i=srand(); print i; srand(i);`

Built-in String Functions

FUNCTION	ACTION, AND VALUE RETURNED
`gsub(`*r*`,`*s*`)`**	substitute *s* for regexp *r* globally in **`$0`**; *return* number of substitutions made
`gsub`(*r*`,`*s*`,`*t*`)`**	substitute *s* for regexp *r* globally in string *t*; *return* number of substitutions made
`index(`*s*`,`*t*`)`**	*return* first position of string *t* in *s*, or 0 if not present
`length(`*s*`)`**	*return* number of characters in *s*
`match(`*s*`,`*r*`)`**	test whether *s* contains a substring matched by regexp *r*; *return* index or 0; sets **`RSTART`** and **`RLENGTH`** (leftmost longest†) *e.g.* **`match($0,/[0-9]+\.htm/)`**
`split(`*s*`,`*A*`)`**	split *s* into array *A* on **`FS`**;‡ *return* number of fields
`split(`*s*`,`*A*`,`*fs*`)`**	split *s* into array *A* on field separator *fs*;‡ *return* number of fields
`sprintf(` *fmt*`,`*expr-list*`)`**	*return expr-list* formatted according to *fmt*
`sub(`*r*`,`*s*`)`**	substitute *s* for leftmost longest† substring of **`$0`** matched by regexp *r*; *return* number of substitutions made
`sub(`*r*`,`*s*`,`*t*`)`**	substitute *s* for leftmost longest† substring of *t* matched by regexp *r*; *return* number of substitutions made
`substr(`*s*`,`*p*`)`**	*return* last part of *s* starting at position *p*
`substr(`*s*`,`*p*`,`*n*`)`**	*return* substring of *s* of length *n* starting at position *p*

† First the leftmost match is found; then it is extended as far as possible
‡ Index values in **`A`** are integers **`1,2,`**…; Element values **`A[i]`** are substrings of *s*

Arrays

Arrays are *associative arrays* whose subscripts (index values) are *strings*. Like variables, array elements come into existence by being mentioned, and have default null values. Array elements are assigned values by

array`[`*subscript*`] =` *expression*

The construct

`for (` *i* `in` *arrayname* `) {` … `}`

assigns successively to *i* the string-valued subscripts of the elements of the array whose name is *arrayname*. Example (a complete **awk** program):

```
BEGIN{ common_name["Clupea harengus"] = "herring" ;
       common_name["Anguilla anguilla"] = "European eel" ;
       common_name["Gadus morhua"] = "cod" ;
       common_name["Salmo salar"] = "Atlantic salmon" ;
     } ;
     { for ( bio_name in common_name )
       { print bio_name " is commonly called the " common_name[bio_name] }
     }
```

whose output will be:

```
Clupea harengus is commonly called the herring
Anguilla anguilla is commonly called the European eel
Gadus morhua is commonly called the cod
Salmo salar is commonly called the Atlantic salmon
```

The order in which the index values are assigned to *i* is not necessarily predictable, especially if new elements are added to an existing array.

The expression

subscript `in A`

has value `1` if `A[`*subscript*`]` exists, `0` otherwise, and does not create `A[`*subscript*`]` if it does not exist, as in

```
if ( "Leuciscus rutilus" in common_name )
```

The statement

`delete` *arrayname*`[`*subscript*`]`

deletes the specified element from the array; `for (i in A) delete A[i]` deletes all elements of `A`.

The subscript of an array can have a "pseudo-multidimensional" form, and **awk** performs an internal concatenation of a comma-separated index list to generate an internally synthesised string-valued subscript: if $i = 1$, $j = 5$, $k = 2$, then the statement

```
A[i, j, k] = "one-five-two"
```

generates the array element

A`["1•5•2"] = "one-five-two"`

where "•" stands for the default value `\034` of `SUBSEP`, the subscript-component separator. Such a subscript value is accessible (e.g. by '`for i in A`'), and can be decomposed into its components using the `split` function:

```
{A[1, 5, 2] = "one-five-two"}; {for (i in A) {print i} } → 1•5•2
{for ( i in A ) {split(i,B,SUBSEP);for(j in B) print B[j]} } → 1
                                                               5
                                                               2
```

Operation

It is important to note that **awk** *always* operates by applying its program to each input line, and generating appropriate output each time. If no input lines are supplied, nothing will happen. For example, let **{...}** denote the above **awk** program for fish names. Then the command line

```
awk '{...}'
```

will apparently hang. In fact, **awk** is waiting for input lines and will generate appropriate output for any lines which are matched (as, in this case, all are). Therefore, nothing will happen unless **Return** is pressed, but the above output will be generated every time it is pressed; press **^D** (**EOF**) to stop it.