Dealer Maintenance Notes by JGM

Table of Contents

Preamble

While the user documentation for dealer is sufficient there is a lack of documentation for those who want to maintain or modify the program.

The original author is Hans van Staveren, Amsterdam, Holland , (circa 1990), the latest maintainer is Henk Uijterwaal circa 1999. There have not been many changes made to dealer in the last 20 years.

Dealer is now used by BBO to deal practice hands for its users.

Dealer was originally intended to just generate deals of interest for study, bidding practice, perhaps to analyze a line of play and so on. It was not, (and is still not) intended to generate hands for play in tournaments and such like. However it was (and is) such a useful program that many people have added functionality to it over the years. I, JGM, wanted to use it to experiment with different methods of hand evaluation. In order to do that I had to learn a fair bit about Dealer's structure and workings. What you read here is my humble attempt to document my findings in case someone else wants to modify Dealer even further.

I should at this time also state that none of the original authors have been involved in my attempt. If there are any errors in my description herein, they are all my own.

Overview

Dealer defines a 'grammar' of user commands in typical BNF form. The user prepares an 'input file' consisting of the instructions written in that grammar as to what kind of bridge deals he wants.

Dealer uses the yacc/bison tools to generate a parser that can process instructions written in that grammar. yacc/bison relies on a 'lexer', in the case of Linux, flex, to parse the user's input into tokens.

So the structure of dealer is made up of:

1) A grammar definition file, called defs.y in the original distribution. JGM has renamed this dealyacc.y in his code.

2) A lexer definition file, called scan.l in the original distribution. JGM has renamed this dealflex.l in his code.

3) A main program and several subroutines, in a file called, dealerv2.c

4) Various header files and supporting routine header and code files

5) Makefiles

6) Examples and regression tests.

Workflow

YACC stands for, "Yet Another Compiler Compiler". The goal of yacc is to parse an input file according to the grammar for the target language.

The Dealer workflow is slightly different, from that of a compiler. The output of YACC/BISON is a routine called yyparse() and its supporting functions as defined in the dealyacc.y file. This routine is compiled into the dealer application along with the other C code. The user enters his 'input' file into dealer and the parser produces several in memory data structures that encode the user's instructions.

Maintenance 2022

The early versions of dealer were developed on a 32 bit SUN workstation which by 2021 standards would not have qualified as a cell phone. Windows95 was not available. So there was considerable effort made to ensure that the code was portable to various flavors of Unix and even to a Unix like environment (Cygwin et. al.) on Windows 3.x. and even on DOS.

As of 2022, Linux has essentially taken over almost all of the Unix market (there is still some BSD, SUN and AIX but these are unlikely to be relevant to a bridge playing program). There is also MacOS which has a Unix pedigree, but is now completely closed and proprietary. The systems are almost all 64 bit and at least 1000 times faster than the original SUN workstation. Furthermore, Windows now includes WSL 2.0 which can run almost all Linux command line applications, and even many graphical (X window) applications, natively.

So it does not make a lot of sense to hobble the development of dealer with 1990 era constraints. Accordingly JGM has decided to use any feature of Linux/Bison/Flex that seems useful, regardless of portability concerns such as POSIX compliance, or Windows tool chains.

Furthermore the (Standard ANSI) C language is much improved from 1990, allowing functions to return structures and unions, including some extra libraries for random number generation, firming up function prototype definitions and the like. These features too, are used whenever appropriate.

Hierarchy of Files

```
dealdefs.h -> dealtypes.h->dealexterns.h----------v
                           -> dealglobals.c  ---> dealyacc.y, dealflex.l, dealerv2.c, deal{xxx}subs.c etc.
                           -> dealprotos.h----------^
```

Dealer Heritage

This next bit is taken from the original dealer documentation; it gives a flavor of what the landscape was like back in 1989-ish when the program was first developed, and in the years immediately following.

<Quote>

Portability

The program has been developed under SunOS on Sun hardware. I (Hans) am a reasonably competent programmer though, and I expect the program to be highly portable. The most suspect part is in the initialization of the random number generator. The system call used here might need to be changed on other OSes.

There has been a DOS and OS/2 port by Pedl Rau. I tested the speed at about 9000 hands a second on my 486DX33. ***Note:*** *this has been superseded by the work from (a.o.) Paul Baxter, Alex Martelli and Danil Suits.*

Henk Uijterwaal has compiled his version under Linux (2.0.x and 2.2.x) without any problem, except that one needed to do:

```
ln -s /usr/bin/flex /usr/bin/lex
```

as flex has succeeded lex on Linux. On Ultrix, DEC Alpha(OSF), BSDI/3.x and SGI, the program compiled without any problem.

A. Martelli (alex@magenta.com) used to build and run Dealer on Win/NT and Win/98 with the free Cygwin GCC C/C++ compiler and Unix emulation environment. Main changes needed were byacc instead of yacc, as well as flex instead of lex, in the Makefile. A Makefile with all these changes is available in the MSDOS subdirectory.

Dealer's latest versions build and run fine using Microsoft Visual Studio 5 and 6 (still using byacc and flex if needed for syntax mods, but with no other known dependency on Cygwin); there is a current dependency on the Winsock2 DLL (for ntohs, ntohl functions) soon to be removed.

</end Quote>

Even in 1999, the last date I have for when work was done on this program by Henk Uijterwaal and others, the Linux kernel was still only in version 2.2.

It is now 2022, so I, JGM, have decided to use any Linux based tool that seemed appropriate without worrying too much about portability. I only started my modifications for my own learning and needs and never expected to release this code, and it is too late now to worry about portability.

Novice's Introduction to Bison/Yacc and Flex

Overview

This bit is a 50,000 foot view of Bison and Flex. Written by a novice, for novices. If you really want to understand the theory behind how Bison generates its parsers, you need to read up on Compiler Design and LALR(1) grammars and so forth.

YACC stands for Yet Another Compiler Compiler and is a tool to help people write compilers for new languages that they come up with. The original author of Dealer, Hans van Steveren (HvS) used YACC in a clever way to develop a parser for the dealer input file commands. It's not quite a compiler, but it is a very powerful parser.

So the overall process is this:

Define your 'language'. A language has a syntax and a grammar. Express your grammar in 'rules' that YACC understands. When you feed these rules into YACC it will produce a piece of magic C code that will parse your input for you. This piece of C code is a procedure called **yyparse().** The grammar definition is usually in a file with a .y extension.

You also define your rules for making tokens that you feed into the parser. You process these token making rules with Flex and it produces a piece of not quite so magic C code that will break your input stream up into tokens. The lexer definition rules are usually in a file with a .l extension.

You then write a main program which calls yyparse to parse the input, and your main program then does whatever is called for with what yyparse produces. Usually there will be significant extra code required over and above just the parsing step, but not always. The GNU Bison examples have a main program that just calls yyparse and nothing else.

yyparse works by assembling 'tokens' into something more complicated; in the case of a parser that is say building sentences, the tokens are words. The bison examples on the GNU web site are various forms of calculators; the tokens are arithmetic operators, variable names, numbers, and so forth.

yyparse, the parser, gets its tokens by calling a C procedure named **yylex(),** the lexer. (Lexing is short for 'lexical analysis' which is essentially breaking text up into its component parts.)

yylex's job is to read the input stream and break it up into tokens according to certain rules. In the case of sentences the tokens are groups of letters between whitespace and punctuation i.e. words. In the case of the calculators the tokens are numbers and symbols and so on.

You can run yyparse() as produced by bison, without involving flex to produce your lexer. If your lexer is simple enough you can code it by hand. In fact that is what the bison examples on the GNU web site do; they do not use flex at all.

You can also run your lexer without involving a parser if your grammar is simple enough; i.e. you can in effect write your parser by hand. That is what some of the examples in O'Reilley's lex and yacc book do.

But usually you use the two of them together as a pair. Bison to produce the parser, **yyparse**, and Flex to produce the lexer, **yylex.** Using these programs rather than hand coding, leads to fewer bugs, quicker development, and much easier maintenance. And the performance is on par with what you could write by hand.

The Lexing Process -- Flex

The lexer's job is to produce tokens for the parser to assemble into something more elaborate. Each token always has an 'ID' associated with it, this is the token type. For example a string of digits might have the token type INTNUM, a string of digits with a decimal point could have the token type FLTNUM. A word could have the token type NOUN or VERB or CONJUNCTION.

Typically in the code we write we express token types in all capitals; other symbols that are made up of several tokens put together are in lower case.

In addition to its type, most tokens also have 'semantic value'. For a VERB say the semantic value would be something like, "does", "is", "has" and so forth; a FLTNUM could have a semantic value of 3.14159 or 2.7182818 say. When the lexer is returning the token to the parser it returns the token type as the value in the return statement, and it puts the semantic value into a global variable that is shared between the lexer and the parser. In the case of bison, this global variable is called yylval, so flex puts the semantic value of the token type it is returning there.

Not all tokens have a semantic value attached; for example in a calculator one of the tokens might be a plus sign, '+' ; this plus sign has no semantic value, it's just a token.

These two pieces of information are the main ones; you can build your application with nothing more. However a compiler (which is what bison/yacc is helping you build) is expected to provide diagnostic error codes and messages, and so the location of the token in the input stream, line and column numbers say, could be very useful. Flex and Bison have a way of sharing the location of the various tokens between them to help with such issues.

The lexer can also process certain tokens on its own; if the token does not need to be assembled into a more complicated structure, then having the 'front end' lexer handle what it can, can simplify the grammar and speed up the parser.

An example is comments; if the parser is just going to ignore the comments anyway, then the lexer might as well just skip over them itself. Another example in the Dealer program is the token 'produce'. This token specifies the maximum number of hands that meet the criteria, the Dealer program output. When the parser sees this token it takes its semantic value and puts it into a global variable that the main procedure will refer to later. But there is no need to involve the parser in such a simple task, the lexer already knows the semantic value of the number associated with the 'produce' word, and can easily populate the global variable itself.

The Lexing Rules File (.l)

The input to the flex program, the dot ell file, is essentially a set of lines that follow the model:

- *Column 1:* Pattern aka Regular Expression
- *Whitespace*
- { C code between curly braces that says what action to take when the pattern is matched. }
  Here is an example of a Flex specification.

```
        [0-9]+{ws} { yylval = atoi(yytext) ; return INTNUM ; }
```

This says that when you see one or more digits with some whitespace following them, use the C function "`atoi`()" to convert the text you just matched to an integer; put the integer value into the global value yylval and return the token type INTNUM to yyparse.

That is the essence of a lexer specification; of course users want the ability to do more than just this, and Flex has many features designed to allow writing powerful lexer specs easily and quickly.

Flex takes specifications such as the above and produces a Finite State Machine that matches the input patterns, breaks the input text up into tokens, takes the corresponding actions, and passes the token type and semantic value to the parser. The name of the C procedure that does this job is yylex(). As stated earlier while it is possible to generate a lexer by hand, it is quicker and easier to maintain, if you use a tool such as Flex to do it. It's the difference between writing declarative statements and procedural ones.

Generating The Parser with Bison

In order to build an application using a language, one needs to parse it. Many years ago, defining new languages was quite common; so someone at Bell Labs, where Unix, C, and many other software tools were invented, wrote the program Yacc, which stood for, "Yet Another Compiler Compiler'. This tool was designed to help people write compilers. The GNU re-implementation of Yacc, is Bison.

Bison, like Yacc before it, will take a series of declarative statements that define a language grammar, and it will produce an 'automaton' (State Machine) that can parse 'programs' written in that language.

The name of the C procedure that does the parsing is **yyparse().** yyparse calls a lexer, most often **yylex()** whenever it needs a token. Then using the context established by previous tokens, and the rules defined in the grammar, the parser can either process the user's input itself (see the various GNU calculator examples) or, in the case of our Dealer program, it creates some data structures (decision tree, and action list) that the main program uses to generate bridge hands of interest.

The parser's job is considerably more complex than that of the lexer. The author of the dealer program, Hans van Steveren, was very clever in his use of the lex and yacc tools to build this application.

The Parsing Rules File (.y)

The file that defines the grammar and the token types typically has a dot wye extension. There are several good examples of such files on the GNU web site in the Bison documentation section.

One of the things about grammar rules is that they are usually recursive; it is extremely common to see rules that read, "An expression is either a number, or an expression in parentheses, or an expression followed by the character '*' and another expression". You can see that the expression is defined in terms of itself, like recursive function calls in many programming languages.

To really understand how a parser automaton is generated and works you need to read up on compiler design, LALR(1) grammars, shift / reduce parsing and so on. That is well beyond what I can go into here, but the GNU Bison site is an excellent place to start.

We turn now to the dealer program itself.

Overview of the Operation of Dealer

The current production version of Dealer, has good documentation on how to use the program, what words and specifications you can put in the input file and so on. But the Dealer documentation is pretty sparse when it comes to describing how the program works, so that those who want to modify or extend it really have to study the code. As explained in the prologue, the following is what I have learned from extending Dealer's functionality.

The program can be thought of as consisting of 4 main parts:

1. Initialization and setup. This portion also processes any options entered on the command line. JGM in version 2.0 has almost doubled the number of parameters that can be passed into Dealer this way.

2. Calling yyparse(). This step processes the user's input file and builds several 'trees' and linked lists in memory that tell Dealer how to generate deals that are 'interesting' to the user.

3. Setting Context. This step is a sort of initialization part II. Some of the results produced by yyparse, are over-ridden by the parameters passed in on the command line.

4. The Main Loop. This bit consists of several parts.

> 4.1 Use a random number generator to shuffle a deck of cards.
>
> 4.2 For each deal generated in 4.1 walk the decision tree produced by the yyparse stage to see if the deal is 'interesting'. An 'interesting' deal is one that satisfies the condition as defined in the input file. This step is explained in more detail later in this document. If the deal is interesting, proceed to step 4.3. If the deal is not interesting discard it and return to step 4.1
>
> 4.3 If the deal is interesting then run through the list of 'actions' defined by the user and perform them. Most actions involve producing some form of output, usually via a print statement. Many times there is only one action statement such as **printall** (the default action). Actions are of two types, those that produce output for every interesting deal, and those that produce output only at the end of the run. Examples of the first kind are the **printall**, or **printoneline** actions. Examples of the second kind are the **average** and **frequency** actions. It not usually useful to have more than one action that produces output for every deal as the output(s) from them would all be mixed together. But it is quite useful to have one action that produces output for every deal, and several other actions that wait till the end of the run to produce their output, such as several Averages, or Frequency counts.
>
> 4.4 The Main Loop finishes when either the Generate or the Produce count is reached, whichever comes first.

5. The end of the run processing.

After finishing the main loop, Dealer outputs the results of the delayed actions, the Averages, the Frequency Counts, the Print Compass and the Evalcontract. If the user has left the verbose mode set to ON it also outputs the time taken and the total number of deals generated and produced, one of which will be less than the user asked for.

Evaluating the Condition Clause

Evaluating the condition clause is the most complex part of what Dealer does. For one thing there are very many aspects of a bridge hand, or deal, that the user may wish to use as a selection criteria. These aspects include such things as: suit lengths, high card points, controls, shape and many others. They can be combined into complex expressions with each other, using many of the standard arithmetic operators such as plus, minus, division and multiplication. The complex expressions can be further combined into logic clauses with standard logic predicates such as 'not', 'and', 'or', and 'if'. See the user's documentation for details.
The parser will assemble the various key words and functions in the input file into a series of 'clauses' known in this instance as 'rules'. Whenever the parser has assembled enough tokens from the lexer to make up a rule, it executes the C code that the programmer has specified for that rule. It is Yacc/Bison, the parser generator, that is responsible for generating the C code that manages to keep the rules coherent and in a logical sequence. The programmer does not need to know it does this. So long as he has specified a grammar with no 'bugs' in it, Yacc/Bison will generate a correct parser.

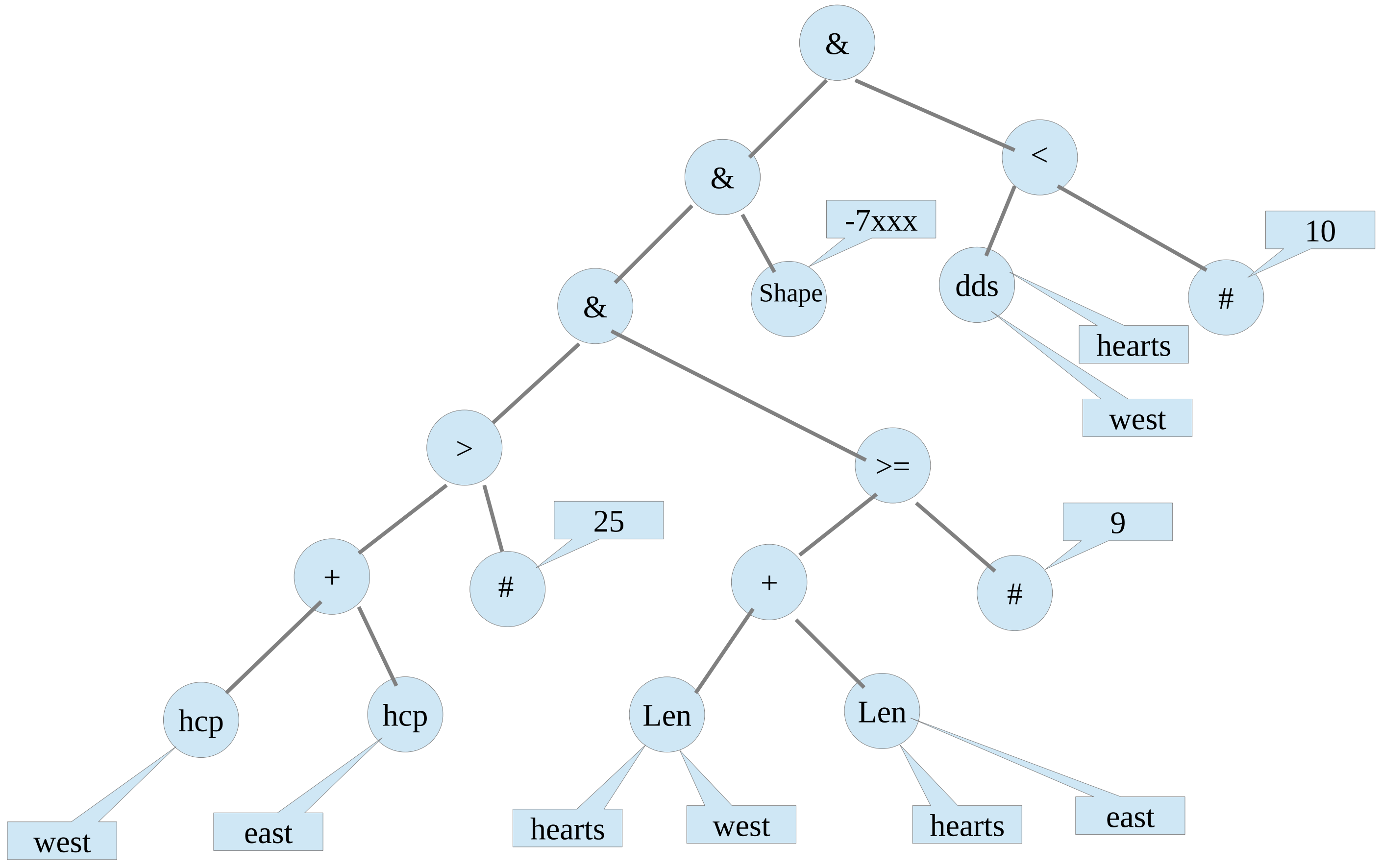
In Dealer the usual 'user action' that is taken when a condition 'rule' is recognized is to add a 'node' to the decision tree. A diagram of a simple decision tree is shown on the next page. The programmer does not have to try to ensure that the new node is added in the right place on the tree. That is the parser's job. The programmer just has to make sure that the node is the correct type and also that whatever values are needed at runtime are part of the data carried with the node.
In the following diagram, each node has: i) a type always present, ii) two integer values that are part of the node structure; one, none, or both of these may be used; and iii) two pointers again 0, 1, or both of these may be used to point to child nodes of the current one.

As an example, when the following condition expression:

```
condition ( ( hcp(west) + hcp(east) )      > 25 &&
            (hearts(west) + hearts(east)) >= 9 &&
             shape(west, - any 7xxx ) &&  dds(west, hearts) < 10 )
```

is parsed by the yacc/bison generated yyparse() function a 'decision tree' is created in memory at run time. This is done once while the input file is being parsed, and is used for the rest of the run. On the next page is a diagram of what the decision tree looks like. In the diagram the type of node is shown inside the circle, the values attached to the node are shown in the rectangular boxes, and the links between the nodes by straight lines.
The evaluation function recursively calls itself, the effect is that the decision tree is evaluated left to right bottom to top. This is pretty much the same order as the clauses appear in the condition statement. This fact will have some practical value as will be explained later.
As soon as the evaluation function can make a decision, it does so and it does not continue to evaluate all the nodes if it is not necessary.
In this example there are many clauses joined by 'and' therefore as soon as one of the branches of an 'and' is false the whole expression must be false and the evaluation function can finish.

Example Decision Tree



Since the evaluation is done left to right, roughly in the same order as specified in the condition statement, it pays to put the quicker evaluations first; if these fail the evaluation routines may never have to execute the slower functions. In this example the DDS (Double Dummy Solver) function is the slowest, since it is more difficult to calculate the number of tricks a hand can take than to count its hcp or the number of cards in a suit. So we put it last in the condition statement; had we put it first, it would have been called on every deal. As shown it is called only for those deals which meet all other conditions.

Code Locations -- src Directory

JGM has (re)-organized the C code that makes up the Dealer application as follows: (all of the following are in the "src" directory.

dealflex.l - there is some C code here that implements Flex action statements and some functions that are called by these action statements. This file also contains the lexing rules for Flex.

dealyacc.y - there is some C code here that is run whenever the parser 'reduces' a rule; typically this C code adds a node to the decision tree or adds another item to the action list or the variable list. There is also C code that consists of functions that are called by the tree building statements in the rules. This file also contains the grammar definition for the Dealer application.

dealerv2.c - This file includes the "main" function, the initialization, and the main loop code.

mainsubs.c - this code is used by the main program to parse command line variables, initialize the random number generator and so forth.

dealparse_subs.c - These are more extensive routines that are run during the parser phase of Dealer. They do not implement evaluation or bridge oriented functionality; they do not form part of the main loop code. The are primarily called from code in the dealflex.l and dealyacc.y files.

dealeval_subs.c - These functions are run during the evaluation phase of the main loop. That is they are run for every random hand that Dealer generates. These functions make up the code that is run depending on which node of the decision tree is being traversed.

c4.c   - This code implements the Kaplan and Rubens Four C's hand evaluation method, as well as the Suit Quality measurements. The code is run during the evaluation phase but it is in its own file to make it easier to maintain.

dealdds_subs.c - This code implements the calls to the Double Dummy Solver library. It is run during the evaluation phase, but the functions have been given their own file to make them easier to maintain.

Dealer_DDS_IF.c - This code provides the interface between Dealer and the DDS library. It is in its own file to avoid name conflicts with code in the other dealer files.

dealaction_subs.c - These functions are run when the evaluation phase marks the deal as interesting. They implement the print, average, frequency and such like functionality.

dealdebug_subs.c - These are functions written by JGM to debug his code and to learn how Dealer works. Unless the symbol JGMDBG is defined at compile time, they should have no effect on the operation of Dealer.

Code Locations -- include Directory

The header files are in the "include" directory

std_hdrs.h - this file contains include statements for all the standard headers needed, such as stdlib.h, stdio.h, assert.h etc. The first statement in this file is to define the macro _GNU_SOURCE. This has the effect of 'turning on' all the GNU extensions that are available in glibc, such as extra string functions and so on. If you want strictly POSIX or strictly ANSI C, you will need to define a different macro.

dealdefs.h - this file defines all the symbolic constants used. It is needed by all the other .h files as well as most of the .c files.

dealtypes.h - this file defines the various structures, arrays, typedefs and so on. It uses the definitions from dealdefs.h

dealglobals.c - This file defines and allocates space for the global variables used in dealer. There are probably too many of these, but several are used to exchange information between Flex, Yacc, and the Dealer main program, as well as between functions compiled in different files. This file should only be included in the main program file, dealerv2.c, not in any of the other files.

dealexterns.h - This file is the counterpart to dealglobals.c. It contains extern statements for the global variables defined in dealglobals.c

dealprotos.h - This file contains prototype statements for most of the functions that are not defined in the same file where they are used.

The above files are usually all that is needed by most of the Dealer .c files. In addition there are a few files that are only needed by a small subset of the code:

c4.h - header info used only by the c4.c file, and defining an interface to that code.

deal_dll.h - this file is an extract from the dll file provided with the DDS library. It imports the minimum amount of information to avoid potential naming conflicts.

deal_dds.h - this file makes available to the code that uses the DDS library the minimal amount of information from the Dealer program.

pointcount.h - included into dealtypes.h because it defines the max size of the pointcount array.

dealdebug_subs.h - some prototypes and definitions for the debugging routines.

Code Locations -- Other Directories

The Prod directory contains the Makefile and the binary for the 'production' version of dealerv2.

The Debug directory contains the Makefile and the binary for the debug version of Dealer, dealdbg
It also contains a DebugExamples sub-directory for testing various mods to Dealer. In this directory the symbol JGMDBG is defined, so that the debugging code in the Dealer source files is included in the resulting binary. The included debugging code is triggered by setting the -D switch on the cmd line to a value between 1 and 9. If the -D switch is set to 0 then all the debugging output is suppressed.

The docs directory contains the User Guide, a README file, this Maintainer's guide, license and copyright files, and other useful background info.

The lib directory contains the pre-compiled DDS library. This is a static library, NOT a shared object. I hope that this means it will not be necessary to recompile it for different Linux kernels. While Dealer itself can be compiled with ordinary C compiler, DDS must be compiled with C++. Since this library was compiled with g++ the final dealerv2 binary link step must be done with g++. The Makefiles have been pre-built to do this.
lib also contains a tarball of the JGM's Perl code that implements the Optimal Point Count hand evaluation method. This code is needed if you intend to use the 'opc' functionality of Dealer.
There is no 'package' install of this, but Dealer has been pre-compiled to run the code from the /usr/local/bin/DOP directory. You will need super-user priviledges to install it in that location.

The Examples directory contains many Dealer 'Input Files' demonstrating how to use the various features of Dealer. Many of these come from the original version. There are also files giving the output from these examples on my machine, and a perl script to compare the latest output to the reference output. This has been copied from version 1.

The DDS directory contains a tarball of Bo Haglunds Double Dummy Solver.
If you intend to rebuild this library it is best to start with BH's distribution, but you might find some of the files in this directory helpful.

Modification Examples

Adding a Title.

The objective is to be able to add a title string to some of the output, so as to describe, for example, what bidding practice is being generated. One can also put this title out as part of the PBN report.
The title will be a simple character array in a global variable. As such there is no real grammar associated with this; we can do it all in the Flex file.
1. Allocate space for the title string, and its length in the dealglobals.c file. Also add an extern definition for these global variables in the dealexterns.h file.
2. In the dealflex.l file define the pattern 'title' beginning in column 1. We could make a regex that matches 'title' followed by whitespace then followed by a quoted string. But we want the user to be able to put the title on a separate line. So we make use of Flex's "exclusive state" feature so that once 'title' is

seen the only thing that Flex will recognize is a quoted string. The user action statement associated with the quoted string, copies the string to the global variable, and also sets the length. It then exits the exclusive state and resumes looking for a token that it can pass to yyparse.
Here is the code as it is in the dealflex.l file:

```
%x TITLE
ws [ \t]+
nl \n
qstring   \"[^\"\n]*[\"\n]
title({ws}|{nl}) { BEGIN(TITLE);  }
<TITLE>{nl}   ;                /* do nothing if {nl} found. */
<TITLE>{qstring}  {
                    strncpy(title, yytext+1, (yyleng-2) ) ;
                    title[yyleng-2] = '\0';
                    title_len = strlen(title) ;
                    BEGIN(INITIAL) ;
                  }
```

The first line defines a new exclusive state to Flex.
The next 3 lines just define some symbolic constants to make reading the rest of the code easier.
The line beginning with 'title' tells Flex to look for 'title' followed by whitespace, or newline. If it sees that, it enters the TITLE exclusive state, looking only for a quoted string.
When Flex finds a quoted string it copies the text it found (less the opening and closing quote marks) to the global variable and adds a null string terminator at the end.
Then for convenience it saves the length of the string in another global variable, and exits the TITLE state and returns to the default, INITIAL state.

Other parts of the program can test the title_len variable and if it is non-zero they know that there is a title that can be output along with the Dealer results. If the user does not want a title mixed in with his output he does not enter one.

This is all done without involving the grammar file, dealyacc.y.

Adding Losing Trick Count (ltc)

The next example adds a new feature to the Dealer vocabulary. I wanted to have a more 'modern' LTC than just counting losers. For example AJT should be counted as 1.5 losers not 2; Qxx should be counted as 2.5 losers not 2. In the input file the user would be able to enter conditions such as:

`ltc(north)<5.5` or perhaps `ltc(north,spades)<2. && ltc(south, clubs)<1.`

You can see that this is very similar to the syntax of other aspects of hand evaluation such as HCP, or length in a suit. So the grammar would also be very similar to the HCP grammar.

The steps involved are as follows:

I - Grammar file (dealyacc.y)

1. Define a new token LTC with a `%token LTC` statement.

2. Write two new rules, the first to handle the case where the user enters only the 'compass' direction [e.g. ltc(north)] and the second where the user enters the compass direction and the suit.

3. In the action statement associated with reducing the first rule, create a new node of type TRT_LTCTOTAL and put the compass direction value (0..3) in the first integer of the node.

4. In the action statement associated with reducing the second rule, create a new node of type TRT_LTC, put the compass value in the first integer, and the suit value in the second integer.

5. Bison will assign an integer value to the new token LTC when it builds the parser from the grammar file. Flex will become aware of this value via the `#include dealyacc.tab.h` file which is generated from the grammar file along with the yyparse C code. In the dealflex.l file we tell Flex what to do when it sees an 'ltc' word: `ltc     return(LTC);`

6. The next step is to write the code that will calculate the LTC values for various suit combinations. The code for this is in the dealeval_subs.c file. The way Dealer is written, after the random deal is generated, but before the decision tree is scanned, there is a routine called 'analyze' which pre-calculates several aspects of hand evaluation, such as High Card Points, suit lengths, controls, in general any aspect of the hand which is likely to be needed on most deals, and which is relatively straightforward to derive. We add to the 'analyze' function a call to the function we write to calculate the new ltc.

7. There is one aspect of the above which is not mentioned here, and that is the use of numbers with decimal points (such as 2.5 in the above example). Version 1 of Dealer had no notion of any number that was not an integer; all the calculations and all the return values were all integers. In version 2, JGM extended the Dealer syntax to allow the user to input numbers of the form 99.99 but internally Dealer remains an integer only application. See the release notes where the new 'Dotnum' feature of Dealer is described for the limitations of this feature.

Here is the code that has been added to the grammar file to implement the LTC feature. It is virtually identical to the code that defines HCP for example:

```
| LTC '(' compass ')'
         { $$ = newtree(TRT_LTCTOTAL, NIL, NIL, $3, 0 ); }
| LTC '(' compass ',' SUIT ')'
         { $$ = newtree(TRT_LTC,      NIL, NIL, $3, $5); }
```

"$3" in the above refers to the value that has been assigned to 'compass'. "$5" refers to the value that has been assigned to "SUIT". *Aside: $1 is LTC , $2 is the* ***(*** *left parens and $4 is the comma in the second rule.*

Random Number Generation

The original author of Dealer, HvS, was quite conscious of the time taken to generate and analyze hundreds of thousands of deals in order to get few hundred that would meet the user's criteria. For example the original dealer code had a very clever way of avoiding modulo division when generating a random number. According to my tests HvS's method cut the time to generate a random number in half.

The system on which the following performance numbers were generated is as follows:

```
System:     Linux Kernel: 5.4.0-96-generic x86_64 bits:
Machine:   Type: Desktop Mobo: ASUSTeK model: P8Z68 DELUXE
CPU:        Topology: Quad Core model: Intel Core i7-2600K bits: 64 type: MT MCP
L2 cache:  8192 KiB
Speed:    1605 MHz min/max: 1600/3800 MHz Core speeds (MHz):  4 CPUs, 8 Cores.
```

Dealer itself was compiled with GCC version 9, with optimization level -O3 or -Og.

<u>Random Number Generation:</u>

```
Avg Random Number (0..100) over 10 million deals using rand :
Average=   49.48, Std Dev=   28.87, Var=    833.37, Sample Size=10000000

10 Million Random Numbers
Generated 10000000 hands
Produced 10000000 hands
Initial random seed 138249490574914
Time needed   18.067 sec
```

Without the Debugging code and symbols, and using -O3 optimization the result is:

```
Avg Random Number (0..100):
Mean=   49.50, Std Dev=   28.87, Var=    833.23, Sample Size=10000000
Random Number Generator Test 10,000,000 No Conditions
Generated 10000000 hands
Produced  10000000 hands
Initial random seed 113
Time needed   12.773 sec
```

The original Version 1.4 code as distributed by Debian (and compiled on my machine with -O3) gives:

```
Avg Random Number (0..100) over 10 million deals using rand : 49.5036
Generated 10000000 hands
Produced 10000000 hands
Initial random seed 1643427436
Time needed    5.046 sec
```

The difference is that in the first two cases JGM has removed HvS's clever workaround which sped up the RNG. The main reason being that I did not quite understand it; it seemed to me that this method

(which is essentially 16 bit?) would result in a relatively short period for the RNG; my concerns were further amplified by HvS's statements that the RNG was the most dubious part of the program, and that the program was not suitable for generating deals for such things as tournament play. I wanted to use the current GNU 48 bit random number generator, (which was not available when Hans first wrote his program) and I wanted to be certain that the period of the RNG was sufficiently long for the simulation studies that I was doing. I figured I could afford a few extra micro-seconds to be certain.
If we assume that we will need 10,000 deals generated per simulation run, and each deal needs 52 random numbers for the shuffle, then we need $5.2*10^5$ random numbers per run. To avoid the 'birthday paradox' the period of the RNG should be greater than the square of this number or $27*2^{33}$. The period of the 2022 GNU RNG is said to be 2^{45}. In addition I have taken the precaution of seeding the RNG with values from the Linux Kernel's entropy pool to ensure as much randomness as possible. Of course the user still has the option of specifying an initial seed value as he could in previous versions.
For reference it is the modulo division required to map the number generated by the RNG into the range 0 .. 52 that takes the extra time. The GNU RNG is not the source of the time taken; a program that does nothing but return 50 Million random numbers takes the following amount of time:

```
CheckRandom:: Version=DOUBLES, Seed=234021249433746
Average over 50000000 loops = 0.500036
real     0m0.376s  user         0m0.375s    sys        0m0.001s

CheckRandom:: Version=LONGS, Seed=47779911134368,
Average over 50000000 loops = 1.07375e+09
real     0m0.295s  user         0m0.289s   sys         0m0.004s
```

The older GNU RNG gives these results:

```
CheckRandom:: Version=SHORTS, Seed=0,
Average over 50000000 loops = 1.07376e+09
real     0m0.423s  user         0m0.419s   sys         0m0.004s
```

For reference the GNU RNG has 3 different modes: 1) generate a double in the interval [0,1), generate a 32 bit integer in the interval $[0, 2^{31})$ and generate a signed integer in the interval $[-2^{31}, 2^{31})$.
They all take about the same amount of time and give about the same average and standard deviation.

Other Performance Figures

Simple Condition File

The next result is for the following condition clause:

```
condition   shape(north, any 4333 + any 4432 + any 5332) &&
            hcp(north)>=20 and hcp(north)<=22
```

```
Generated 1,550,673 hands
Produced 10,000 hands
Initial random seed 113
Time needed    2.777 sec
real     0m2.784s user          0m2.768s       sys      0m0.022s
```

For a somewhat more complicated Stayman evaluation such as:

```
east_notrump = shape(east, any 4333 + any 4432 + any 5332) and
               hcp(east) >= 15 and hcp(east) <= 17
west_stayman = shape(west, 3451 + 4351) and hcp(west) <= 7

east_2d = shape(east, xxxx - 4xxx - x4xx - 5xxx - x5xx)
east_2h = shape(east, x4xx)
east_2s = shape(east, 4xxx - x4xx)
east_2n = shape(east, 5xxx + x5xx)

d = diamonds(east)+diamonds(west)
h = hearts(east)+hearts(west)
s = spades(east)+spades(west)

found_d_fit = east_2d and d>=8
found_h_fit = east_2h and h>=8
found_s_fit = east_2s and s>=8
found_M_fit = east_2n and (h>=9 or s>=9)

condition east_notrump and west_stayman

action
     average "fit" found_d_fit or found_h_fit or
                    found_s_fit or found_M_fit
fit: Mean=0.58, Std Dev=0.50, Var=0.25, Sample Size=220
```

```
Generated 1,000,000 hands
Produced 220 hands
Initial random seed 113
Time needed    2.249 sec
real    0m2.255s    user    0m2.237s    sys    0m0.025s
```

Double Dummy Analysis with GIB and DDS

Version 1.4 of Dealer (the version distributed by Debian) used GIB in double dummy mode, as a way of determining the number of tricks that South could take in a given strain for the deal being analyzed. Compared to the other metrics used to evaluate hands, using GIB in this way is quite slow. On order of a tenth of a second per call to GIB. (.085 secs). There are a couple of reasons for this, firstly actually solving a deal to see how many tricks can be taken is not a trivial exercise; it is much harder than just counting HCP. Secondly each call to GIB means creating a temporary file to hold the deal in, writing the deal to that file, starting up shell program which in turn starts up the GIB program, waiting for the program to complete its analysis and write to its output file, opening and reading said file, and finally getting the number of tricks. Frankly I was surprised it was a quick as it was. However I was interested in seeing if I could improve on this for two reasons; the first of course is performance, the second it that the use of GIB in this way is a bit clumsy; not everyone may have a copy of GIB (which is no longer available on Ginsberg's web site), and GIB itself needs to be installed with certain files in certain directories. Changing directories can mean copying a couple of files from elsewhere.

I therefore added functionality to Dealer to give it the ability to use Bo Haglund's Double Dummy Solver (DDS). Since GIB was designed to be a bridge playing program with a DD mode added on, and DDS was designed to be a double dummy solver, you might expect that DDS is more efficient, and offers more features than does GIB for this purpose. Such is indeed the case. To use DDS does not involve running any external program, you compile the DDS library right into Dealer. To solve one board, for one strain takes DDS only about one fifth the time it takes GIB. (0.0145 secs). But this is using DDS in the most inefficient way; DDS is optimized to obtain many results in parallel through multi-threading and re-using partial results from other calculations for the same deal. DDS can get all 20 possible results (4 declarers * 5 strains) in about the same time it would take GIB to get two results. And this is still not the best way to use DDS, but for Dealer's purposes it is more than sufficient. DDS also gives you the ability to calculate the PAR result on any given deal. However even for DDS the time taken is not trivial; doing a DDS calculation on thousands of deals will take some time.

In my implementation of DDS for Dealer I have borrowed a technique from the folks who did the GIB implementation and I have 'cached' the results as they are calculated.

At this point I should digress a bit and explain that Dealer does NOT store the results of previous calculations, but recomputes them each time they are called for.

For example if in the condition clause you ask for

```
hcp(north) >= 15 and hcp(north) <= 17
```

this will result in two calls to whatever function evaluates the HCP. Even if you specify a variable like `NorthHCP=hcp(north)` and then say `NorthHCP >=15 and NorthHCP <=17` this still results in two calls to the HCP evaluation function. In fact this is less efficient than the previous method since it involves first finding the variable NorthHCP in the variable list and from there calling the HCP function. Dealer does both the find and the function call twice, once for each time the variable is used. This is one reason why Dealer implements the 'analyze()' function described earlier; analyze() pre-computes many of the commonly used metrics and stores them in a structure so that whenever they are needed it is a simple matter of table lookup. Analyze however does NOT pre-compute the GIB (or DDS) results since a) they are not often needed, and b) they would slow down EVERY deal generated whether the deal satisfied the other conditions or not. The end result would make Dealer unbearably slow.

Therefore the GIB and DDS functions save the results of previous calls in a cache so that the second time the same result is needed on the same deal, it does not result in a second call to GIB (or DDS) but instead a simple array lookup.

End of Digression.

In my implementation of DDS for Dealer, there are two ways that DDS can be called: a) the way that returns a single result and b) the way that returns all 20 possible results. Method (a) is about 5 times as fast as method (b) so obviously if you only need one to four different results , such as comparing 4 of a major to 3NT, you are better off to use method (a). But if you want PAR calculations, or you are trying to build up a library of deals for future study and you want more than 4 different results per deal, you are better off to use method (b). See the user documentation for details.

DDS also gives the user the option of specifying how many threads to run while solving the deal(s). If you are using method (a) there is no advantage to specifying more than 1 thread. If you are using method (b) you can specify between 1 and 15 threads. On my machine with 8 cores, when using method (b) I saw a gradual improvement as I increased the number of threads from 3 to 9. After that there was no additional improvement in time taken. The code therefore ships with 9 threads as the default when using method (b). The user can override this from the command line.

Experimental Results.

The following shows the results of running Dealer with different DD solver setups, and with the DD clause before or after the other clauses in the condition statement. (see the code in the next section.) The Total Calls figures are the calls to the cached results and the calls to populate the cache. The Solve Calls figures are the number of times the actual work of solving the deal was done. (Mode 1 is single solution, mode 2 is 20 solutions)

```
../dealer Descr.DDS_slow -M2 -s17 -D2
DDS with tricks clause at start of condition -- Mode 2
Generated 395 hands
Produced 20 hands
Initial random seed 17
Time needed  100.382 sec
Tot Calls to DDS =  7100, DDS Solve Calls=   396
Tot Calls to GIB =     0, GIB Solve Calls=     0
-----------------------------------------------------
../dealer Descr.DDS_slow -M1 -s17 -D2
DDS with tricks clause at start of condition -- Mode 1
Generated 395 hands
Produced 20 hands
Initial random seed 17
Time needed   51.637 sec
Tot Calls to DDS =  7100, DDS Solve Calls=  1980
Tot Calls to GIB =     0, GIB Solve Calls=     0
-----------------------------------------------------
../dealer Descr.DDS_quick -M2 -s17 -D2
DDS with tricks clause at end of condition -- Mode 2
Generated 395 hands
Produced 20 hands
Initial random seed 17
Time needed    6.198 sec
Tot Calls to DDS =   402, DDS Solve Calls=    22
Tot Calls to GIB =     0, GIB Solve Calls=     0
------------------------------------------------------------
../dealer Descr.DDS_quick -M1 -s17 -D2
DDS with tricks clause at end of condition -- Mode 1
Generated 395 hands
Produced 20 hands
Initial random seed 17
Time needed    3.030 sec
Tot Calls to DDS =   402, DDS Solve Calls=   110
Tot Calls to GIB =     0, GIB Solve Calls=     0
-------------------------------------------------------------
```

```
../dealer Descr.GIB_quick  -s17 -D2 -p20
Tricks clause at end of condition statement; 5 calls to GIB when needed
Generated 395 hands
Produced 20 hands
Initial random seed 17
```
Time needed   8.109 sec
```
Tot Calls to DDS =     0, DDS Solve Calls=     0
Tot Calls to GIB =   402, GIB Solve Calls=   110

../dealer Descr.GIB_slower  -s17 -D2 -p20
Tricks clause at Start of condition statement; 5 calls to GIB on each hand
Generated 395 hands
Produced 20 hands
Initial random seed 17
```
Time needed 152.405 sec
```
Tot Calls to DDS =     0, DDS Solve Calls=     0
Tot Calls to GIB =  7100, GIB Solve Calls=  1980
---------------------------------------------------------------------------
 ../dealer Descr.DDS1x_slow -M1 -s17 -D2
DDS with tricks clause at START of condition only 1 call per hand
Generated 362 hands
Produced 20 hands
Initial random seed 17
```
Time needed   8.016 sec
```
Tot Calls to DDS =   363, DDS Solve Calls=   363
Tot Calls to GIB =     0, GIB Solve Calls=     0
----------------------------------------------------------------------------
 ../dealer Descr.GIB1x_slower  -s17 -D2 -p20
GIB with tricks clause at START of condition only 1 call per hand
Generated 362 hands
Produced 20 hands
Initial random seed 17
```
Time needed   30.054 sec
```
Tot Calls to DDS =     0, DDS Solve Calls=     0
Tot Calls to GIB =   363, GIB Solve Calls=   363
==============================================================
```

The condition statement used in the 5x tests

```
# These next statements calculate the maximum number of tricks North can take in any strain.
# The five calls per deal that populate the cache
nNtrix = tricks(north, notrumps)
nStrix = tricks(north, spades)
nHtrix = tricks(north, hearts)
nDtrix = tricks(north, diamonds)
nCtrix = tricks(north, clubs)
# the IF statements that generate the extra calls to the DD solver.
ntrixM = nStrix > nHtrix ? nStrix : nHtrix
ntrixm = nDtrix > nCtrix ? nDtrix : nCtrix
ntrixT = ntrixM > ntrixm ? ntrixM : ntrixm
ntrix  = ntrixT > nNtrix ? ntrixT : nNtrix
condition    ( ntrix > 6  ) &&              /* ntrix > 6 clause goes here for the slow runs */
     shape(north, any 4333 + any 4432 + any 5332) and
     hcp(north)>=15 and hcp(north)<=18 and hcp(south) <= 12    /* && (ntrix > 6) goes here for quick runs*/
# The ntrix>6 clause is at the beginning of the condition statement for the 'slow' runs, and at the end for the 'quick' runs.
=====================================================================================
```

Conclusion:
1. Avoid calling a double dummy solver on EACH hand, whether DDS or GIB
2. Caching makes a big difference especially when there are many calls using the same value of tricks.
3. DDS mode 1 is 3-4 times faster than GIB for 'ordinary' use. i.e. 1-5 double dummy calls per hand.
4. DDS mode 2 is quicker than GIB by about 30% when there are 5 calls to GIB per hand. When there are only one or two calls it is slower.
5. GIB consistently takes about 0.08 seconds per call. DDS mode 1 about 0.0017 seconds

Appendix - Adding the CSV Report

The most extensive piece of functionality I added to Dealer was the CSV report. The DDS functionality had more C code, but did not touch quite as many pieces of Dealer as the CSV report functionality. The following can be used as a kind of checklist if you want to add functionality of your own. Not every step that I took here will be needed.

1. Define the grammar in dealyacc.y
    - 1.1 a new %token for: CSVRPT, DEAL, SIDE, and TRIX.
    - 1.2 a new type in the union structure, a pointer to a `struct. struct csvterm_st *y_csv`
    - 1.3 a new %type for csvlist, `%type <y_csv> csvlist`
    - 1.4 new rules for csvlist, allowing for expressions, strings, and collections of bridge hands, and trick taking.
    - 1.5 new rules for the csvrpt action, giving the code to build the new node action node of type CSVRPT and add it to the list of actions.
    - 1.6 new rules for the various terms in the csvlist, to add a node to the list of expressions that are to by printed to the csv report. Populate the node with the type and semantic value.

2. Define the token patterns in dealflex.l
    - 2.1 when the pattern csvrpt is seen return the value of CSVRPT
    - 2.1 when the patterns for the csvlist bridge hands are seen set yylval accordingly and return the related tokens. (DEAL, TRIX, SIDE, STRING, etc.)
3. Header Files in include directory
    - 3.1 Define a new type in dealdefs.h: a struct that implements the various types of terms in the csvlist
    - 3.2 Define new variables to hold the output buffer of the report, and an array to hold the results of the trix calculation. Modify dealglobals.c and dealexterns.h accordingly.
    - 3.3 Modify the dealtypes.h file to reflect the new command line options structure.
    - 3.4 Modify the dealprotos.h file to provide an external link to the new functions in dealdds_subs.c and dealaction_subs.c

4. Modify the dealaction_subs.c file to contain the code to run when a CSVRPT type of action is found. Essentially modify the case statement in the action() function. Also add to the case statements in the setup_action() and cleanup_action functions.
5. Add code to the dealaction_subs.c file to format the deal for printout;
6. Add code to dealdds_subs.c to call the double dummy solver routines, and to change the mode of the double dummy solver, if required.

7. Add code to the mainsubs.c file, and the dealerv2.c file to process the new command line switch that gives the filename and write mode of the CSV report file.
8. Write new page in users guide.
9. Conduct regression tests to ensure I did not break anything.