7. Unix Utilities

The output of the above programs are ideal for use with the standard unix utilities such as `egrep, cut, join` and `nawk`. These may also be used to query the data files directly although this is not very efficient. For example, the following stores all HIP identifiers of entries in hip_main with a DSS chart in file hip.DSS:

```
cut -f2,70 -d "|" hip_main.dat | egrep D| cut -f1 -d"|"  >hip.DSS
```

On a sparc 20 this pipeline took over 5 minutes.

The resulting file may be used for further operations. For example to get a verbose print of each of these HIP entries the following could be typed:

```
shipmain 'cat hip.DSS'
```

However only some fields may be required. Because it is a verbose print each field is labelled so they may be selected by name using `egrep`. Hence to get `H1` plus all other fields with identifier in the description use:

```
shipmain 'cat hip.DSS' | egrep -i "H1 |identifier" > hipG.ids
```

This pipeline took under 10 seconds to run on the same machine.

These commands are very flexible. To use them properly consult the corresponding man pages. Perl is another scripting language which combines all of these features but is not installed as standard on all systems.

7.1 cut

The unix cut command is equivalent to the relational project operation. It allows one to vertically select columns from a file. So in the above example 'cut -f2,70' retrieves the second and 70th column from hip_main.dat. Note these are H1 and H69: cut does not know about column names, and it only works on column number. Hence for tyc_main and hip_main to get Tn or Hn cut -f(n+1) since tyc_main and hip_main start at field 0. The default field separator for cut is a tab, while the data files provided use '|'. Hence when using cut the option '`-d "|"`' must be used.

7.2 egrep/grep

Grep and egrep are similar, the main difference being that egrep allows more complex expressions. These functions provide functionality similar to the relation select operation. They search each line of a file to see if it matches the provided regular expression. Hence in the first example above 'egrep D' will match any line with the letter 'D' in it. Since these functions are not aware of field boundaries, this must be built into the expression.

'|' is a special character for egrep; hence it must be 'escaped' in expressions (it is the 'or' function otherwise, and will give unexpected results). Likewise '.' is a special character which matches any character in an expression. So, for example, to get entries with parallax > 9 and Johnson V magnitude = 4.nn an expression such as the following is required:

```
cut -f2,6,12 -d"|" hip_main.dat | \
egrep "^ *[0-9]+\| 4...\| *( 9|[1-9][0-9])"
```

Consider this expression in detail. First of all the cut gets the fields of interest i.e. H1 H5 and H11. Next the grep expression:

^	matches the beginning of sentence ($ matches the end of sentence).
*	matches zero or more occurrences of the character occurring before it: in this case any number of spaces.
[0-9]	The '[]' matches one character within the brackets. Number and character ranges may be specified, or individual characters or numbers. This example matches any single digit 0 to 9.
+	is like * but expects at least one occurrence of the character which appears before it.
|	to match the '|' field delimiter character it must be 'escaped' like this.
4	match a space followed by the digit 4.
...	match any 3 characters.
()	closing an expression in braces allows a sub expression to be constructed. Basically this expression says match a 9 or any number with two or more digits starting with the digit 1 through 9.
|	the 'or' operator.

Hence this says: start at the beginning of the line, skip any spaces then look for a number (any number) followed by the character '|', then look for a 4 followed by any 3 characters followed by '|'. Next skip any spaces and look for a number greater than 9. If grep at any time fails one of these tests then the line does not match the expression and will not be printed.

7.3 nawk

Nawk is a powerful context based script language. For example, to do something similar to the above nawk could be used as follows:

```
nawk 'BEGIN {FS="|"} \
$6~" +4..." {  if ($12 >= 9) {print $2"|"$4"|"$5"|"$6"|"$12 }}' hip_main.dat
```

`BEGIN` is a special tag which means this statement is only executed before reading the input. `FS` is the field separator.

The second statement says only for lines where $6 is 4.nn then if $12 is also at least 9, print columns 2 4 5 6 and 12.

7.4 perl

One problem with the above unix utilities is that it is often necessary to write scripts using all of the tools in complex pipe lines. The perl scripting language combines all of the

above features in a very fast language. For example to again perform the above selection the perl program would be as follows:

```
#!/usr/local/bin/perl -w

$infile="hip_main.dat";
#Open infput file
open(INFILE,"<$infile") || die ("Could not open $infile.");
# loop through file
while (<INFILE>)
{
   #Split line up using seperator
   @tline=split(/\|/);
   printf "%s|%s|%s|%s|%s\n",$tline[1],$tline[3],$tline[4],
                          $tline[5],$tline[11]
   if ($tline[5] =~ / +4.../ && $tline[11] >= 9)
}
close(INFILE);
```

The `split` operator allows a line of input to be split into fields according to a field separator which is a regular expression. Note that arrays start at reference zero hence to access H5 of the above line `$tline[5]` is used. Since there is no ';' after the `printf` statement it is only executed if the following if statement is true. In the if statements both numerical and regular expression operators have been used to provide the same match as in the previous examples.

With perl there is also lots of flexibility to do other interesting tasks, the following script computes the summary information given on certain fields in Section2.2 of Volume 1.

```
#!/usr/local/bin/perl -w
#Find number of occurences of each possible value in a given
#set of fields in the tyc_main file.
$infile="tyc_main.dat";

#Open infput file
open(INFILE,"<$infile") || die ("Could not open $infile.");
$RECS=0;
#The field numbers of the fields we want to do a statistic on
@FIELDS=(2,6,7,10,36,39,40,42,47,48,49,50,57);

##############################################
# loop through file
# Look at FIELDS count values accordingly
while (<INFILE>)
{
     # Dump "\r\n" from line
     chop;
     chop;

     #Split line up using seperator
     @tline=split(/\|/);
     $RECS +=1;

     #Look at each field we are interested in
     foreach $field (@FIELDS)
     {
        #Contruct Key name to be fieldname_value for the associative array
        $NAME=sprintf("T%s_%s",$field,$tline[$field]);
        #Increment this variable by 1 in the associative array %sums
        $sums{$NAME}++;
     }

}
close(INFILE);
#Print the results
&printInfo;

sub printInfo
{
   printf "Done %6d recs at TYC %s\n", $RECS, $tline[1];
   foreach $key (sort keys %sums)
   {
      printf "%8s     %6d\n",$key,$sums{$key};
   }
}
```