Investigating the Design and Implementation of Geographic Interfaces

Richard T. Pascoe

Department of Computer Science, University of Canterbury

June 1988

ACKNOWLEDGEMENT

The author would like to acknowledge the contribution made by his supervisor, J. P. Penny, during the completion of this research

Table of Contents

1. Introduction

The original objective set by LINZ was to design a modular software package to enable data conversion between the proposed New Zealand Transfer File Format standard for digital cartographic data and other transfer file formats. However it was found to be more practical to specify the architecture of the geographic interface and apply existing software tools to implement modules within that architecture. A geographic interface is produced by integrating these modules.

Interfaces have the same underlying architecture;

1) read the source data

2) reorganise the source data elements into the target data structure

3) write the data out in the target format

and a set of modules can be specified according to this architecture;

1) The Source Decoder

2) The Conversion Manager

3) The Target Encoder

The approach taken here involves

1) Specifying transfer file formats:

   A) a formal definition of the source and target transfer file formats using Backus-Naur Form (BNF) as suggested by van Roessel et al [*1986*]

   B) a relational data model for the source and target transfer file formats

2) Applying the formal definition of the source transfer file format with the unix software tools Yacc [*Johnson 1979*] and Lex [*Lesk et al 1979*] to generate the source decoder

3) Using the Ingres relational database management system [*Stonebraker et al, 1976*] to reorganise data from the source relational data model into the target relational data model

This report assumes that the reader is familiar with BNF, the unix software tools Yacc and Lex, and the embedded quel and relational operators provided by Ingres (though Yacc and Lex, and Ingres were used here, other equivalent software tools can also be used).

In taking this approach the solution that is attained is far more powerful as it provides a precise design methodology for implementing a geographic interface between any two transfer file formats. A detailed description explaining the reasons why such an approach is justified and the description of a completed interface will be available when my thesis is published.

This report provides a practical demonstration of how a source decoder module may be implemented for the Colourmap transfer file format, and investigates the suitablity of the approach taken if the American Spatial Data Transfer Specification (SDTS) [*SDTS Vol1, Vol2, 1988*] was the target transfer file format.

2. Implementing Modules for a Geographic Interface

To demonstrate the approach, the Colourmap transfer file format [*CSIRONET, 1986*] was selected as the source transfer file format because large sets of data were available to test the interface as it was developed.

2.1 Implementing a Source Decoder

The source decoder extracts data from the Colourmap transfer file format and places it into a relational database managed by Ingres. The source decoder is divided into two components: the parser, and the lexical analyser. The two components are implemented using the unix software tools Yacc and Lex respectively. To generate the source decoder the following must be specified:

1). a BNF grammar for the Colourmap transfer file format

2). a source relational data model into which the data is to be placed

2.1.1. A BNF Grammar for Colourmap

Appendix A contains a BNF grammar of the three major file types used in the Colourmap transfer file format: zone files, line overlays, and standard attribute files. It is anticipated that there would be no problem with the remaining files, and so they are ignored. The grammar given in Appendix A is provided using the notation required by Yacc.

Also contained in Appendix A is the set of token definitions supplied to Lex which generates the lexical analyser to be used by the source decoder. Not all aspects of the Colourmap transfer file format could be defined using a BNF grammar. These exceptions are described in the following sections.

2.1.1.1. Recognising Enumerated Repeating Groups

Within the Colourmap transfer file format, there are data elements, ie. numbers, that specify the number of data objects, eg. coordinate pairs, contained within the datafile. Such a structure is referred to as an enumerated repeating group (ERG). The syntax of an ERG can be defined using BNF:

```
ERG.:                number.of.objects
                     data.objects

data.objects:        data.object data.objects
```

To illustrate an ERG consider an enumerated list of coordinate pairs. The syntactic definition of this ERG would be expressed using BNF as:

```
coordinate.list:     number.of.coordinates
                     coordinates

coordinates:         coordinate.pair
                |    coordinates coordinate.pair

coordinate.pair:     x y
```

The concept that the value of number.of.coordinates specifies the number of coordinate pairs in the list cannot be explicitly defined using a BNF grammar; thus, to incorporate this concept into the source decoder generated by the Yacc and Lex software tools the following technique is used.

Yacc permits an action to be performed before parsing of a production in the grammar is completed. This facility is used to set a counter to the number of data objects in the repeating group. The BNF grammar for the colourmap transfer file format is augmented with an End Of Repeating Group (EORG) token. The above illustration of an enumerated group of coordinate pairs would be specified in the definition of the source decoder as follows

```
coordinate.list:     number.of.coordinates
                     {INITRG(number.of.coordinates);}
                     coordinates EORG

coordinates:         coordinate.pair
                |    coordinates coordinate.pair

coordinate.pair:     x y
                         { DECRG; }
```

Each time the lexical analyser is called to supply a token to the parser, the lexical analyser checks to see if a repeating group is being parsed, and if so whether the last data object in the group has been found. If the last data object in the group has been recognised then the lexical analyser generates an EORG token. Otherwise it continues reading the datafile and generating tokens in the usual manner. Each time the parser recognises a data object, the counter is decremented.

Appendix B lists the data structures, macros, and alterations to the definition of the lexical analyser, necessary to implement this technique, and examples of its use can be found in the complete listing of the definition files for the Colourmap source decoder (Appendix D).

2.1.1.2. Explicit Sequencing of Data Objects within Enumerated Repeating Groups

Associated with an ERG is the implicit sequence in which the data objects occur within the repeating group. When decoding such a group and placing the data objects into the source relational data model, the source decoder may have to provide an explicit sequence number for each data object in the repeating group. The sequence number may be required to comply with the non-ordering property of the relational model as specified by Codd [*Codd, 1970*].

When using the Yacc and Lex software tools to generate the source decoder it is possible to label each data object with its sequence number in the following way:

Yacc provides a facility for associating a value with a terminal or non-terminal symbol in the grammar. By setting the value returned by the non-terminal symbol representing the data object to be the sequence number of that data object and then using that value to label the next data object the necessary sequencing of data objects can be achieved.

Consider the ERG of coordinate pairs used in the previous section. To explicitly sequence the coordinate pairs the following would be used:

```
1) coordinate.list: number.of.coordinates
                    {INITRG(number.of.coordinates);}
                    coordinates
                    EORG

2) coordinates:     coordinate.pair
                        { $$ = 1; }
                  |  coordinates coordinate.pair
                        { $$ = $1 + 1 }

3) coordinate.pair: x y
                        { DECRG; }
```

In rule 2 the sequence number associated with the non-terminal symbol coordinates is that of the next coordinate pair in the sequence.

2.1.1.3 Controling lexical analysis of Datafiles

In the Colourmap transfer file format specification the value of some data elements is a character string. Because a character string may contain alphabetic characters, spaces, and digits without being enclosed by quotation marks the size of a character string is the only way of ensuring that the correct number of characters are associated with a data element.

Data elements with character values come in different sizes depending on what the data element is: a comment is 80 characters long, a zone name is 10 characters long, and the site name or annotation for a point is 40 characters long. A technique for enabling the parser to instruct the lexical analyser on the size of the next character string (a STRING token) is defined as follows:

Yacc provides the facility of performing an action before a rule is completely recognised; thus, the size of the required character string can be set (the value of the variable called datasize) before the data element in a rule. For example a rule for the zone identifier with a character string size of 10 is;

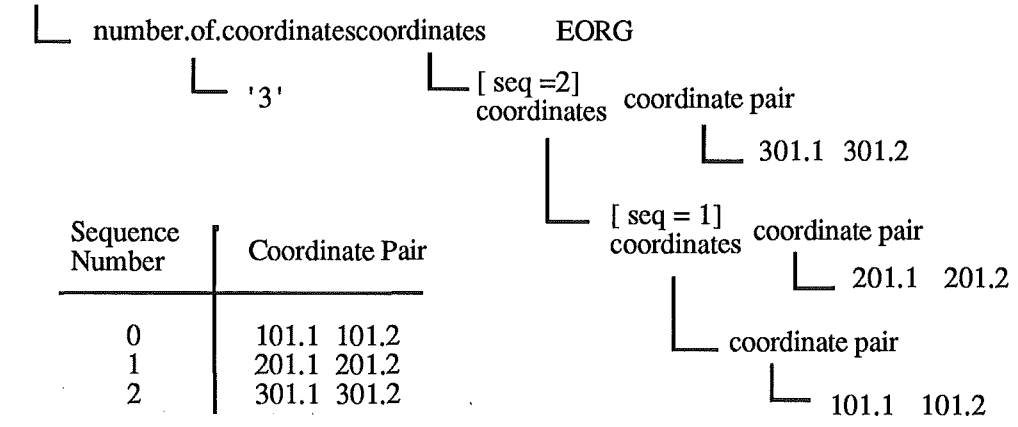
Figure 1

An Example of Sequencing Data Ojects within an Enumerated Repeating Group

Consider the enumerated repeating group of coordinate pairs:

3 101.1 101.2 201.1 201.2 301.1 301.2

The parse tree generated, along with the sequence number associated with each non-terminal symbol 'coordinates' would be:



Sequence Number	Coordinate Pair
0	101.1 101.2
1	201.1 201.2
2	301.1 301.2

```
_zone.id_:                    { datasize = 10;}
                          string
```

The lexical analyser uses the value assigned to the variable datasize to determine how many characters to read in for the value associated with the string token generated.

There is a complication with this technique: before the action is performed the next token must have been returned by the lexical analyser, ie the string token would have to be read in before the action setting its size is performed.

To solve this problem, the rule

```
string:                   SETSIZE STRING
```

for character strings is defined so that the token (SETSIZE) following the action which sets the size of the character string will always be generated by the lexical analyser before the STRING token with its associated character string is returned by the lexical analyser. In effect, the lexical analyser sends a message (the SETSIZE token) to the parser requesting more information before returning a STRING token. Incorporating the rule for character strings with the above example using the zone identifier would result in;

```
_zone.id_:                    { datasize = 10; }
                          SETSIZE STRING
```

When the lexical analyser recognises a sequence of characters that match the regular expression for a STRING token, it checks to see whether the size of the character string has been set by the parser. If it is, the lexical analyser takes the number of characters required leaving any excess characters for the next token, otherwise the lexical analyser leaves all of the matched characters for the next call and generates a SETSIZE token.

2.1.2 Relational Data Model for Colourmap

Design of the relational data model for the Colourmap transfer file format is based on the I2 structure [*van Roessel et al, 1986*], and the data elements contained within records of the transfer file format. Figure 2, an informal entity-relationship diagram, defines the data model used and Appendix C contains a set of Quel commands to create the corresponding Ingres relational database.

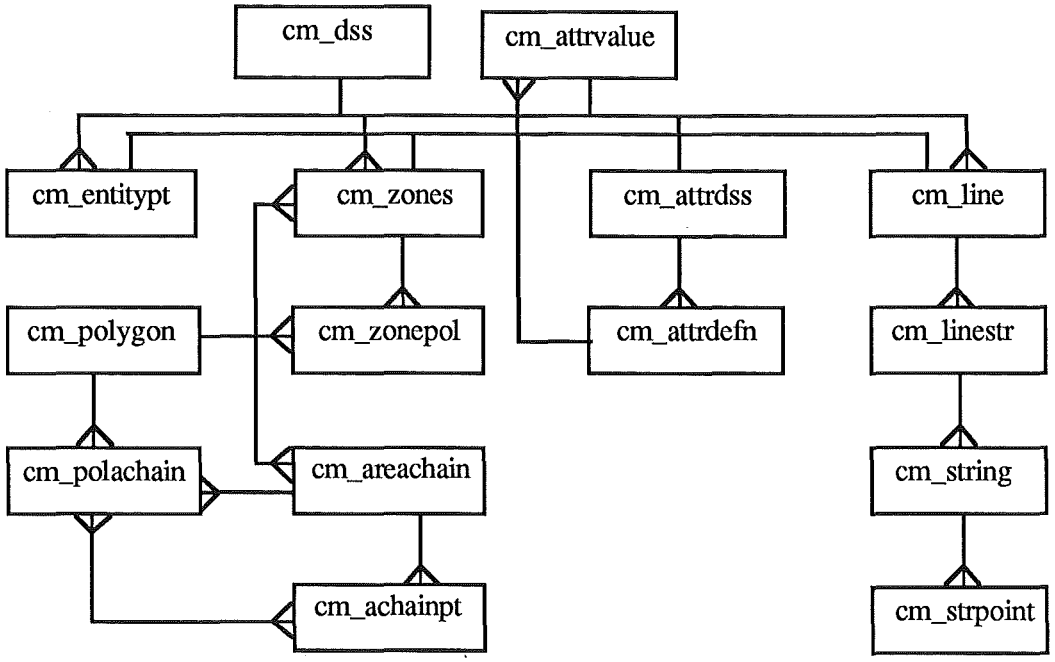
2.2 Implementing a Conversion Manager and a Target Encoder

As it is likely that the American SDTS will become a much used exchange medium investigation was made into converting data from the Colourmap relational model into the relational form of the American Spatial Data Transfer Specification.

The investigation indicates that the conversion can be achieved easily using relational operators, as suggested by Penny [*Penny, 1986*], because the two data models are very similar and, while unproven, it is expected that for transfer file formats using a coordinate, vector representation this would be true for many source and target transfer file format pairs.

Figure 2

The Colourmap Relational Data Model



Implementing a target encoder using the ISO 8211 specification [*ISO 8211-1985*] for a data descriptive file for information interchange as required by SDTS should not prove to be difficult.

3 Conclusions

The development time for implementation of a new geographic interface should be greatly reduced by dividing the interface into modules, and using existing software tools such as Yacc and Lex and a relational database management system like Ingres to implement these modules.

The approach of using relational data models as the intermediate exchange data structures simplify the conversion process for comparable source and target transfer file formats.

References

Codd 1970 — E. F. Codd, "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, Volume 13, Number 6, pp 377 - 387

CSRIONET 1986 — "CSIRONET Reference Manual No 19, Command Driven Colourmap, User's Guide, Version 1" Graphics System Section, CSIRONET.

ISO 8211-1985 — BS 6690 : 1986 / ISO 8211-1985 "A data descriptive file for information interchange", British Standards Institution, July 1986

Johnson, 1979 — S. C. Johnson, "Yacc: Yet Another Compiler-Compiler", Unix Time-Sharing System: Unix Programmer's Manual, Seventh Edition, Volume 2B, January, 1979

Lesk *et al* 1979 — M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator", Unix Time-Sharing System: Unix Programmer's Manual, Seventh Edition, Volume 2B, January, 1979

Martin *et al* 1986 — J. Martin, C. McClure "Diagramming Techniques for Analysts and Programmers", published by Prentice Hall

Penny 1986 — J. P. Penny, "Relational methods for format conversion of map data", Cartography, Volume 15, Number 1, March 1986

SDTS Vol 1 1988 — Spatial Data Transfer Specification, "Volume 1 - The Standard", The American Cartographer, Volume 15, Number 1, pp 21 -, January 1988

SDTS Vol 2 1988 — Spatial Data Transfer Specification, "Volume 2 - Initial Entity and Attribute Definitions", The American Cartographer, Volume 15, Number 1, pp 144 -, January 1988

Stonebraker *et al* 1976 M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES", ACM Transactions on Database Systems, volume 1, number 3, pages 189-222

van Roessel *et al* 1986 J. van Roessel, D. Bankers, V. Connochioli, S. Doescher, G. Fosnight, M. Wehde, and D.Taylor, "Vector Data StructureConversion at the EROS Data Center", Final Report, Phase I, (Draft)

Appendix A

The BNF specification of selected colourmap datafiles

The following BNF description (without the EORG and SETSIZE tokens) recognises three types of colourmap data files:

1) Zone files

2) Line overlay files

3) Standard attribute files

The notation used in the BNF description that follows is:

lexical.symbol ..... Lexical symbol

TOKEN ..... a token returned by lex

non-terminal ..... yacc non-terminal symbol

/** Comment **/ ..... description of production

```
datafile :  _comment_ NEWLINE header data
      ;

data :            /** Standard attribute file **/
            names
            attribute.sectn

      |            /** Zone file **/
            zone.part
            poly.part
            seg.part

      |            /** line overlay **/
            seg.part
      ;
```

```
zone.part : ZONES NEWLINE
            _num.zones_ NEWLINE
            zone.defn.list
            ;

poly.part :
      |     POLYGONS NEWLINE
            _num.polygons_ NEWLINE
            polygon.defn.list

seg.part :
      |     SEGMENTS NEWLINE
            _num.segments_ NEWLINE
            segment.defn.list
      ;

map.window :
            _X.min.regn_ _X.max.regn_ _Y.min.regn _Y.max.regn_
            NEWLINE
      ;

map.projection :
            _map.proj.code_ _zone.type_ NEWLINE
      |     _map.proj.code_ NEWLINE
      ;

zone.defn.list :
            z.defn
      |     zone.defn.list z.defn
      ;

polygon.defn.list :
            poly.defn
      |     polygon.defn.list poly.defn
      ;
```

```
segment.defn.list :
            segment.defn
      |     segment.defn.list segment.defn
      ;
z.defn :    _zone.id_ _poly.in.zone_ NEWLINE
            poly.id.list NEWLINE
      ;

poly.defn : poly.defn.hdr
            _x.poly.label_ _y.poly.label_ _polygon.area_ NEWLINE
            seg.id.list NEWLINE

      |     poly.defn.hdr _x.poly.label_ _y.poly.label_ NEWLINE
            seg.id.list NEWLINE

      |     poly.defn.hdr _polygon.area_ NEWLINE
            seg.id.list NEWLINE
      ;

poly.defn.hdr :
            _poly.id_ _zone.id_ _segs.in.poly_ _dsp.level_ NEWLINE
            _x.min.poly.regn_ _x.max.poly.regn_
            _y.min.poly.regn_ _y.max.poly.regn_
      ;

segment.defn :
            _segment.id_ _seg.left.zone_ _seg.right.zone_
            _points.in.segment_ NEWLINE
            xy.defn.list NEWLINE
            ;

seg.id.list :
      |     seg.id.list NEWLINE

      |     seg.id.list INTEGER
      ;
```

```
poly.id.list :
      |     poly.id.list NEWLINE

      |     poly.id.list INTEGER
      ;

xy.defn.list :
            REAL REAL REAL REAL

      |     xy.defn.list NEWLINE

      |     xy.defn.list REAL REAL
      ;

header :    MAP NEWLINE map.window map.projection

      |     _num.attr_ _num.names_ _missing.dat.val_ NEWLINE
      ;

names :     name.list
      ;

name.list :
      |     name.list _name_ NEWLINE

      |     name.list _name_
      ;

attribute.sectn :
            attribute.list
      ;

attribute.list :
      |     attribute.list attribute
      ;
attribute :       attr.defn attr.val.list
      ;
```

```
attr.defn :       _description_ _units_ NEWLINE
      ;

attr.val.list :
      |     attr.val.list _attr.value_ NEWLINE

      |     attr.val.list _attr.value_
      ;

/*-----------------------------------------------*/
/*          Lexicon definitions                  */
/*-----------------------------------------------*/

_attr.value_ :          REAL
      ;
_comment_ :             string
      ;
_polygon.area_ :        REAL
      ;
_description_ :         string
      ;
_dsp.level_ :           INTEGER
      ;
_map.proj.code_:        INTEGER
      ;
_poly.in.zone_ :        INTEGER
      ;
_segs.in.poly_ :        INTEGER
      ;
_points.in.segment_ :   INTEGER
      ;
_missing.dat.val_ :     REAL
      ;
_name_ :                string
      ;
_num.attr_ :            INTEGER
      ;
```

```
_num.segments_    :     INTEGER
      ;
_num.names_       :     INTEGER
      ;
_num.polygons_ :        INTEGER
      ;
_num.zones_ :           INTEGER
      ;
_poly.id_ :             INTEGER
      ;
_segment.id_ :          INTEGER
      ;
_seg.left.zone_   :     string
      ;
_seg.right.zone_ :      string
      ;
_units_ :               string
      ;
_X.min.regn_ :          REAL
      ;
_X.max.regn_ :          REAL
      ;

_x.max.poly.regn_ :     REAL
      ;
_x.min.poly.regn_ :     REAL
      ;
_x.poly.label_ :        REAL
      ;
_y.max.poly.regn_ :     REAL
      ;
_Y.max.regn_ :          REAL
      ;
_Y.min.regn_ :          REAL
      ;
_y.min.poly.regn_ :     REAL
      ;
```

```
_y.poly.label_ :        REAL
      ;
_zone.id_ :             string
      ;
_zone.type_ :           INTEGER
      ;
/*-------------------------------------------------*/
/*          General Constructs                     */
/*-------------------------------------------------*/

string      :           STRING
      ;
```

To complete the definition the following Lex regular expressions define the tokens in terms of ascii characters:

Token	Definition
INTEGER	[\t]+[+-]?[0-9]+[\t]*
MAP	MAP[\t]*\n
POLYGONS	POLYGONS[\t]*\n
REAL	[\t]*[+-]?[0-9]+"."[0-9]*([E][-+][0-9][0-9])?[\t]* | [\t]*[+-]?[0-9]*"."[0-9]+([E][-+][0-9][0-9])?[\t]*
SEGMENTS	SEGMENTS[\t]*
STRING	(([\,\-A-Za-z0-9'$#!%^&~`:?/]("."?)?)+[\t]*)+
ZONES	ZONES[\t]*\n

Appendix B

Data structures used to implement enumerated repeating groups

Var iable Declarations:

```
#define maxrgflg 5
#define INITRG(A) rgcnt[++rgflg] = A;
#define DECRG rgcnt[rgflg]--;
int rgcnt[maxrgflg], rgflg = 0;
```

Table B.1 indicates how the analyser can assertain the state of the Repeating Group Mechanism using the values of rgflg, and rgcnt[rgflg].

Table B.1

Condition	Value of rgflg	Value of rgcnt[rgflg]
Mechanism inactive	0	0
Within incomplete repeating group	non-zero	remaining number of data objects to be parsed
Repeating group completed	non-zero	0

The definition of the lexical analyser includes an alternative input macro that is to be used by the lexical analyser;

```
#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :
(((yytchar=yysptr>yysbuf ? U(*--yysptr): \
getc(yyin)) == 10 ? (yylineno++,yytchar): \
yytchar)==EOF ? 0 : yytchar))
```

When the input macro establishes that a repeating group has been completed an '@' character is inserted into the character stream. A token definition associating the '@' character to the EORG token is included within the set of token definitions supplied to Lex so that the '@' character inserted into the character stream will result in the desired EORG token to be generated by the lexical analyser. The '@' character is absorbed out of the input character stream by the lexical analyser to maintain data integrity.

Appendix C

Ingres Database for Colourmap Data Model

```
create cm_attrdefn (
    atid        =i4,
    description =c40,
    units       =c10
)
create cm_polygon  (
    polid       =i4,
    cmptachain  =i4,
    enclzoneid  =c11,
    dsplevel    =i4,
    xminregn    =f8,
    xmaxregn    =f8,
    yminregn    =f8,
    ymaxregn    =f8,
    xlabel      =f8,
    ylabel      =f8,
    area        =f8
)
create cm_zone     (
    zoneid      =c11,
    cmptpol     =i4
)
create cm_entitypt (
    siteid      =i4,
    x           =f8,
    y           =f8,
    description =c40
)
create cm_zonepol  (
    zoneid      =c11,
    polid       =i4
)
create cm_strpt    (
    achainid    =i4,
    seq         =i4,
    x           =f8,
    y           =f8
)
create cm_objname  (
    nameseq     =i4,
    name        =c11
)
create cm_attrvalue(
    atid        =i4,
    nameseq     =i4,
    value       =f8
)
```

```
create cm_attrdss  (
    nattributes =i4,
    nnames      =i4,
    missdatval  =f8,
    rfdssid     =i4
)
create cm_line     (
    lineid      =i4,
    cmptstr     =i4,
    xminregn    =f8,
    xmaxregn    =f8,
    yminregn    =f8,
    ymaxregn    =f8
)
create cm_linestr  (
    lineid      =i4,
    strid       =i4
)
create cm_areachain(
    achainid    =i4,
    leftzoneid  =c11,
    rightzoneid =c11,
    cmptpt      =i4
)
create cm_polachain(
    polid       =i4,
    seq         =i4,
    achainid    =i4
)
create cm_string   (
    strid       =i4,
    cmptpt      =i4
)
create cm_achainpt (
    achainid    =i4,
    seq         =i4,
    x           =f8,
    y           =f8
)
create cm_dss      (
    description =c80,
    srcfile     =c30,
    tffkey      =i4,
    zonetype    =i4,
    xminregn    =f8,
    xmaxregn    =f8,
    yminregn    =f8,
    ymaxregn    =f8,
    nzones      =i4,
    npolygons   =i4,
    nlines      =i4,
    nareachains =i4,
    nstrings    =i4,
    npoints     =i4,
    nentitypts  =i4,
    projection  =i4
)
```

Appendix D

Definition files for colourmap source decoder

The Yacc Definition file

```
%{
    /** $Header$ **/

#include <stdio.h>

#include "eql.h"

#define maxrgflg 5
#define INITRG(A) rgcnt[++rgflg] = A;
#define DECRG rgcnt[rgflg]--;

#define newtkn(A) yylval.t.code = A;
#define itv(A) A.v.ival
#define dtv(A) A.v.dval
#define stv(A) A.v.sval

int rgcnt[maxrgflg], rgflg = 0;

char *sf;   /** source datafile name           **/

    int datasize = 80; /** used by lex to determine size of string **/
                       /** (initialised for size of comment)       **/

    extern int eqlappend(), eqlreplace();
    extern char *strsave();

    double atof();

    char *zid;
    int pid, sid, nattr, nname, attrseq;
%}

%start datafile

%union {
    struct TKN {
      int code;
      union val {
      int ival;
      double dval;
      char *sval;
      }v;
    }t;
    int filetype, itmseq;
    struct PD {
      int pid, sip;
      double xminregn, xmaxregn, yminregn, ymaxregn;
    } polydefn;
  }

%type <filetype> data
%type <polydefn> poly.defn.hdr
%type <t> _zone.id_ _poly.in.zone_ _poly.id_ _segs.in.poly_ _num.attr_
```

```
%type <t> _dsp.level_ _x.min.poly.regn_ _x.max.poly.regn_
      _y.min.poly.regn_
%type <t> _y.max.poly.regn_ _x.poly.label_ _y.poly.label_
      _polygon.area_
%type <t> _segment.id_ _seg.left.zone_ _seg.right.zone_
      _points.in.segment_
%type <t> _num.zones_ _num.polygons_ _num.segments_  string
%type <t> _num.names_ _missing.dat.val_ _name_ _description_ _units_
%type <t> _attr.value_ _comment_

%type <itmseq> seg.id.list xy.defn.list name.list attribute.list
%type <itmseq> attr.val.list

%token <t> MAP POLYGONS ZONES SEGMENTS LINES POINTS NEWLINE
%token <t> INTEGER REAL STRING

/** system tokens **/

%token <t> EORG SETSIZE STRCONT

%%
/*-----------------------------------------------------------------*/
/*                Notation                                         */
/*                ========                                         */
/*                                                                 */
/*    _symbol.name_     => Lexical symbol                          */
/*    TOKEN       => a reserved symbol/data type                   */
/*    symbol.name       => yacc non-terminal symbol                */
/*                                                                 */
/*-----------------------------------------------------------------*/

datafile :  _comment_ NEWLINE header data
      {
          eqlreplace("cm_dss", "srcfile", sf, "tffkey", $4, EOAL,
EOQL);
          free(stv($1));
      }
      ;

data :            names attribute.sectn
      {
          $$ = 10;
      }
      |     zone.part
            poly.part
            seg.part
      {
          /** Zone file **/
          $$ = 0;
      }
      |     line.part
            seg.part
      {
          /** line file **/
          $$ = 2;
      }
      |     seg.part
      {
          /** line overlay **/
          $$ = 4;
      }
      |     point.part
      {
```

```
        /** Either A: site file, B: name overlay, C: marker overlay**/
          $$ = 1;
      }
      ;

zone.part :       ZONES NEWLINE
            _num.zones_ NEWLINE
              { INITRG(itv($3)); } zone.defn.list EORG
        {
          eqlreplace("cm_dss", "nzones", itv($3), EOAL, EOQL);
      }
            ;

poly.part :
      |     POLYGONS NEWLINE
            _num.polygons_ NEWLINE
            { INITRG(itv($3)); } polygon.defn.list EORG
        {
          eqlreplace("cm_dss", "npolygons", itv($3), EOAL, EOQL);
      }
      ;

seg.part :
      |     SEGMENTS NEWLINE
            _num.segments_ NEWLINE
            { INITRG(itv($3)); } segment.defn.list EORG
        {
          eqlreplace("cm_dss", "nareachains", itv($3), EOAL, EOQL);

      }
      ;

line.part   : LINES
            ;

point.part  : POINTS
            ;

map.window :      REAL REAL REAL REAL NEWLINE
      {
          eqlreplace("cm_dss", "xminregn", dtv($1), "xmaxregn",
dtv($2),
            "yminregn", dtv($3), "ymaxregn", dtv($4), EOAL, EOQL);
      }
      ;

map.projection :
            INTEGER INTEGER NEWLINE
      {
          eqlreplace("cm_dss", "projection", itv($1), "zonetype",
itv($2),
                  EOAL, EOQL);
      }
      |     INTEGER NEWLINE
      {
          eqlreplace("cm_dss", "projection", itv($1), EOAL, EOQL);
      }
      ;
```

```
zone.defn.list :
            z.defn
      |     zone.defn.list z.defn
      ;

polygon.defn.list :
            poly.defn
      |     polygon.defn.list poly.defn
      ;

segment.defn.list :
            segment.defn
      |     segment.defn.list segment.defn
      ;

z.defn :    _zone.id_ _poly.in.zone_ NEWLINE
          { INITRG(itv($2)); zid = strsave(stv($1));}
            poly.id.list EORG NEWLINE
      {
          eqlappend("cm_zone", "zoneid", zid, "cmptpol", itv($2),
EOAL);

          DECRG; free(stv($1)); free(zid);
      }
      ;

poly.defn :       poly.defn.hdr
            _x.poly.label_ _y.poly.label_ _polygon.area_ NEWLINE
          { INITRG($1.sip); pid  = $1.pid; }
            seg.id.list EORG NEWLINE
      {
          char *s1;

          s1 = (char *) malloc(200); *s1 = '\0';
          sprintf(s1, " cm_polygon.polid = %d", $1.pid);

          eqlreplace("cm_polygon", "xlabel", dtv($2), "ylabel",
dtv($3),
                   "area", dtv($4), EOAL, s1, EOQL);
          DECRG; free(s1);
      }
      |     poly.defn.hdr _x.poly.label_ _y.poly.label_ NEWLINE
          { INITRG($1.sip); } seg.id.list EORG NEWLINE
      {
          /**
           ** calculate area
           **/

          double area;
          char *s1;

          area = ($1.xmaxregn - $1.xminregn) *
            ($1.ymaxregn - $1.yminregn);

          s1 = (char *) malloc(200); *s1 = '\0';
          sprintf(s1, " cm_polygon.polid = %d", $1.pid);

          eqlreplace("cm_polygon", "xlabel", dtv($2), "ylabel",
dtv($3),
                   "area", area, EOAL, s1, EOQL);
          DECRG; free(s1);

      }
```

```
      |       poly.defn.hdr _polygon.area_ NEWLINE
          { INITRG($1.sip); } seg.id.list EORG NEWLINE
      {
          /**
           ** calculate centroid
           **/

          double cx, cy;
          char *s1;

          cx = ($1.xminregn + $1.xmaxregn) / 2;
          cy = ($1.yminregn + $1.ymaxregn) / 2;

          s1 = (char *) malloc(200); *s1 = '\0';
          sprintf(s1, " cm_polygon.polid = %d", $1.pid);

          eqlreplace("cm_polygon", "xlabel", cx, "ylabel", cy,
                  "area", dtv($2), EOAL, s1, EOQL);
          DECRG; free(s1);
      }
      ;

poly.defn.hdr :   _poly.id_ _zone.id_ _segs.in.poly_ _dsp.level_
NEWLINE
            _x.min.poly.regn_ _x.max.poly.regn_
            _y.min.poly.regn_ _y.max.poly.regn_
      {
          eqlappend("cm_polygon", "polid", itv($1), "enclzoneid",
stv($2),
                  "cmptachain", itv($3),"dsplevel", itv($4),
                  "xminregn", dtv($6), "xmaxregn", dtv($7),
                  "yminregn", dtv($8), "ymaxregn", dtv($9),
                  EOAL);

          $$.pid = itv($1); $$.sip = itv($3);
          $$.xminregn = dtv($6); $$.xmaxregn = dtv($7);
          $$.yminregn = dtv($8); $$.ymaxregn = dtv($9);
          free(stv($2));
      }
      ;

segment.defn :    _segment.id_ _seg.left.zone_ _seg.right.zone_
            _points.in.segment_ NEWLINE
          { INITRG(itv($4)); sid = itv($1); }
            xy.defn.list EORG NEWLINE
      {
          int i;

          eqlappend("cm_areachain", "achainid", itv($1), "leftzoneid",
stv($2),
                  "rightzoneid", stv($3), "cmptpt", itv($4), EOAL);
          DECRG; free(stv($2)); free(stv($3));
      }
        ;
```

```
seg.id.list :
      {
          $$ = 0;
      }
      |     seg.id.list NEWLINE
      {
          $$ = $1;
      }

      |     seg.id.list INTEGER
      {
          eqlappend("cm_polachain","polid",pid,"seq", $1, "achainid",
                  itv($2), EOAL);

          $$ = $1 + 1; DECRG;
      }
      ;

poly.id.list :
      |     poly.id.list NEWLINE

        |   poly.id.list INTEGER
      {
          eqlappend("cm_zonepol", "zoneid",zid,"polid",itv($2), EOAL);
          DECRG;
      }
      ;

xy.defn.list :    REAL REAL REAL REAL
      {

      eqlappend("cm_achainpt","achainid",sid,"seq",0,"x",dtv($1),"y",
                  dtv($2), EOAL);

      eqlappend("cm_achainpt","achainid",sid,"seq",1,"x",dtv($3),"y",
                  dtv($4), EOAL);

          $$ = 2; DECRG; DECRG;
      }
      |     xy.defn.list NEWLINE
      {
          $$ = $1;
      }
      |     xy.defn.list REAL REAL
      {
          eqlappend("cm_achainpt","achainid",sid,"seq",$1,"x",dtv($2),
                  "y", dtv($3), EOAL);

          $$ = $1 + 1; DECRG;
      }
      ;

header      :     MAP NEWLINE map.window map.projection
      |     _num.attr_ _num.names_ _missing.dat.val_ NEWLINE
      {
          nattr = itv($1); nname = itv($2);

          eqlappend("cm_attrdss", "nattributes",nattr,"nnames", nname,
                  "missdatval", dtv($3), EOAL);
      }
      ;
```

```
names       :     { INITRG(nname); } name.list EORG
      ;

name.list :
      {
          $$ = 0;
      }
      |     name.list _name_ NEWLINE
      {
          eqlappend("cm_objname", "nameseq", $1, "name", stv($2),
EOAL);

          $$ = $1 + 1; DECRG; free(stv($2));
      }
      |     name.list _name_
      {
          eqlappend("cm_objname", "nameseq", $1, "name", stv($2),
EOAL);

          $$ = $1 + 1; DECRG; free(stv($2));
      }
      ;

attribute.sectn :
            { INITRG(nattr); } attribute.list EORG
      ;

attribute.list :
      {
          $$ = 0;
      }
      |     attribute.list { attrseq = $1; datasize = 30; } attribute
      {
          $$ = $1 + 1; DECRG;
      }
      ;

attribute :       attr.defn { INITRG(nname); } attr.val.list EORG
      ;

attr.defn :       _description_ _units_ NEWLINE
      {

      eqlappend("cm_attrdefn","atid",attrseq,"description",stv($1),
                "units", stv($3), EOAL);
          free(stv($1)); free(stv($3));
      }
      ;

attr.val.list :
      {
           $$ = 0;
       }
      |     attr.val.list _attr.value_ NEWLINE
      {
          eqlappend("cm_attrvalue", "atid", attrseq, "nameseq", $1,
                "value", dtv($2), EOAL);

          $$ = $1 + 1; DECRG;
      }
```

```
      |     attr.val.list _attr.value_
      {
          eqlappend("cm_attrvalue", "atid", attrseq, "nameseq", $1,
                  "value", dtv($2), EOAL);

          $$ = $1 + 1; DECRG;
      }
      ;

/*-----------------------------------------------*/
/*          Lexicon definitions                  */
/*-----------------------------------------------*/

_attr.value_ :    REAL
      { $$ = $1; }
      ;
_comment_ :       { datasize = 80; } string
      {
            eqlappend("cm_dss", "description", stv($2), EOAL);
      }
      ;

_polygon.area_ :
            REAL
      { $$ = $1; }
      ;

_description_ :
            { datasize = 30; } string
      { $$ = $2; }
      ;

_dsp.level_ :     INTEGER
      { $$ = $1; }
      ;

_poly.in.zone_ :
            INTEGER
      { $$ = $1; }
      ;

_segs.in.poly_ :
            INTEGER
      { $$ = $1; }
      ;

_points.in.segment_ :
            INTEGER
      { $$ = $1; }
      ;

_missing.dat.val_ :
            REAL
      { $$ = $1; }
      ;

_name_ :    { datasize = 10; } string
      { $$ = $2; }
      ;

_num.attr_ :      INTEGER
      { $$ = $1; }
      ;
```

```
_num.segments_    :
            INTEGER
      { $$ = $1; }
      ;

_num.names_       :
            INTEGER
      { $$ = $1; }
      ;

_num.polygons_ :
            INTEGER
      { $$ = $1; }
      ;

_num.zones_ :     INTEGER
      { $$ = $1; }
      ;

_poly.id_ :       INTEGER
      { $$ = $1; }
      ;

_segment.id_ :    INTEGER
        { $$ = $1; }
      ;

_seg.left.zone_   :
          { datasize = 10; } string
      { $$ = $2; }
      ;

_seg.right.zone_ :
          { datasize = 10; } string
      { $$ = $2; }
      ;

_units_ :   { datasize = 10; } string
      { $$ = $2; }
      ;

_x.max.poly.regn_ :
            REAL
      { $$ = $1; }
      ;

_x.min.poly.regn_ :
            REAL
      { $$ = $1; }
      ;

_x.poly.label_ :
            REAL
      { $$ = $1; }
      ;

_y.max.poly.regn_ :
            REAL
      { $$ = $1; }
      ;
```

```
_y.min.poly.regn_ :
            REAL
      { $$ = $1; }
      ;

_y.poly.label_ :
            REAL
      { $$ = $1; }
      ;

_zone.id_ :  { datasize = 10;} string
      { $$ = $2; }
      ;

/*-----------------------------------------------*/
/*          General Constructs                   */
/*-----------------------------------------------*/

string      :     SETSIZE STRING
      {
          $$ = $2;
      }
      ;

%%
#include "lex.c"

yyerror(s)
     char *s;
{
  printf("yytext:%s:\n", yytext);
}
main()

{
    FILE *freopen();

    sf = (char *) malloc(30);

    printf("Source data filename ?              "); fflush(stdout);
    scanf("%s", sf); getc(stdin);

    freopen(sf, "r", stdin);

    eqlopen("kiwi::cm", "-c10");
    yyparse();
    eqlclose();


       }
```

The Lex definition file

```
%a 5000
%o 9000
%{
#define input() ((!(rgcnt[rgflg]) && rgflg) ? (rgflg--, '@') :\
(((yytchar=yysptr>yysbuf?U(*--yysptr):\
getc(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar))

#define REAL_SIZE 10
#define INT_SIZE  10
int sizeset = 0;
%}
D           [0-9]
E           [E][-+]{D}{D}
W           [\t ]
BYTE        [\,\-A-Za-z0-9'$#!%^&~`:?/]
%%
{W}*\n                  { return( NEWLINE ); }

MAP{W}*\n               { yyless(yyleng - 1); return( MAP ); }

POLYGONS{W}*\n          { yyless(yyleng - 1); return( POLYGONS ); }

ZONES{W}*\n             { yyless(yyleng - 1); return( ZONES );}

SEGMENTS{W}*\n          {  yyless(yyleng - 1); return( SEGMENTS );}

LINES{W}*\n             {  yyless(yyleng -1); return( LINES );}

POINTS{W}*\n            {  yyless(yyleng - 1); return( POINTS );}

{W}*[+-]?{D}+"."{D}*({E})?{W}*     |
{W}*[+-]?{D}*"."{D}+({E})?{W}*  {
                        if (yyleng > REAL_SIZE)
                          yyless(yyleng - (yyleng - REAL_SIZE));
                          newtkn( REAL );
                        dtv(yylval.t) = atof(yytext);
                        return( REAL );
                    }
{W}+[+-]?{D}+{W}*       {
                        if (yyleng > INT_SIZE)
                          yyless(yyleng - (yyleng - INT_SIZE));
                              newtkn( INTEGER );
                        itv(yylval.t) = atoi(yytext);
                        return( INTEGER );
                    }
(({BYTE}(".")?)+{W}*)+ {
    if (!sizeset) {
      yyless(0); sizeset++;
      return( SETSIZE );
    }
    else if (yyleng > datasize)
      yyless(datasize);
    newtkn( STRING );
    stv(yylval.t) = strsave(yytext);
    sizeset = 0;
    return( STRING );
}
@       { return( EORG ); }
%%
```