Gambit Language Description

NOTES - November 19, 1993 - AMM & RDM

This file contains notes concerning the description of a command language for Gambit.

I. DESIGN PRINCIPLES

The purpose of the Gambit Command Language is to provide a method of directing the operation of Gambit that is analogous to that of high level general purpose programming languages. It should allow intermediate and advanced users to greatly expand the set of tasks they can easily accomplish with Gambit. It should provide the advanced user with a rich and flexible set of tools adequate to support complex research projects.

Although it should be capable of independent operation, the command language is expected to be used in conjunction with the tools for creation and editing of extensive form games provided by the visual interface. In addition to batch mode and visual-interactive mode, the language should also support terminal-interactive mode in conjunction with a "viewer" displaying the state of CurrentGame. The three interfaces should be coordinated in the following respects:

1. Every interactive command should have a command language version, and a byproduct of an interactive session should be a logfile containing the command language translation of what happened.
2. It should be possible to pass from an object generated by an interactive session (most commonly an extensive game) to an "efficient" block of code for generating that object. (It may be desirable to save objects in other forms as well.)
3. It should be possible to "paste" a sequence of commands into a terminal window in terminal-interactive mode, as in Mathematica.

The principles guiding language design are the same as for other high level languages: simplicity; convenience; power; ease of learning; ease of use.

To the extent possible, features and grammatical idioms are borrowed from other popular languages. Mathematica is especially influential, providing the central paradigm - functions, and the syntactic for form of function calls. The following general features should be noted.

A. The command language is interpreted, not compiled.
This is necessary in order to support terminal-interactive mode.

B. Variables are declared implicitly, with type inferred from context.
Again, if not strictly necessary, this is at least strongly suggested by the support of terminal-interactive mode.

OPTIONAL DECLARATION?

C. Variables can be passed either by value or by reference, which is indicated by placing "VAR" prior to the variable's declaration in the function declaration, as in Pascal.

As in the UNIX/C environment, combination of existing tools
by redirection of IO should be facilitated by the structure
of the program's IO behavior, and by the ability to save
information in forms that are readable by other programs.

E. Pointers are not supported (yet).

F. A rich set of commands operating on lists is available.

II. CHARACTER SET

All alphanumeric characters.
"Standard" supplementary characters - '~!@#$%^&*()_-+=|\{[]:;"'<,>.?/
(Some of these may find no use.)
space
\t (tab)
\n (newline)
\r (return)

III. COMMENTS AND FILTERING WHITE SPACE (Batch mode only)

A source file in batch mode is filtered by extracting
a. included files (saved as buffers)
b. strings (saved for later reinclusion)
c. \\n combinations
d. comments
then replacing each sequence of space and \t with a single space.
At this point strings are reinserted. This amounts to the
following conventions.

backslash at end of line eliminates subsequent sequence of \n and \r
up to the first occurrence of another

/* - multiple line comments - */
// - single line comment (arbitrary length) \n

Multiple pieces of white space: each sequence of space and \t
is reduced to a single space, and spaces and \t's at the beginnings
and ends of lines are eliminated.

IV. PROGRAM DESCRIPTION

After filtering of comments and whitespace a correct input file is
a sequence of "statements" separated by sequences of
";", "\n", and "\r". (In the remainder ";" will designate the
statement terminator.) In addition to statements allowed in
terminal-interactive mode, batch mode supports the following two
types of statements.

1. Batch mode supports the inclusion of named files in the source code
(as in shell scripts), according to the idiom

```
/## - FILENAME
The first character in this file is the 'T' at the beginning of this line.
The last character in this file is the '.' at the end of this line.
##/EOF
```

n the terminology of this description, each included file is a "statement." This feature is useful for incorporating the specification of games into the text, for instance to send to another user.

2. Batch mode supports a number of "interpreter directives," which are also "statements", and may occur between any pair of statements. A sample interpreter directive displaying the idiom is

```
/! - BeginnerMode //Verbose warnings
```

As the language is developed, interpreter commands should be developed for each class of warnings, so that the advanced user can access the messages of interest without being overwhelmed or annoyed by messages whose import he/she has alrady chosen to disregard.

V. TOKEN DESCRIPTION

Roughly the parser will need to recognize the following elements.

Comment symbols - /* */ //

Source code file initiator and terminator - \n/## ##/EOF ← \N - not strictly true

String initiator and terminator - "

Compiler directive preface - \n/!

Single character tokens - ; , ( ) { } [ ] + - / * ^ "

Comparison operators - = != < > <= => <>

Logical operators - && || ! ← prefered to AND, OR, NOT ?

Assignment - := -> <->

Names - A string consisting of an alphabetic character followed by alphanumeric characters (including "_") possibly terminated by '?' to indicate a boolean query ([a-zA-Z][a-zA-Z0-9_]+\?) [or something like this]

Nonnegative Integer -

Signed Integer -

Floating Point Number -

Reserved Words - if else elseif then do done while begin end (curly brackets will be treated as synonymous) and or (&& and || treated synonymously) break return for VAR

ELSIF?

TRUE, FALSE, NOT

Format elements - %d, %s, %f, %e ← what about extended types (NODE, etc)

VI. FUNCTION FORMAT

As in Mathematica, the central paradigm is that we think

f everything as a function, which may return a value, and which may or may not have some special shorthand format. Arguments are in the form Argumentname->Argumentvalue. In this format it is permissible to abbreviate Argumentname by giving an initial segment sufficient to distinguish it from other argument names of the function.

We allow variables to be passed both by reference (preferencing the variable declaration by VAR, as in Pascal) and by value. An attempt at a formal description of the grammar of a function call is:

```
expression:   name
    | function
    ;

argumentname: name
    ;

argumentvalue: expression
    ;

argument: argumentname -> argumentvalue
    | argumentname <-> argumentvalue
    ;

argumentlist:
    | argument
    | argument, argumentlist
    ;

function: functionname[argumentlist];
    ;
```

NOTE - The construction 'argumentname<->argumentvalue' is required when the argument is variable, i.e. preceeded by VAR in the function declaration.

The other principal type of statement is function definition. The idea is to emulate Mathematica here as well, and rather than attempting a careful description of the syntax, it seems more to the point to simply refer to the Mathematica manual. The principal difference that seems likely is that in the variable list of the function being declared, the type will be explicit, as in the the function descriptions below.

VII. DATA TYPES

This section contains a partial enumeration of the main data types considered by the language, with a sketchy verbal description and a sample of calls applied to these types, emphasizing those calls that elicit attributes.

all standard C data types, and also bool (boolean) and string

lists (ordered, with elements of any type)
CALLS
AttachListToList[list TargetList, list AttachmentList] - what we have in mind is that an element of TargetList should

```
    have a unique pointer to the type of the elements of
    AttachmentList
      ********* TED - CAN WE GET AWAY WITH THIS?? ***********

  [Something approximating the list handling features of Mathematica]

game - an extensive form game, consisting of a collection of
  nodes, players, information sets, outcomes, actions,
  the assignments of payoffs to (outcome,player) pairs,
  and the assignment of probabilities to actions controlled
  by chance.
    CALLS
    node RootNode[] - returns the root node of the game
      OPTION
        game Game - default is CurrentGame

    list PlayerList[] - returns list of players
      OPTION
        game Game - default is CurrentGame

    list NodeList[] - returns list of nodes
      OPTION
        game Game - default is CurrentGame

    list ActionList[] - returns list of actions
      OPTION
        game Game - default is CurrentGame

    list ISetList[] - returns list of isets
      OPTION
        game Game - default is CurrentGame

    list OutcomeList[] - returns list of outcomes
      (orders of each list are described below)
      OPTION
        game Game - default is CurrentGame

node - a node in an extensive form game
    CALLS
    void NameNode[node Node, name Nodename] - assigns a name to a node,
      by which it can be referred to later

    name NodeName[node Node] - returns name of Node, if it has one
      (at birth a nodes name is null)

    node Parent[node Node] - returns Node's parent (null if Node is root)

    int NumberOfChildren[node Node] - returns the number of children of
      Node

    node NthChild[node Node, int Number] - returns
      Number'th child of Node, with error if Number too large

    int ChildNumber[node Node] - returns the number of siblings at least
      as old as Node, including Node (returns 0 if Node is root)

    integerlist ChildNumberSequence[node Node] - returns list of
      ChildNumbers of Node's ancestors and Node itself
```

how lists container, a container of a certain type?

this is not the same as "Game" class we have now-beware!

161355

child-ref notation ... or something..

```
    iset ISetOfNode[node Node]

    bool NodeIsRoot?[node Node] - boolean query

    bool NodeIsTerminal?[node Node] - boolean query

    outcome OutcomeOfNode[node Node] - returns outcome attached to Node


iset - an information set in an extensive form game, consisting of
  a collection of nodes, at which a particular player can choose
  from a collection of actions
    CALLS
    void NameISet[iset ISet, name ISetName] - assigns ISetName to ISet

    name ISetName[iset ISet] - returns name of Iset, if it has one

    player PlayerOfISet[iset ISet] - the player who controls the choice
      of move at ISet (typically intialized to Dummy)

    int NumberOfActionsAtISet[iset ISet] - returns the number of
      action available at ISet

    action NthAction[iset ISet, int Number] - returns
      Number'th action available at ISet, with error if Number too large

    list NodeListOfISet[iset ISet] - returns list of nodes in ISet, in
      the lexicographical order described below

    list ActionListOfISet[iset ISet] - returns list of actions that can
      be chosen at ISet, in the lexicographical order described below

  node NthNodeInISet[iset ISet, int Number] - the nodes are ordered
    lexicographically, according to ChildNumberSequence,
    with the wayward child of the seventh son of the heir of root
    succeeding the seventh son of the heir, but preceeding
    the heir of the seventh son of root


action - the moves that may be chosen at the VARious information sets
    CALLS
    void NameAction[action Action, name Name] - assigns
      Name to Action

    name ActionName[action Action] - returns name of Action, if it
      has one

    iset ISetOfAction[action Action] - returns the information set
      at which Action may be chosen

    rational ProbabilityOfAction[action Action] - returns the
      probability that the action will be chosen if the information
      set of the action is controlled by Chance, and otherwise
      is an error


player - the actors who choose moves and receive payoffs
    CALLS
    void NamePlayer[player Player, name Name] - assigns Name to Player
```

```
      name PlayerName[player Player] - returns name of Player, if
        he/she has one

      list ISetListOfPlayer[player Player] - returns list of information
        sets at which Player chooses the action


  outcome - different terminal nodes may be equivalent in the rules of
    the game, and outcomes are equivalence classes
      CALLS
      void NameOutcome[outcome Outcome, name->outcome_name] - assigns
        outcome_name to Outcome

      name OutcomeName[outcome Outcome] - returns name of Outcome,
        if it has one

      rational Payoff[player Player, outcome Outcome] - returns the payoff of
        player Player at outcome Outcome

  normalform - a normal form game consists of a list of agents,
    sets of pure strategies for each agent, a function mapping
    vectors of strategies to probabilities over outcomes, and a
    payoff funtion mapping outcomes to vectors of rationals
    In addition there is an associated extensive game, which
    should be fixed by default in the language except when the
    (expert) user explicitly reassigns it.
      CALLS
      list NormalFormPlayerList[] - returns list of players
        OPTION
          normalform NormalForm - default is CurrentNormalForm

      list NormalFormStrategyList[player Player] - returns list
        of pure strategies for Player
        OPTION
          normalform NormalForm - default is CurrentNormalForm

      list NormalFormOutcomeList[] - returns list of outcomes
        (orders of each list are described below)
        OPTION
          normalform NormalForm - default is CurrentNormalForm

      game ExtensiveFormAntecedent[] - returns associated
        extensive form of NormalForm
        OPTION
          normalform NormalForm - default is CurrentNormalForm


  extendedinteger
  rational
  polynom
```

VIII. GLOBAL VARIABLES

Certain global variables are maintained to allow for different modes of behavior. It should be possible to reset all of these that pertain to terminal-interactive mode by direct assignment statements. For those that pertain only to batch mode, it seems more appropriate to use the format of interpreter directives, but it may also be sensible to allow direct assignments.

```
    bool InterimInvalidGamesAllowed - if False then error messages
      to the user are sent after any maneuver that creates an unsolvable
      structure

    int Mode - potential modes include FiniteExtensive (default), Repeated,
      NormalForm, and Stochastic
```

In anticipation that the user will frequently be dealing with a single game, so that it would be onerous, and clutter the source code, if this had to be declared explicitly at each stage, we provide the following objects that are default variables in appropriate commands.

```
    game CurrentGame - points to NullGame until initialized, used to make the
      Game argument optional for many functions

    normalform CurrentNormalForm -  points to NullNormalForm until
      initialized, used to make the NormalForm argument optional
      for normal form functions
```

IX. FUNCTIONS FOR TREE BUILDING AND MANIPULATION

This sections begins the description of the specifics of the language by giving a collection of commands for the construction and rearrangement of an extensive game.

```
      game NewGame[] - This creates a game consisting of a single
        node, with null outcome, with
        Nature as the only player, and NullOutcome attached, and sets
        CurrentGame to return value
        OPTION
          bool DoNotResetCurrentGame

      node AppendNode[node Node] - adds a new branch beginning at
        Node and ending at a terminal node, which is the new
        youngest descendant of Node -
        creates a singleton information set containing
        Node and controlled by Dummy (error if Node is nonterminal)
        the new node(s) are assigned NullOutcome, and
        the assignment of outcome to Node is changed to NullOutcome
        OPTIONS
          int Branches - specify number of new branches (returns first)
          iset ISet - specify information set Node is added to
          player Player - specify player who owns newly
                    created singleton information set
          bool RetainOutcome - retain outcome assigned to Node

      node InsertNode[node Node] - insert a new node between a child
        and its parent - if child is root, then
        root is reassigned to new node
        OPTIONS
          int Branches - number of new (younger) branches (NullOutcome assigned
                  to new terminal nodes)
          bool Youngest - new branches are older if TRUE
          iset ISet  - specify information set containing Node
          player Player - specify player owning singleton information set
              containing Node
              (default is a singleton information set for Dummy)
```

```
void DeleteSubtree[node Node] - deletes all nodes descended from Node
  and removes all deleted nodes from containing information sets
  removes Node from its containing information set, and deletes
  newly emptied information sets
  OPTIONS
    bool RetainEmptiedISets - retains emptied information sets
    bool RemoveDeadPlayers - remove players who no longer have
      information sets
    outcome Outcome - Outcome is attached to Node

void AssignPlayerToISet[player Player, iset ISet]

iset AddISet[] - the player of the new information set
  is Dummy
  OPTIONS
    name Name - name of new iset is Name
    player Player - player at new iset is Player
    game Game - default is CurrentGame

void DeleteISet[iset ISet]
  OPTION
    game Game - default is CurrentGame

void MergeISets[iset ISet1, ... , iset ISetn] - Iset1 retained,
  with other information sets discarded from list of isets for the game
  matches action numbers of different isets, retaining names in ISet1
  error if different number of actions at two isets, and also
  if different players assigned to two isets

iset SplitISet[iset Source, node Node1, ... , node Nodek] -
  n1, ... nk are ejected from Source, and placed in the newly created
  iset.  Same errors as MergeISets.  Returns newly created iset.

void NewAction[iset ISet] - adds new branches beginning at each node
  in ISet and ending at terminal nodes, which are the new youngest
  descendants of these nodes
  OPTIONS
    int NumberElderSibs - specify number of older siblings of new
                          branches
    int Branches - specify number of new branches

void DeleteAction[action Action] - deletes all branches
  associated with Action, including all descendant
  nodes, removes all deleted nodes from containing information
  sets (including nodes in the information set containing Action
  if they become terminal), and deletes newly emptied information sets
  OPTIONS
    bool RetainEmptiedISets - retains emptied information sets
    bool RemoveDeadPlayers - remove players who no longer have information sets

void DemoteAction[action Action] - interchanges the number of a
  and the next younger sibling - error if Action is youngest
  OPTION
    int Number - specify number of demotions

void AssignOutcome[node Node, outcome Outcome] - makes Outcome the
  outcome of Node

void AssignOutcomeList[] - attaches the current outcome
```

```
      list to the the current list of terminal nodes
      OPTIONS
        list OutcomeList - list of outcomes assigned
        list NodeList - list of nodes (not necessarily terminal)

    player AddPlayer[] - adds a new player to the player list of
      Game. Payoffs of this player for all outcomes are
      initialized to zero
      OPTIONS
        name Name - the newly created player is named Name
        game Game - default is CurrentGame

    outcome AddOutcome[] - adds a new outcome to the outcome list
      of the game. Payoffs of existing players are initialized to zero
      OPTION
        name Name - assigns the name Name to the newly created outcome
        game Game - default is CurrentGame

    void AssignPayoff[outcome Outcome, player Player, rational Payoff] - sets
      payoff to Player in Outcome equal to Payoff

    void AssignPayoffVectors[list OutcomeList, list PayoffVectorList]

    void AssignProbability[outcome Action, rational Probability] - sets
      probability of Action to Probability if Action is controlled by
      Chance, with error otherwise

    void AssignProbabilities[list ActionList,list ProbabilityList]

    game AgentExtensiveForm[] - returns the agent extensive form
      (each information set treated as a different player)
      derived from Game
      OPTION
        game Game - default is CurrentGame
        bool DoNotResetCurrentNormalForm


X. SAMPLE PROGRAM - Constructing a Prisoners' Dilemma

//Get Going
PrisonersDilemma := NewGame[]
cursor := RootNode[]


//Add Players and Outcomes to Game
AddPlayerToGame[P->Alphonse]
AddPlayerToGame[P->Gaston]
AddOutcomeToGame[Num->4, NameL->("CC","CD","DC","DD")]


//Add Branches at Root
AppendNode[N->cursor, Br->2, P->Alphonse]


//Add Grandchildren of Root
cursor := NthChild[Node->cursor, Num->1]
iset1 := ISetOfNode[Node->cursor]

cursor := NthChild[Node->Parent[cursor], Num->2]
iset2 := ISetOfNode[N->cursor]
```

← GUI allows implicit player initialization

```
//Create the Information Set for Gaston
MergeISets[ISetl->isetl,ISet2->iset2]


//Assign Outcomes, Action Names, and Payoffs
    //Assign Outcomes According to Order of Current Lists
AttachOutcomeList[]
    //Assign Action Names
AttachNameList[L->ActionListOfISet[I->isetl], NameL->{"C_Alp","D_Alp"}]
AttachNameList[L->ActionListOfISet[I->iset2], NameL->{"C_Gas","D_Gas"}]
    //Assign Payoffs According to Order of Outcome List
AssignPayoffVectors[PayoffL->{{9,9},{0,10},{10, 0},{1,1}}]
```

XI. NORMAL FORM FUNCTIONS

When one passes from the extensive form to one of the possible normal forms, one is typically interested not only in the given and produced objects, but also in the correspondence between them. In the routines of this sort below we specify VAR variables called lexicons, which are thought of as lists containing the relevant information. It is important to note that the contents of the lists may vary across the various types of normal form. In the following routines it is assumed that part of a normal form is a pointer to the extensive game from which it was derived.

normalform AgentNormalForm[] - returns the agent normal form derived from Game. We do not provide special routines for building normal form games, since it is no more difficult to build the corresponding extensive form, then apply this function
Sets CurrentNormalForm to return value
OPTIONS
game Game - default is CurrentGame
bool DoNotResetCurrentNormalForm
VAR list Lexicon

normalform NormalFormOfGame[] - returns the normal form derived from Game, setting its antecendent extensive form to Game
Sets CurrentNormalForm to return value
OPTIONS
game Game - default is CurrentGame
bool DoNotResetCurrentNormalForm
VAR list Lexicon

normalform TerminalReducedNormalFormOfGame[] - returns the reduced normal form derived from Game by identifying all pure strategies that are equivalent in terms of the terminal node reached.
The antecendent extensive form is set to Game.
Sets CurrentNormalForm to return value.
OPTIONS
game Game - default is CurrentGame
bool DoNotResetCurrentNormalForm
VAR list Lexicon

normalform OutcomeReducedNormalFormOfGame[] - returns the reduced normal form derived from Game by identifying all pure strategies that are equivalent in terms of the outcome reached.
The antecendent extensive form is set to Game.

```
      Sets CurrentNormalForm to return value.
      OPTIONS
        game Game - default is CurrentGame
        bool DoNotResetCurrentNormalForm
        VAR list Lexicon

    normalform PayoffReducedNormalFormOfGame[] - returns the reduced
      normal form derived from Game by identifying all pure strategies
      that are equivalent in terms of the payoff vectors reached.
      The antecendent extensive form is set to Game.
      Sets CurrentNormalForm to return value.
      OPTIONS
        game Game - default is CurrentGame
        bool DoNotResetCurrentNormalForm
        VAR list Lexicon

    normalform MixedPayoffReducedNormalFormOfGame[] - returns the reduced
      normal form derived from Game by eliminating all pure strategies
      that are payoff-equivalent to mixtures of othe pure strategies.
      The antecendent extensive form is set to Game.
      Sets CurrentNormalForm to return value.
      OPTIONS
        game Game - default is CurrentGame
        bool DoNotResetCurrentNormalForm
        VAR list Lexicon

    normalform LoadNormalForm[file InputFile] - loads a normal form from
      disk, and sets the antecedent extensive form to the canonical
      extensive form.  Both CurrentNormalForm and Current Game are
      reset by default.
      OPTIONS
        bool DoNotResetCurrentNormalForm
        bool DoNotResetCurrentGame

    game CanonicalExtensiveGame[] - returns the canonical extensive
      form associated with NormalForm, resetting CurrentGame
      OPTIONS
        normalform NormalForm - default is CurrentNormalForm
        bool DoNotResetCurrentGame
        VAR list Lexicon

    game ExtensiveGameOfNormalForm[] - returns the extensive
      form associated with NormalForm.
      OPTION
        normalform NormalForm - default is CurrentNormalForm


**************************

IMPLIED REQUIRED CHANGES TO CODE

  This portion of the file is a scratch area for notes
concerning changes to the current code that are required
by the design of the command language.

1.  Make behavioral strategy n-tuples and belief
  n-tuples separate structures:  lists of "probability elements" -- each
  probability element is a number and a pointer to an action (node)
```