Report Number: WUCS-93-25

1993-01-01

The Study of Computer Science Concepts through Game Play

Benjamin M. Weber

The Study of Computer Science Concepts through Game Play

Benjamin M. Weber

WUCS-93-25

May 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

THE STUDY OF COMPUTER SCIENCE CONCEPTS THROUGH GAME PLAY

by

Benjamin M. Weber

Prepared under the direction of Professor Ronald P. Loui

An Honors Thesis presented to the Sever Institute of

Washington University in partial fulfillment

of the requirements for the degree of

BACHELOR OF SCIENCE

WITH DISTINCTION

To

My parents and family whose support and love made

my education possible

and to Paulette for keeping me sane.

I. Introduction

That games and simulations can be a useful tool in the learning process is a well-known fact. As early as the eighteenth century, games and simulations were used in the field of military training. In the 1950's, business management training programs recognized its benefits as a means of developing decision-making skills. In the early 1960's, use of these techniques spread into education, most specifically within teacher training the social sciences. Finally, from the 1970's on, gaming and simulation techniques spread to a constantly increasing range of other subjects, including the teaching of science and mathematics (Ellington et al., 1981).

In each of these aforementioned fields of study, game play has been recognized to be very useful. Not just for enjoyment purposes, well-designed games can achieve positive transfer of learning - the ability of participants to apply skills acquired during the exercise in other situations (Twelker, 1981). A further bonus of the game approach is that games are able to make normally tedious material something that the student will enjoy studying - even during free time. Learning takes on new meaning and more attention is centered on the problems at hand. In addition, for games in which a competitive element is included, the participants have even stronger motivation to focus their attention on the problem at hand. This increase in attentiveness obviously will have a positive effect on the amount learned by the student.

Other benefits of games over ordinary study techniques may not be as obvious, but remain important. First, The ease with which game rules and situations can be constructed to meet specific situations or sets of constraints enables the teacher or researcher to focus attention on specific areas of study. Games also include a much greater stimulus for participants to use and develop creative thought. This is quite a positive trait for budding computer scientists, as our field cultivates divergent thought processes. Additionally, gaming involves participation and often cooperation between people, skills which are vital to education as well as positive qualities for people in general.

Finally, in designing games, there is the need to specify a set of rules and to determine if these rules, and the game itself, conform to the specifics of the concept on which it is based. This forces both players and game designers to study the concept in greater detail then would otherwise be necessary. An example of this is discussed below within the analysis of the Argumentation Game. For all of these reasons, and many more which may depend on the specific topic and game, study through game play can be a extraordinarily powerful learning tool.

So far, the term "game" has been used where "game or simulation" would be more appropriate. A "game" is defined as an activity which is carried out by cooperating or competing decision-makers seeking to achieve their objectives within a framework of

rules (Gibbs 1974a). In contrast, a "simulation" is a dynamic representation which uses substitute components and relationships to replace their real or hypothetical counterparts (Gibbs 1974b). Although many researchers stress this distinction, for the purposes of this thesis it is irrelevant, as both games and simulations have the potential for providing all of the benefits discussed and for the same reasons. Thus, "games" will be the term of choice henceforth.

II. Computer Science Concepts Casted as Games

"How do general concepts or topics of study become games?" is an important consideration in using games to study Computer Science. Almost anything can become a game, if thought is put into its development. For a game to be successful, it must teach the concept well while providing enjoyment for the player and initiating the benefits discussed above. The possibilities for this are endless.

The process by which a game is developed and exploited has been divided into three distinct phases (Ellington et al., 1981). The first is the development of the basic idea. Here, designers discuss the concepts to be taught by the game, the basic idea of the game, and the choice of structure for the game. The next, and generally longest, design phase is the process by which this general idea is transformed into a viable educational package and tested for its effectiveness. Here, the task of the designer is to convert knowledge of the concept within the realm of Computer Science to the rules and structure of the game. The success of the game design

is thus determined by the ease with which this knowledge is transferred to future game participants. After a design is completed, it should be tested to see how well it emphasizes the important aspects of the concept, refining the rules as problems are found. Note that, in the author's experience, it is much easier to work on the first two phases with another person (or group of people). The interchange of ideas speeds up the development process and gives rise to a better final project. Also, in multiple player games, it is simpler to analyze critically the game when using the approximate group size that the game requires.

The third and final phase of development is the exploitation of the exercise. After the game is created and tested, it should be presented to students for use in their studies. Care should be taken to describe how the game relates to the target concepts and to the field of Computer Science in general. If this facet is overlooked, game players will have proficiency with the concept itself, but may not realize this or understand how it fits in to the "big picture". Often, the success of the exercise depends on this. Also, if the game is sufficiently successful, thought should be given to publishing the discoveries in appropriate journals.

III. Three Games

At this point, three games will be introduced. Each was designed in accordance with the above procedure and can be powerful for the study of a particular area of Computer Science. For each

game, the rules will be presented in the form which would be distributed to potential players. After that, the benefits of the game - what is taught in a novel or powerful way - will be discussed.

III.a The Grammar Game

III.a.1 Rules

The Grammar Game Rules

Context-free and context-sensitive grammars (grammars, hereafter) are an important part of Computer Science, especially because of their use in describing the syntax of programming languages. The Grammar Game is intended to make a student comfortable with grammars. It teaches substitutions in grammars: what substitutions are possible with a given grammar and what is needed in a grammar to make further substitutions (a feature well suited for Computer Science considerations).

Before the Grammar Game is defined, some discussion of grammars and their terminology is in order.

- A grammar for a particular language consists of one or more productions, each occurring on its own line.
- A production states some of the transitions that language symbols (or sets thereof) can undergo and consists of three parts:

1) A head, the syntactic category on the left hand side of the arrow, which indicates the group of symbols that can be changed,
2) An '->', separating the left hand side (LHS) from the right hand side (RHS),
3) A body consisting of zero or more syntactic categories and/or terminals on the right hand side of the arrow.

- A syntactic category is any string in the language.
- Given a string which contains (as a part or the entire string) one of the heads in the grammar, it is possible to substitute any of the syntactic categories in the body. The initial string is thus changed by substituting that syntactic category for the portion of the given string identical to the head.
- Productions which contain zero syntactic categories are said to be ε-productions, allowing the substitution of the null string for the head.

Number of Players: Two or more players compete against each other.

The Pack: Three packs of cards with jokers removed are recommended for two players plus an additional pack for each additional 2 players. The cards are placed (shuffled) into a pot, the initial production is dealt, and then players remove cards from the pot as required (possibly in groups of indeterminate size). Jokers are removed from the decks.

Additional tools: Pencil and paper is normally needed, both for scoring and for scratch work on productions and substitutions.

The Cards: Color and suit are irrelevant to the game. Cards turned upside-down indicate either separation of the head from the RHS or a choice (an "or") between syntactic categories within the body.

Object of Play: The object is to score points (thirty is advised) by being the first player successfully to reveal a production "solving" the task.

The Task: The task consists of a list of one or more syntactic categories for both the initial string and the final string. Multiple possibilities within the initial and/or final string are separated by upside-down cards. The task is met by changing any of the initial strings to any of the final strings using a list of the substitutions contained within the current grammar's production rules.

Production Rules: The body is separated from the head by an upside-down card. Each of the possibilities within the body, if more than one, are to be separated by an upside-down card. For example, given the production:

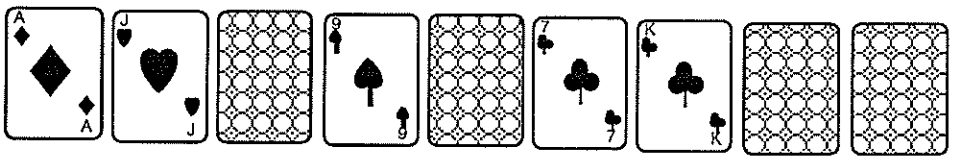

Figure 1: Example Production Rule

it is possible to change K-A-J-4 to K-9-4, K-7-K-4, or K-4, as the two upside-down cards next to each other indicate an Epsilon-production (substitute the null string for the LHS). From now on, suits will be disregarded and * will be used for an upside-down card. Also note that, given a LHS of A and a RHS of A-J, it is possible to produce an A followed by any number of J's through repeated substitution (some or all of which could then be used in subsequent substitutions).

A rule consisting of only a LHS is an ε-production and instances of that LHS may only have null substituted for them. Rules with RHS's may or may not, depending on the current status of the game, have ε-productions, indicated by two upside-down cards next to each other. Such a production is shown in Figure 1. Some examples of valid transitions are shown following the rules section.

The Deal: The players decide the form of the initial grammar and task. Any number of cards may be placed as the head, as well as any number of 'or's in the body, for each of the possibly multiple productions. The task also may be of any desired length.

Scoring: A player gains 5 points for the successful substitution from task LHS to RHS of each turn. Stopping the turn for a failed production loses 3 points.

Rules of Play: The game consists of a series of turns until a player achieves a set score (20 is recommended). To end a turn , a player indicates that they are ready (by saying loudly, "STOP!" or slapping

the table - slapping opponents is not allowed) and demonstrates a successful transition from the initial string to the final string using the rules. After stopping the turn, the player may add as many upside-down cards as desired, but none may be placed face-up. This allows the addition of ε-productions, making it unnecessary to add ε-productions hurriedly when a player first sees a transition..

At a given point in a turn, if none of the players is able to demonstrate such a transition (within a rational amount of time - verbal agreements to go on are not normally required), the next player (going clockwise) may add one face-up card to any production at any position in the production. In a given rule, it is therefore legal to add the card on the LHS, or on the RHS within a current substitution possibility, or as a new substitution possibility on the RHS. Any number of upside-down cards may also be added (if the player wants to add Epsilon-productions, for example). Similarly, a new production may be created (though it would begin as an ε-production) as only one card may be laid face-up during that particular player's turn..

After a turn, the player who received the points must change one of the rules or the task so that they can find no current successful productions. The player may achieve this by adding a single card of his or her choice somewhere in the task or within one of the rules, or by deleting as many of the productions from the grammar as desired. For example, if the task was to convert Q-A to K-Q-A, it would be a valid move to add a 9 to the Q-A making it Q-9-

A, 9-Q-A, or Q-A-9. The same sort of additions may be done to one of the rules either to the head or to one of the syntactic categories. If that player makes a change for which there is an existing successful transition within the grammar (possibly by making an irrelevant change), any other player may immediately signal stop and gain 5 points.

Example Transitions: This section will show some transitions to help reveal some of the possibilities. The notation, which is suggested for the scratch pad as well, is

LHS -> RHS where RHS is a list of sequences of cards such as

J-K-4 | 3-2-6 | |

Given the following grammar:

6 -> 10 | 4-Q

3 -> 3-3 | | K

Q-K -> A-2-A

it is possible to complete the task of changing 6-3 to 4-A-2-A-3 by the following substitutions:

6-3 > 4-Q-3 > 4-Q-3-3 > 4-Q-K-3 > 4-A-2-A-3

This could be reported by reading the ">" as "goes-to".

Another vital consideration, given a task and a grammar, what additions to the grammar could complete the task. As an example of this, consider the task:

LHS: 5 | 5-K | 6-2 RHS: 3 | 3-6 | 5-3-5-2

and the grammar:

5 ->

6 -> 5 | 6-5

7-Q ->

K -> K-3-K | K-2

A good possibility is to notice that 6-2 > 6-5-2 > 6-5-5-2 > 5-3-5-2 and try to flip a 3 on the RHS of the first production. An alternate possibility would be to flip a 5 on the RHS of the last production and to add an epsilon-production, making the rule K->K-3-K | K-2 | 5 | | and giving:

5-K > 5-K-3-K > 5-K-3-K-2 > 5-K-3-5-2 > 5-3-5-2.

Game Variations: There are many ways to change or adapt the Grammar Game. For example,

- Disallowing the scratch pad, forcing the players to keep the transitions in their heads. This can get quite difficult for large grammars, so care is necesary to avoid incorrect stoppages of play (which costs players points).
- Add a rule which says that placing an upside-down card to the left of the head (as the leftmost card in the production) disallows the addition of cards to the head from then on. This hinders opponents from playing "defensively" and protects rules a player believes are going to be important.
- Add a rule stating that additions may only be made on the end of productions during a turn. This reduces the possibilities and makes it more difficult to find certain transitions, but it also makes the game more predictable for the same reasons.

III.a.2 Grammar Game Discussion

The Grammar Game can be an effective tool for the study of grammars within Computer Science. It has a large competitive component which keeps the players constantly on their toes. The concentration on the current state of the grammar teaches the players manipulation strategies at a level not normally reached in conventional education. In addition, there is a need to think ahead for changes to the current grammar which may enable a successful transition. This feature of the game is sorely lacking in most conventional treatments of grammars, but is vital to the uses of grammars within Computer Science.

To test the clarity of the rules and the success of the Grammar Game at conveying an understanding of grammars, the author taught the game to a 16 year-old high school student who had no previous exposure to grammars. After becoming accustomed to the terms and symbols of the game, the subject began to compete excitedly with the author and even won a few turns. In approximately 30 minutes, he progressed from no knowledge to a functikonal understanding of how grammars and their productions are manipulated.

III.b The Argumentation Game

III.b.1 Rules

Argumentation Game Rules

This game is designed to investigate protocols and strategies for resource-bounded disputation. The rules presented here correspond closely to the Computer Science problem of controlling search in an actual program (Loui and Chen, 1992).

This game is quite complex in both terminology and rules, so every term which needs defining will be underlined at its definition. It is recommended that when an unknown term is found, its definition (which may be located later in the instructions) should be read immediately. Much of the rules are presented in list form to facilitate the learning of them.

The Deck: Four decks of cards, without jokers, are used. The cards are placed (shuffled) into a pot, from which the board, as well as any resource cards, are drawn and returned at the appropriate time.

The Cards: Cards are distinguished by color and value, not by suit. A card's opposite is a card of the same value and opposing color while its equivalent is a card with the same value and color.

The Players: There are two players, designated in advance to be red and black. After the bidding described below is complete, one player will be assigned the task of arguing for a specific card and is called the declarer. The opponent, or defender, must hinder the declarer by questioning the declarer's argument(s) and/or arguing for the opposite card.

Object of Play: To successfully complete the declarer's or defender's task.

The Deal:

- one of the two players is designated the initial dealer.
- deal alternates between the players.
- cards are placed mixed with faces down in a pool.
- first, three evidence cards, facts which are deemed indisputable, are drawn from the pool. If a card is drawn whose value (regardless of color) is already evidence, it is returned to the pool.
- the evidence cards are placed in the evidence area.
- next draw 10 short cases and 10 long cases.
- a case consists of a decision, face-up, upon either 2 (for short cases) or 10 (for long cases) face-down cards called facts. A case is said to be for its decision. When cards face-down are later displayed, these are said to be displayed facts of the case. A case with any displayed facts is opened and a case with all cards displayed is exhausted.

Bidding: The dealer opens the bidding and then the players alternate bids until one of the players, called the defender, challenges the other, called the declarer, to complete the last bid. Each bid consists of:

- a claim, the card to be shown by argument, which must have the same color as the bidder and for which there exists at least one red and one black decision for that card on the board.
- a number of resource cards, which are used up by the declarer and given to declarer by the defender when searching through cases at times in the game.
- a non-negative number of burden shift cards (whose use will be described later).
- a difficulty level.

The degree of difficulty is ranked lexicographically from highest to lowest as shown in Figure 2. Each bid must be stronger than all of the previous ones, with bids ordered first by increasing level of difficulty, proceeding downwards on the above chart, (sideways moves are permitted but do not strengthen a bid) then by decreasing burden shift cards and finally by decreasing resource cards.

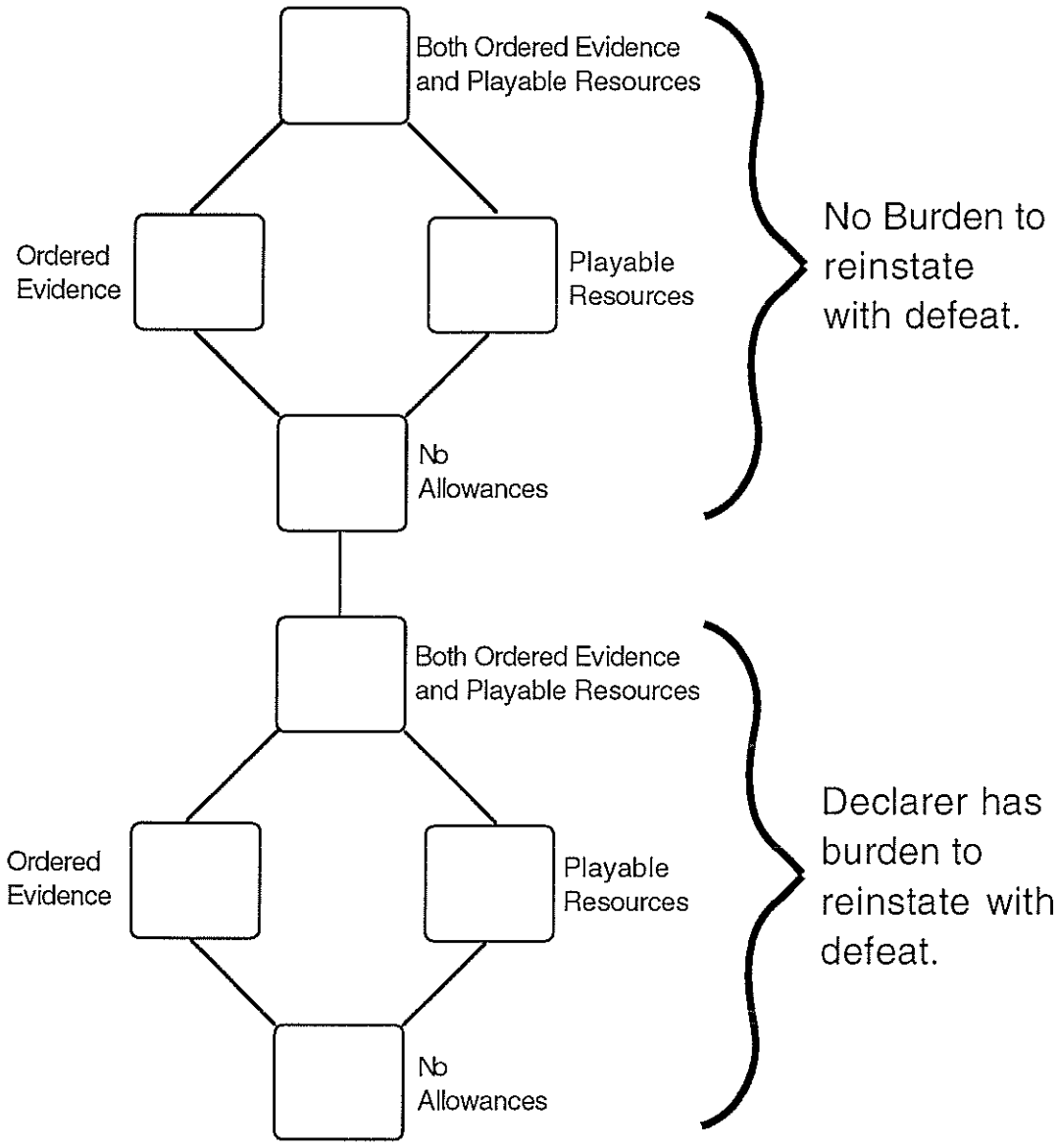

Figure 2: Difficulty Ordering

Object of Game: The declarer attempts to fulfill the bid by successfully supporting the claim within declarer's resources and burden cards, and with the constraints of the difficulty level. The defender attempts to foil the declarer by disproving or calling declarer's argument into question.

Start of Game: The initial declarer's resources are drawn from the pool, the number of which was determined by the winning bid. Jokers are given to the declarer as burden shift cards as necessary.

The Play: The declarer initiates play, after which play strictly alternates. A player must end his or her turn by engaging in dialogue that establishes sufficiency. In many instances this may be tacit - as long as the dialogue is unambiguous and both players understand it, the dialogue step may be omitted. Play terminates and defender wins if it is the declarer's turn and declarer has exhausted his or her resources. Play terminates and the opponent wins if either player is unable to make a sufficient response.

The Dialogue: A dialogue move consists of one of the following:

- challenging a card included in the opponent's last dialogue,
- refuting a card included in the opponent's last dialogue,
- an argument list supporting the claim which has appropriate strength and it is the declarer's turn, or
- an argument list against the claim which has appropriate strength and it is the defender's turn.

The Challenge: A challenge is sufficient if the object of the challenge is unsupported somewhere in the opponent's dialogue (which must have been either a refutation or argument list) and is not an evidence card. Such a challenge remains in effect and can not be used in an argument until the card is supported.

The Refutation: A refutation is sufficient if the card being refuted (1) is not an evidence card; (2) appeared in the opponent's dialogue (which was an argument list); and (3) is an argument is cited with appropriate strength. Such a refutation remains in effect, making that card unusable in an argument until the card is supported.

The Argument: An argument for a card is a collection of cases that can be organized into a tree by the following steps:

- taking the card to be the root;
- taking a unique case for the card to define the children of the root by using any (including possibly all) of its admissible facts as children of the root;
- checking that the leaves of the tree are each admissible.

The Argument List: An argument list is a collection of arguments. It has appropriate strength if each of its component arguments also have appropriate strength.

Appropriate Strength: An argument has appropriate strength if:

- it contains no currently challenged or refuted cards; and
- it is either for the declarer who has no burden to reinstate with defeat or it is for the defender and the argument is not less specific than any argument for the opposite card in the opponent's dialogue; or
- it is for the declarer who has burden to reinstate with defeat and the argument is more specific than every argument for the opposite card in the opponent's dialogue.

- if the declarer uses one of declarer's burden shift cards on a card in an argument, then the onus of being more specific falls on defender at that point in the argument tree.

Specificity: Specificity adjudicates between arguments for opposite cards. It chooses the argument which meets the broader criterion of being more specific than another. An argument is deemed more specific ifthere is a set of cards that can be cited that activates the lesser argument, but does not activate the greater argument. If there is ordered evidence, then the set of cards is first augmented by any evidence cards less than the highest evidence card in the set (if any), before checking for activation. An argument for a card is activated by a set of cards if the set of cards contains a cut-set (not including the root) of the argument. If neither argument activates the other, the two arguments are said to be incomparable. If both activate the other, the two arguments are said to be equi-specific.

Argument Manipulation: A player may support a card in an argument by flipping cards on unexhausted cases during his turn. If an admissible card is flipped on a case and is not removed, it may be then added as support of that decision. A flipped card is burned, or removed from a case and placed back into the pool, if it is anti-evidence, it has already appeared in the case, or its opposite has already appeared in the case.

Admissible: A card is admissible (or live) if

- it is an evidence card; or
- it is a card for which there is no opened case for the opposite card and for which there are no current challenges; or
- a tree of potential argument for the card (henceforth, an argument) can be cited for the card which is undefeated for the player.

Resources: Each flip for the declarer consumes one resource card. Each flip for the defender contributes one card to the declarer's resources. Any resources consumed while flipping from the first sufficient response to a challenge are not transferred to the other side if the challenge is met by producing an admissible card among the facts of the first response.

III.b.2 Argument Game Discussion

The Argumentation Game, originally designed by R.P. Loui and William Chen, is based on the LMNOP program (Loui, Norman, Stiefvater, Merrill, Olson, Costello, 1992). LMNOP is based on the idea of a non-demonstrative or defeasible rule: a rule that admits exceptions. It adopts a representational convention that supposes there is an implicit preference of more specific rules over less specific rules. In fact, it automatically adjudicates between competing arguments when one argument meets the broader criterion of being more specific than another.

The argument game is an invaluable tool for the study of argumentation strategies and specificity. Players are forced to consider the question of where to spend limited resources. Logical methods of argumentation are reinforced and can shortly become second nature. Specificity, a difficult concept when applied to argumentation, is necessary to the understanding of the game and easily learned in the context of the game.

III.b.3 Program for Argumentation Game

To further demonstrate the power of games in the study of Computer Science, the game was coded using c. A listing of the program as well as some sample output are included as an appendix. During the process of coding, it was discovered just how little of this area is specified. The rules of specificity and argumentation protocol were largely designed anew. The very act of designing therefore became a useful tool for learning the concept itself.

The program is currently near completion. In the next year, it will be finished and a graphics driver will be placed on it as a front-end. Plans are being made to distribute or sell it to community High Schools and Jr. High Schools for use within debate clubs or similar activities.

III.c The Layout Game

III.c.1 Rules

Layout Game Rules

For quite obvious reasons, the layout of interconnecting devices and boards within architectures is a vital study within computer science. Speed and efficiency of a computer are greatly affected by the hardware placement. The Layout Game deals with solving a set of constraints while maximizing the efficiency of the layout.

Number of Players: Two players.

The Cards: Suit and color are irrelevant.

The Pack: Two packs are required and one given to each player. Cards are an abstraction for hardware devices and are laid out according to the rules detailed below.

The Object of Play: To satisfy the current constraints with the least amount of points, determined by the efficiency of the solution, while simulataneously giving the opponent difficult tasks to perform.

The Game Layout: There are two areas of play. In the constraint area, tasks are built which detail what cards must be inter-connected. Tasks consist of a leftmost card called the head. followed by an upside-down card and then a list of up to 5 connection cards. Each task asserts that each occurrence of the head must be connected to any occurrence of the connection cards. Four examples of tasks are shown below in Figure 3

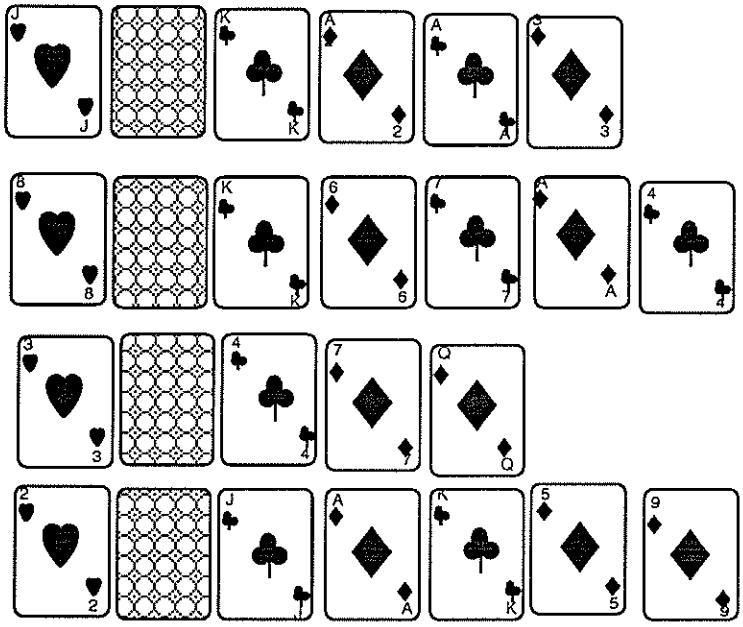

Figure 3: Task Area

The other area is the layout area. In this area are the cards (representing hardware devices) which must be connected according to the tasks. For example, with the tasks shown in Figure 3, any occurrence of 2 must be connected in some way to any J, A, K, 8, or 9. This connection may be accomplished via placement of these cards adjacent to the 2 in any of the 8 compass directions, or by using a bus. A bus is a connection of wires and can be used to connect any number of cards with connections to the bus to all of the others. In Figure 5 below, the layout satisfies the constraint that the 3 is connected to all of the J's, but none of the K's.

Therefore, if a player has to create a valid layout for the following cards: K, J, 8, A, 7, Q, 4, and 3, one valid solution is shown

in Figure 4. This solution layout satisfies all of the connection for the head cards J and 3. Note, however, that this is not necessarily the only optimal solution.

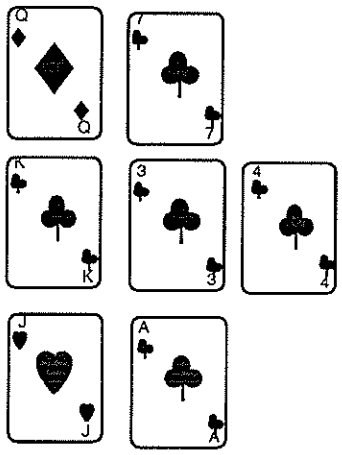

Figure 4: Example Layout

The Play: The two players alternate roles each turn. One player is the constraint builder, whose objective is to create difficult tasks for his opponent, the designer, as well as checking the design for constraint satisfaction. It is the job of the designer to meet these constraints as effectively as possible. One design is generally more "effective" than another if it has a smaller radius or uses fewer bus cards, which are represented by upside-down cards. Precise rules on judging the effectiveness of layouts and scoring is given below in The Scoring.

First, the constraint builder places 4 cards at desired places on the board. The location of the card is chosen before the card is revealed. Each of the cards may go in either the task area or in the

layout area. Cards placed in the task area may be added to an existing rule with fewer than 5 connection cards, thus creating more constraints the designer must follow, or cards may be placed that add tasks. If a new task is added whose head already exists, the new task is removed and the constraints on it must be placed on the other occurrence. Cards placed in the layout area give the designer more cards that must be fit within the layout.

Next, it is the designer's turn either to augment or to adapt the last layout, or to create a wholly new layout such that all of the constraints are met. When the designer is satisfied with the layout, the constraint builder checks to make sure all of the constraints are satisfied. If so, the score is computed, added to the score of the current designer, and the roles are reversed. When both players have had the same amount of turns as designer and one player has reached 100 points, the other player wins the game. If both players reach 100 within the same turn pair, the player with the lower score of the two loses. In case of a tie, the game becomes sudden death, with the first player to get a more effective solution than the other during the turn pair wins.

The Scoring: Scoring is rather complex. It is computed by the sum of the each connected component in the layout. The scores for each are computed by the following formula:

$$\sum_{i=length,width}[radius(component_i)] * \sum_{i} 2^{buslength(bus_i)}$$

For example, the consider the layout in Figure 5:

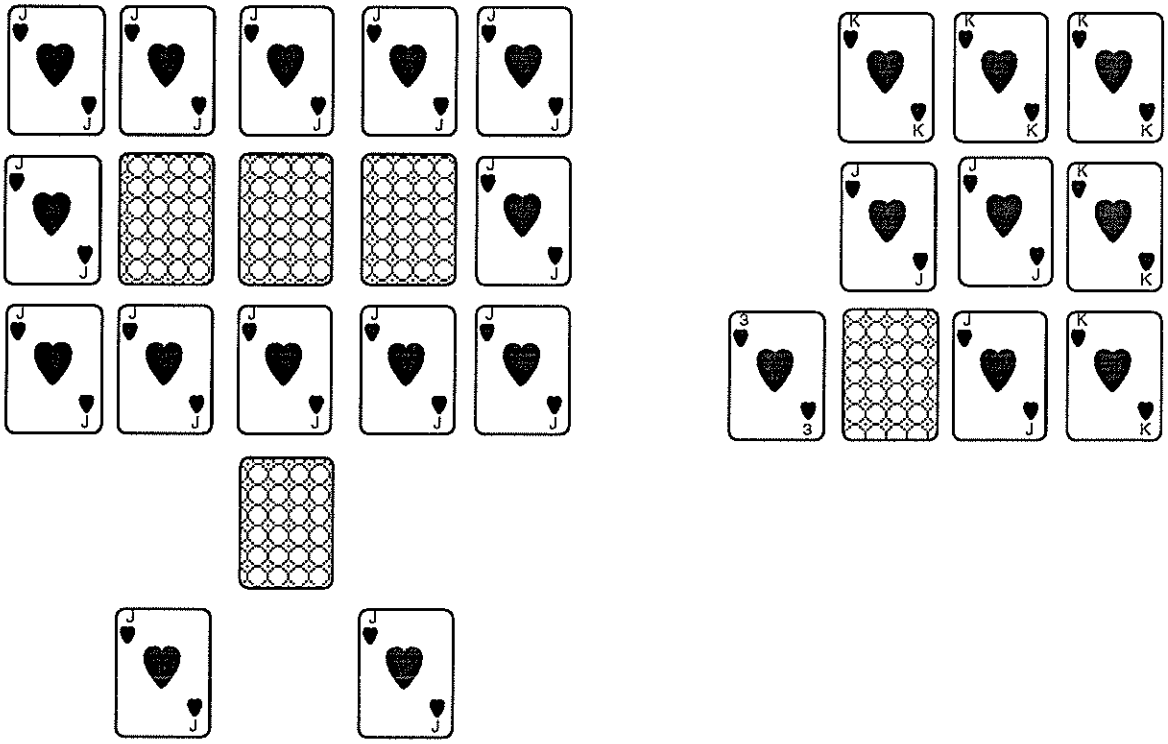

Figure 5: A Layout Example

The score for that would be:

$$[(5+5)*(2^3+2^1)]+[(4+3)*2^1]=10*10+7*2=114$$

Game Variations: One possibility is to make the game more parallel to real life hardware problems by penalizing for any lines that cross. In this version, instead of buses, the players would use specific wires to connect two cards. Also, the longer the lines, the less effective the solution. Some equation would be needed to compute the score of a layout, possibly computed by doing the sum of the length of the wires (in inches) and adding the number of times wires cross. The weighting of this would have to be worked out.

Another idea is that the cards in the task are prioritized according to the amount that the device it represents is used. With this alternative, the game could be extended to make a result less effective because of, for example, bus-overload.

III.c.2 Layout Game Discussion

The purpose of the layout game is to study the way in which computer designers are forced to solve a given set of layout rules while maximizing efficiency. The rules and constraint creation system accomplish this end through a simple yet enjoyable game. The game deals with both the expense and problems of bus use when compared to direct connections, as well as fitting the layout in as small a space as is possible. The addition of the prioritization by usage is also well suited for the study of real-world layout techniques. Further, the constraint satisfaction techniques contained within the layout game can be helpful in many other areas because constraint satisfaction is found everywhere and maximization of efficiency is a prevalent task in all of computer science.

IV. Conclusions - Using Games as Teachering Tools

When striving to create dynamic and exciting academic games, it is imperative to remember that there is a necessary distinction between simply enjoyable games and enjoyable academic games. The primary task of academic games is to teach and/or study a certain

topic or concept, not to replace television as entertainment for students.

With this in mind, it is crucial for the teacher and designer of the game to keep in mind, at all times, the target concept which the game is to teach. The game should be crafted so that the successful use of the concept during play should reward the player. For example, in the Layout Game, the scoring is devised so that just those things that are important in the computer science topic of layout design score points for the player. Such a focus on the topic will maximize the effectiveness and educational value of the game.

Additionally, the introduction and follow-up interaction between teacher and students is critical to a game's success. As stated above, the introduction gives background information to provide the student with a frame of reference and focus. The follow-up can serve as a reinforcement of the important concepts, as an overview of the topic after the students are familiar with its workings, and as a review of what was learned so that the teacher is more able to gauge the lesson's effectiveness. Without these steps, a game can teach, for example, people to have great proficiency in manipulating grammars without providing an understanding of anything with respect to computer science in general.

Game-play as a tool for learning is now widely accepted. However, there currently is too much emphasis on developing games for younger children. Adults and teenagers also need to have

positive feelings towards their studies and enjoying themselves while learning. While adults don't necessarily have to play to sit still, there are considerable benefits to gaming. Games such as the ones above can be an excellent tool in all areas of learning - including Computer Science.

Appendix - Argumentation Game Program

Here is a printed run of a game between two players using the Argumentation Game Program. Some of the invalid moves and checking of board and argument status have been removed for compactness. Although players deal with argument construction and specificity multiple times during this run, it is not an entire game.

```
Welcome to the Two-player version of An Argument Game.
        R. P. Loui, W. Chen, patent pending.


During each turn, the valid responses are:
    d[isplay]a[rguments] -> displays the current arguments.
    d[siplay]b[oard] -> displays the current board status.
    a[dd] -> adds a card to one of your arguments, parameters:
        argument number, card to add, card to add under.
    r[emove]a[rgument] -> removes an argument.
    r[estate] -> restate arguments - allows you to add or remove
        arguments to your current argument list.
    f[lip]r[esource] -> flips a resource card.
    f[lip]c[ase] -> flips a card under a case, parameters are:
        l/s (long or short case), case number.
    e[nd] -> ends turn.
    q[uit] -> ends game.
Pretty boring, eh?


Current status of the board is as follows:
EVIDENCE:  2 Di   5 Sp   7 Sp

SHORT CASES:
0: 10 Sp(2)          1: 9 Sp (2)
2: Q Cl (2)          3: 4 Sp (2)
4: 6 Sp (2)          5: K Sp (2)
6: A Cl (2)          7: 6 Cl (2)
8: 9 He (2)          9: A Sp (2)

LONG CASES:
10:  9 Di (10)       11:  8 He (10)
12:  6 He (10)       13:  K Sp (10)
14:  10 Di(10)       15:  8 Sp (10)
```

16: 10 Sp(10) 17: 9 Cl (10)
18: J Cl (10) 19: 9 Sp (10)

For ease of input, setting player 0 trying to argue for 9 Sp
with 20 resource cards.

Player 0, it is your turn.
Your current arguments:
Argument 0: 9 Cl

Your opponent's current arguments:

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 6 Cl leaving 9 facts left unturned in that case.
You flip the valid support card 6 Cl.
Attempting to add the card to the 1 occurance in your argument.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > d
You can:
(0) challenge an argument in your opponent's current dialogue,
(1) refute a card in your opponent's current dialogue,
(2) present a (new) dialogue, or
(3) go back to options?
> 2
Please input the argument(s) you would like to use (negative will
end inputing). > 0
> -1
You are using the following arguments:
0: 9 Cl (6 Cl)

Player 1, it is your turn.
Your current arguments:

Your opponent's current arguments:
Argument 0: 9 Cl (6 Cl)

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > db

Current status of the board is as follows:
EVIDENCE: 2 Di 5 Sp 7 Sp

SHORT CASES:
0: 10 Sp(2) 1: 9 Sp (2)
2: Q Cl (2) 3: 4 Sp (2)
4: 6 Sp (2) 5: K Sp (2)

6: A Cl (2) 7: 6 Cl (2)
8: 9 He (2) 9: A Sp (2)

LONG CASES:
10: 9 Di (10) 11: 8 He (10)
12: 6 He (10) 13: K Sp (10)
14: 10 Di(10) 15: 8 Sp (10)
16: 10 Sp(10) 17: 9 Cl (10)
18: J Cl (10) 19: 9 Sp, 6 Cl (9)

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > a
What card is being added (form: J Cl)? 9 Di

Your argument is now:
Argument 0: 9 Di

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 10
The card revealed is K Cl leaving 9 facts left unturned in that case.
You flip the valid support card K Cl.
Attempting to add the card to the 1 occurance in your argument.
Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > da

Your current arguments are:
Argument 0: 9 Di (K Cl)

Your opponent's current arguments are:
Argument 0: 9 Cl (6 Cl)

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > a
What argument would you like to add to? > 1
What card is being added (form: J Cl)? 6 He

Your argument is now:
Argument 0: 9 Di (K Cl)
Argument 1: 6 He

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 12
The card revealed is A He leaving 9 facts left unturned in that case.
A He is not alive. Removing it from the case.
Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 12
The card revealed is 2 Cl leaving 8 facts left unturned in that case.

```
2 Cl is Anti-Evidence.  Removing it from the case.
Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > fc
What case would you like to flip on, using the short cases as cases
      0 through 9 and the long ones as 10 through 19.
  > 12
The card revealed is 9 Sp leaving 7 facts left unturned in that case.
You flip the valid support card 9 Sp.
Attempting to add the card to the 1 occurance in your arguments.
Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > d
You can:
      (0) challenge an argument in your opponent's current dialogue,
      (1) refute a card in your opponent's current dialogue,
      (2) present a (new) dialogue, or
      (3) go back to options?
  > 1
What card in your opponent's argument are you trying to refute?  > 6 Cl
And what argument of yours are you using to refute?  > 1

Your response has been determined to be sufficient, as it refutes a card
in your opponent's dialogue arguments with a more specific argument.

Player 0, it is your turn.
Your current arguments:
Argument 0:   9 Cl (6 Cl)

Your opponent's current arguments:
Argument 0:   9 Di (K Cl)
Argument 1:   6 He (9 Sp)


Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > d
You can:
      (0) challenge an argument in your opponent's current dialogue,
      (1) refute a card in your opponent's current dialogue,
      (2) present a (new) dialogue, or
      (3) go back to options?
  > 1
What card in your opponent's argument are you trying to refute?  > 6 He
And what argument of yours are you using to refute?  > 0
You must refute your opponent's card, 6 He with an argument starting with
   an opposite card.
You formed no valid refutation.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > db


Current status of the board is as follows:
EVIDENCE:  2 Di   5 Sp   7 Sp

SHORT CASES:
0: 10 Sp(2)          1: 9 Sp (2)
2: Q Cl (2)          3: 4 Sp (2)
```

4: 6 Sp (2) 5: K Sp (2)
6: A Cl (2) 7: 6 Cl (2)
8: 9 He (2) 9: A Sp (2)

LONG CASES:
10: 9 Di, K Cl (9) 11: 8 He (10)
12: 6 He, 9 Sp (7) 13: K Sp (10)
14: 10 Di(10) 15: 8 Sp (10)
16: 10 Sp(10) 17: 9 Cl (10)
18: J Cl (10) 19: 9 Sp, 6 Cl (9)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > a
What argument would you like to add to? > 1
What card is being added (form: J Cl)? 6 Cl

Your argument is now:
Argument 0: 9 Cl (6 Cl)
Argument 1: 6 Cl

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 7
The card revealed is 8 Cl leaving 1 facts left unturned in that case.
You flip the valid support card 8 Cl.
Attempting to add the card to the 2 occurances in your arguments.

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > da

Your current arguments are:
Argument 0: 9 Cl (6 Cl [8 Cl])
Argument 1: 6 Cl (8 Cl)

Your opponent's current arguments are:
Argument 0: 9 Di (K Cl)
Argument 1: 6 He (9 Sp)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > d
You can:
(0) challenge an argument in your opponent's current dialogue,
(1) refute a card in your opponent's current dialogue,
(2) present a (new) dialogue, or
(3) go back to options?
> 1
What card in your opponent's argument are you trying to refute? > 6 He
And what argument of yours are you using to refute? > 1
Neither is More Specific.
Your refutation is not sufficient, as it was not more specific than
your opponent's.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc

What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 2 Cl leaving 8 facts left unturned in that case.
2 Cl is Anti-Evidence. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is J He leaving 7 facts left unturned in that case.
J He is not alive. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 9 Di leaving 6 facts left unturned in that case.
9 Di is contrary to case cards. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 7 He leaving 5 facts left unturned in that case.
7 He is Anti-Evidence. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 7 Di leaving 4 facts left unturned in that case.
7 Di is Anti-Evidence. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 9 Cl leaving 3 facts left unturned in that case.
9 Cl is already in the case. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is 5 Cl leaving 2 facts left unturned in that case.
5 Cl is not alive. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19
The card revealed is J He leaving 1 facts left unturned in that case.
J He is not alive. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 19

The card revealed is 8 Sp leaving 0 facts left unturned in that case.
You flip the valid support card 8 Sp.
Attempting to add the card to the 2 occurances in your arguments.
That may sufficiently deal with the challenge on 9 Cl.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > db

Current status of the board is as follows:
EVIDENCE: 2 Di 5 Sp 7 Sp

SHORT CASES:
0: 10 Sp(2) 1: 9 Sp (2)
2: Q Cl (2) 3: 4 Sp (2)
4: 6 Sp (2) 5: K Sp (2)
6: A Cl (2) 7: 6 Cl,8 Cl (1)
8: 9 He (2) 9: A Sp (2)

LONG CASES:
10: 9 Di, K Cl (9) 11: 8 He (10)
12: 6 He, 9 Sp (7) 13: K Sp (10)
14: 10 Di(10) 15: 8 Sp (10)
16: 10 Sp(10) 17: 9 Cl (10)
18: J Cl (10) 19: 9 Sp, 6 Cl, 8 Sp (0)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > da

Your current arguments are:
Argument 0: 9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1: 6 Cl (8 Cl)

Your opponent's current arguments are:
Argument 0: 9 Di (K Cl)
Argument 1: 6 He (9 Sp [8 Sp])

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > d
You can:
(0) challenge an argument in your opponent's current dialogue,
(1) refute a card in your opponent's current dialogue,
(2) present a (new) dialogue, or
(3) go back to options?
> 0
What card in your opponent's argument are you trying to challenge? > 8 Sp
8 Sp in argument number 1 is now under challenge.
Your response has been determined to be sufficient, as it challenges an unsupported card in your opponent's dialogue arguments.

Player 1, it is your turn.
Your current arguments:
Argument 0: 9 Di (K Cl)
Argument 1: 6 He (9 Sp [8 Sp])

```
Your opponent's current arguments:
Argument 0:   9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1:   6 Cl (8 Cl)


Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > r

Your current arguments:
Argument 0:   9 Di (K Cl)
Argument 1:   6 He (9 Sp [8 Sp])

First, lets delete the cards you dont' want to use.
Name cards to delete - if a card with spport is named,all of its
  support will be deleted as well.
Inputing x x on a line will finish deleting.

What card is being deleted (form: J Cl)?   > 8 Sp
8 Sp removed and replaced by the last support, 8 Sp.

Your current arguments:
Argument 0:   9 Di (K Cl)
Argument 1:   6 He (9 Sp)

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > d
You can:
      (0) challenge an argument in your opponent's current dialogue,
      (1) refute a card in your opponent's current dialogue,
      (2) present a (new) dialogue, or
      (3) go back to options?
  > 2
Please input the argument(s) you would like to use (negative will
end inputing).  > 0
  > -1

You are using the following arguments:
0:  9 Di (K Cl)

Challenged card has been removed from argument.
Your dialogue arguments create a valid response by successfully
dealing with your opponent's last challenge.

Player 0, it is your turn.
Your current arguments:
Argument 0:   9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1:   6 Cl (8 Cl)

Your opponent's current arguments:
Argument 0:   9 Di (K Cl)
Argument 1:   6 He (9 Sp)
```

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > db

Current status of the board is as follows:
EVIDENCE: 2 Di 5 Sp 7 Sp

SHORT CASES:
0: 10 Sp(2) 1: 9 Sp (2)
2: Q Cl (2) 3: 4 Sp (2)
4: 6 Sp (2) 5: K Sp (2)
6: A Cl (2) 7: 6 Cl,8 Cl (1)
8: 9 He (2) 9: A Sp (2)

LONG CASES:
10: 9 Di, K Cl (9) 11: 8 He (10)
12: 6 He, 9 Sp (7) 13: K Sp (10)
14: 10 Di(10) 15: 8 Sp (10)
16: 10 Sp(10) 17: 9 Cl (10)
18: J Cl (10) 19: 9 Sp, 6 Cl, 8 Sp (0)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > d
You can:
(0) challenge an argument in your opponent's current dialogue,
(1) refute a card in your opponent's current dialogue,
(2) present a (new) dialogue, or
(3) go back to options?
> 0
What card in your opponent's argument are you trying to challenge? > K Cl
K Cl in argument number 0 is now under challenge.
Your response has been determined to be sufficient, as it challenges an unsupported card in your opponent's dialogue arguments.

Player 1, it is your turn.
Your current arguments:
Argument 0: 9 Di (K Cl)
Argument 1: 6 He (9 Sp)

Your opponent's current arguments:
Argument 0: 9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1: 6 Cl (8 Cl)

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > a
What argument would you like to add to? > 2
What card is being added (form: J Cl)? K Sp

Your argument is now:
Argument 0: 9 Di (K Cl)
Argument 1: 6 He (9 Sp)

```
Argument 2:  K Sp

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > fc
What case would you like to flip on, using the short cases as cases
      0 through 9 and the long ones as 10 through 19.
  > 13
The card revealed is 8 Sp leaving 9 facts left unturned in that case.
You flip the valid support card 8 Sp.
Attempting to add the card to the 2 occurances in your arguments.
That may sufficiently deal with the challenge on K Cl.
Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > da

Your current arguments are:
Argument 0:  9 Di (K Cl [8 Sp])
Argument 1:  6 He (9 Sp)
Argument 2:  K Sp (8 Sp)

Your opponent's current arguments are:
Argument 0:  9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1:  6 Cl (8 Cl)

Player 1's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > d
You can:
      (0) challenge an argument in your opponent's current dialogue,
      (1) refute a card in your opponent's current dialogue,
      (2) present a (new) dialogue, or
      (3) go back to options?
  > 2
Please input the argument(s) you would like to use (negative will
end inputing).  > 0
  > -1
You are using the following arguments:
0:  9 Di (K Cl [8 Sp])

Challenged card supported.
Your dialogue arguments create a valid response by successfully
dealing with your opponent's last challenge.

Player 0, it is your turn.
Your current arguments:
Argument 0:  9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1:  6 Cl (8 Cl)

Your opponent's current arguments:
Argument 0:  9 Di (K Cl [8 Sp])
Argument 1:  6 He (9 Sp)
Argument 2:  K Sp (8 Sp)


Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > db
```

Current status of the board is as follows:
EVIDENCE: 2 Di 5 Sp 7 Sp

SHORT CASES:
0: 10 Sp(2) 1: 9 Sp (2)
2: Q Cl (2) 3: 4 Sp (2)
4: 6 Sp (2) 5: K Sp (2)
6: A Cl (2) 7: 6 Cl,8 Cl (1)
8: 9 He (2) 9: A Sp (2)

LONG CASES:
10: 9 Di, K Cl (9) 11: 8 He (10)
12: 6 He, 9 Sp (7) 13: K Sp, 8 Sp (9)
14: 10 Di(10) 15: 8 Sp (10)
16: 10 Sp(10) 17: 9 Cl (10)
18: J Cl (10) 19: 9 Sp, 6 Cl, 8 Sp (0)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > a
What card is being added (form: J Cl)? 9 Cl

Your argument is now:
Argument 0: 9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1: 6 Cl (8 Cl)
Argument 2: 9 Cl

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 17
The card revealed is K He leaving 9 facts left unturned in that case.
K He is not alive. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 17
The card revealed is K Cl leaving 8 facts left unturned in that case.
You flip the valid support card K Cl.
Attempting to add the card to the 2 occurances in your arguments.
ILLEGAL: No case with the proper facts was found.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > fc
What case would you like to flip on, using the short cases as cases
0 through 9 and the long ones as 10 through 19.
> 17
The card revealed is 5 Di leaving 7 facts left unturned in that case.
5 Di is Anti-Evidence. Removing it from the case.
Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > db

Current status of the board is as follows:

EVIDENCE: 2 Di 5 Sp 7 Sp

SHORT CASES:
0: 10 Sp(2) 1: 9 Sp (2)
2: Q Cl (2) 3: 4 Sp (2)
4: 6 Sp (2) 5: K Sp (2)
6: A Cl (2) 7: 6 Cl,8 Cl (1)
8: 9 He (2) 9: A Sp (2)

LONG CASES:
10: 9 Di, K Cl (9) 11: 8 He (10)
12: 6 He, 9 Sp (7) 13: K Sp, 8 Sp (9)
14: 10 Di(10) 15: 8 Sp (10)
16: 10 Sp(10) 17: 9 Cl, K Cl (7)
18: J Cl (10) 19: 9 Sp, 6 Cl, 8 Sp (0)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > da

Your current arguments are:
Argument 0: 9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1: 6 Cl (8 Cl)
Argument 2: 9 Cl (K Cl)

Your opponent's current arguments are:
Argument 0: 9 Di (K Cl [8 Sp])
Argument 1: 6 He (9 Sp)
Argument 2: K Sp (8 Sp)

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?) > r

Your current arguments:
Argument 0: 9 Cl (6 Cl [8 Cl], 8 Sp)
Argument 1: 6 Cl (8 Cl)
Argument 2: 9 Cl (K Cl [8 Sp])

First, lets delete the cards you dont' want to use.
Name cards to delete - if a card with spport is named,all of its
support will be deleted as well.
Inputing x x on a line will finish deleting.

What card is being deleted (form: J Cl)? > 8 Sp
More than one instance of 8 Sp occurrs in the argument.
Which of the following instances did you want?
Instance 0:
Argument 0: 9 Cl(6 Cl(8 Cl), *8 Sp*)
Argument 1: 6 Cl(8 Cl)
Argument 2: 9 Cl(K Cl(8 Sp))

Instance 1:
Argument 0: 9 Cl(6 Cl(8 Cl), 8 Sp)

```
Argument 1:   6 Cl(8 Cl)
Argument 2:   9 Cl(K Cl(*8 Sp*))

  > 0
8 Sp removed and replaced by the last support, 8 Sp.

Player 0's turn (da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > d
You can:
      (0) challenge an argument in your opponent's current dialogue,
      (1) refute a card in your opponent's current dialogue,
      (2) present a (new) dialogue, or
      (3) go back to options?
  > 2
Please input the argument(s) you would like to use (negative will
end inputing).  > 2
  > -1
You are using the following arguments:
0:  9 Cl (K Cl [8 Sp])

Your dialogue arguments create a valid response by its being a
more effective argument than your opponent's.
```

End of example

As you can see, the players both demonstrated knowledge of how the arguments are manipulated by revealing evidence and adapting arguments. Also, and more importantly, they came into contact with and eventually mastered specificity and its importance in refutation and new dialogue presentation.

The c code for the Argumentation follows.

```
From bmwl@cecl.wustl.edu Fri May  7 13:22:34 1993
Received: from cecl.wustl.edu by ai.wustl.edu with SMTP
        (5.65a/1.35); id AA01956; Fri, 7 May 93 13:22:26 -0500
Return-Path: <bmwl@cecl.wustl.edu>
Received: by cecl.wustl.edu
        (5.65a/1.35); id AA14560; Fri, 7 May 93 13:22:23 -0500
Message-Id: <9305071822.AA14560@cecl.wustl.edu>
From: bmwl@cecl.wustl.edu (Ben Weber)
Date: Fri, 7 May 1993 13:22:23 -0500
X-Mailer: Mail User's Shell (7.2.4 2/2/92)
To: loui@ai.wustl.edu
Status: R

/* ********************************************************** *
 *                                                            *
 *                       Benjamin M. Weber                    *
 *                    An Argumentation Game                   *
 *                                                            *
 *                 based on game by Ronald P. Loui            *
 *                                                            *
 *                          11/10/92                          *
 *                                                            *
 * ********************************************************** */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <sys/time.h>

#define MAXCARDS 208
#define MAXNODES 10
#define MAXHEADS 30
/* more than 10 supporting cards should never happen realistically */

#define wordsize 30
typedef char word[wordsize];
typedef int boolean;

#define AUTOADD 1  /* automatically adds supports when flip? */
#define DEBUG 0
#define TEMPDEBUG 1
/* temp debug tests will be removed when perfected */

/* ********************   CARD ADT definitions ***************** */

typedef enum { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE,
               TEN, JACK, QUEEN, KING, ACE, JOKER } RANKTYPE;

char * RANKS [ ] = { "2","3","4","5","6","7","8","9","10",
                     "J","Q","K","A","Joker" };

typedef enum { CLUBS, HEARTS, DIAMONDS, SPADES } SUITTYPE;

typedef enum { RED, BLACK } COLORTYPE;

char * SUITS [ ] = { "Cl", "He", "Di", "Sp" };

char * COLORS { ] = { "Red","Black" };


typedef struct {
                 RANKTYPE Rank;
                 SUITTYPE Suit;
                 COLORTYPE Color;
               } CARD;
```

```
/* ********************   end CARD definitions ***************** */

/* ********************   ADT CASES definitions ***************** */

typedef struct
   {
        CARD FactEvid [11];  /* each case has a number of cards *
                              * the 0th card being the decision */
        int FactsShown; /* the number of facts shown already */
        int CardsLeft;  /* and keeps track of the number of *
                         * facts left                       */
   }  CASES;


typedef struct
   {
        CARD Evidence [3];   /* the 3 evidence cards */
        CASES Cases [20];    /* the 20 short cases           *
                              * 0-9 are short, 10-19 are long */
   } BOARD;


/* ********************   end CASES definitions ***************** */

BOARD GameBoard;  /* the board with evidence, 10 short, and 10 long */

/* ********************   ADT POOL definitions ***************** */

typedef struct
   {
        int CardsLeft;  /* and keeps track of the number in pool */
        CARD PoolCards [MAXCARDS];  /* the pool has a number of cards */
   }  POOLTYPE;

/* ********************   end POOL definitions ***************** */

POOLTYPE Pool;  /* the pool of cards */

/* ******************   ADT RESOURCE definitions   ************** */

typedef struct
   {
        int ResNum;  /* the number of resource cards */
        CARD Resources [30];
   }  RESTYPE;

/* ******************   end RESOURCE definitions   ************** */

RESTYPE Res;

/* ******************   ADT ARGUMENT definitions   ************** */

typedef struct nodestruct
   {
        CARD C;              /* the card at that node */
        int numsupports;     /* how many supporting cards the node has */
        boolean challnode;   /* does this node have an unmet challenge *
                              * or refutation? */
        struct nodestruct *support [MAXNODES];
                             /* each supporting card is a nodestruct */
        struct nodestruct *parent; /* the parent node */
        int argnumber;       /* which argument are you in? */
        int depth;           /* how deep in the arg are you, supt wise */
   }  NODE;


typedef struct
   {
        NODE * HeadCards [MAXHEADS];  /* pointer to the head cards */
        int numheads;        /* number of head cards used */
        boolean burden;      /* does that player have burden? */
        NODE * Challenged [MAXHEADS];
        /* are any of this argument's cards under challenge OR refutation?
   } ARGUMENT;


/* ******************   end ARGUMENT definitions   ************** */

ARGUMENT Arg [1];

boolean Ordered;  /* is the evidence ordered?? */


/* ******************   ADT DIALOGUE definitions   ************** */


typedef struct currnodestruct
   {
      int type;        /* 0 = challenge, 1 = refute, 2 = present dialogue */
      NODE * object;  /* node of card being challenged/refuted */
      NODE * Arg[10]; /* the current argument(s) of this player */
      int numargs;
   }  DIALOGUE;


/* ******************   end DIALOGUE definitions   ************** */

DIALOGUE Dial [1];

/* ******************** RANDOM Number Generator ****************** */


unsigned long int next;  /* for random number */


void srand ()
{  /* sets seed for rand() */
struct timeval tp;  /* for the time */
unsigned long int i;
   gettimeofday(& tp,0); /* gets a pointer to the time */
   next = tp.tv_sec;     /* uses the time in seconds to do get next */
   for (i = (next % 500); i > 1; i--)
      Random(i);
}


int randm ()
{  /* returns pseudo-random integer on 0..32767 */
   next = next * 1103515245 + 12345;
   return ((unsigned int) (next / 65536) % 32768); /* 0..32767 */
}
```

```
int Random (c)
int c;
{ /* Returns a randomly selected number between 1 and c */
   return (randm()%c);
}


/* ****************** end RANDOM Number Generator ***************** */


/* ********************  MISCELLANEOUS Functions ***************** */


/* ******************  end MISCELLANEOUS Functions ************** */


/* ********************  CARD ADT functions  ******************* */


RANKTYPE GetRank (C)
CARD C;
{ /* returns the rank of a given card */
   return (C.Rank);
}


SUITTYPE GetSuit (C)
CARD C;
{ /* returns the suit of a given card */
   return (C.Suit);
}


COLORTYPE GetColor (C)
CARD C;
{ /* returns the color of a given card */
   return (C.Color);
}


CARD GetCard (i,Carray)
int i;
CARD Carray [];
{  /* returns the card at position i in Card Array Carray */
   return (Carray [i]);
}

void StoreCard (i,C,Carray)
int i;
CARD C;
CARD Carray[];
{ /* destructively stores card C in position i in Card Array Carray */
   Carray[i] = C;
}


void PrintCard (C)
CARD C;
{ /* writes out a description of one card */
   printf (RANKS[((int)GetRank(C))]);
   printf (" ");
   printf (SUITS[((int)GetSuit(C))]);
}
```

```
void PrintCardCol (C)
CARD C;
{ /* writes out a description of one card */
   printf (COLORS[((int)GetColor(C))]);
   printf (" ");
   printf (RANKS[((int)GetRank(C))]);
}


void HiLitPrintCard (C)
CARD C;
{ /* writes out a card with *  * around it... */
   printf ("*");
   printf (RANKS[((int)GetRank(C))]);
   printf (" ");
   printf (SUITS[((int)GetSuit(C))]);
   printf("*");
}



boolean SameCard (C1,C2)
CARD C1;
CARD C2;
{ /* returns whether the given cards have the same rank and suit */
  return ((C1.Rank == C2.Rank) && (C1.Suit == C2.Suit));
}


boolean EquivCard (C1,C2)
CARD C1;
CARD C2;
{ /* returns whether the given cards have the same rank and color */
  return ((C1.Rank == C2.Rank) && (C1.Color == C2.Color));
}


boolean OppCard (C1,C2)
CARD C1;
CARD C2;
{ /* returns whether the given cards have the same rank and opp. color */
  return ((C1.Rank == C2.Rank) && (C1.Color != C2.Color));
}


CARD MakeCard (r,s)
RANKTYPE r;
SUITTYPE s;
{ /* constructs a card with rank r and suit s and color depending *
   * on the suit                                                  */
CARD C;
   C.Rank = r;
   C.Suit = s;
   if ((s == DIAMONDS) || (s == HEARTS)) C.Color = RED;
   else C.Color = BLACK;
   return (C);
}


CARD MakeTheCard (r,s)
char * r;
char * s;
{ /* constructs a card with rank r and suit s, r and s are strings *
   * color is determined here and made - depends on suit obviously */
RANKTYPE Rk;
SUITTYPE Su;
COLORTYPE Co;
CARD C;
boolean error;
   error = 0;
   switch (r[0])
     {
        case '2': Rk = TWO; break;
        case '3': Rk = THREE; break;
        case '4': Rk = FOUR; break;
        case '5': Rk = FIVE; break;
        case '6': Rk = SIX; break;
        case '7': Rk = SEVEN; break;
        case '8': Rk = EIGHT; break;
        case '9': Rk = NINE; break;
        case '1':
                  if (r[1] == '0')  Rk = TEN;
                  else error += 1; break;
        case 'J': Rk = JACK; break;
        case 'Q': Rk = QUEEN; break;
        case 'K': Rk = KING; break;
        case 'A': Rk = ACE; break;
        case 'x': Rk = JOKER; break; /* for dummy card */
        default : error += 1;
     }
   switch (s[0])
     {
        case 'C' : Su = CLUBS; Co = BLACK; break;
        case 'H' : Su = HEARTS; Co = RED; break;
        case 'D' : Su = DIAMONDS; Co = RED; break;
        case 'S' : Su = SPADES; Co = BLACK; break;
        case 'x'  : Su = SPADES; Co = BLACK; break;  /* for dummy card */
        default : error += 2;
     }
   C.Color = Co;
   C.Suit = Su;
   C.Rank = Rk;
   if (error)
      {
        printf("Error making card, invalid");
        if (error != 2)  printf (" rank");
        if (error == 3) printf (" and");
        if (error >= 2) printf (" suit");
        printf(".\n");
        C.Suit = SPADES;
        C.Rank = JOKER;
      }
   if (DEBUG)
     {
        printf ("Made: ");
        PrintCard (C);
        printf ("\n");
     }
   return (C);
}


/* ********************  end CARD functions   ****************** */

CARD Ten;

/* ********************  POOL ADT functions   ****************** */

void NewPool ()
{ /* sets up the initial configuration of the Pool */
int i,dks;
RANKTYPE r;
SUITTYPE s;
   i = 0;
   for (dks = 0; dks <= 3; dks++)
   for (s=SPADES; (int)s >= (int)CLUBS; ((int)s)--)
          for (r=ACE; (int)r >= (int)TWO;((int)r)--)
            {
               Pool.PoolCards[i] = MakeCard(r,s);
               i++;
            }
   Pool.CardsLeft = --i;
   Ten.Rank = TEN;  /* to test to see if 10 for printing */
}

void Swap (cr,c)
int cr,c;
{ /* swaps two cards in the pool of cards - used to shuffle it */
CARD tmp;
   tmp = Pool.PoolCards [cr];
   Pool.PoolCards[cr] = Pool.PoolCards[c];
   Pool.PoolCards[c] = tmp;
}


void Shuffle ()
{ /* shuffles the pool of cards */
  /* This shuffle algorithm swithces the last card in the deck with
   *    the random card.  Then it does the same with the next to last,
   *    and so on. */
int cursor,rnd,tmp;
   tmp = Pool.CardsLeft;
   for (cursor = tmp; cursor >= 1; cursor--)
      {
        rnd = Random(cursor);
        Swap (Random(cursor),cursor);
      }
}



CARD GetPool ()
{  /* gets last from pool, decrements cards left */
   return Pool.PoolCards[Pool.CardsLeft--];
}


void ReturnPool (C)
CARD C;
{  /* returns a card to the pool - puts it somewhere random */
   Pool.PoolCards[++Pool.CardsLeft] = C;
   Swap (Pool.CardsLeft,Random(Pool.CardsLeft));
}


void PrintPool ()
{  /* prints out the current pool */
int count;
   printf("The %d cards in the pool are:\n",(Pool.CardsLeft+1));
   for (count = 0; count <= Pool.CardsLeft; count ++)
     {
        PrintCard(Pool.PoolCards[count]);
        if (GetRank(Ten) == GetRank(Pool.PoolCards[count]))
           printf ("  ");
        else
           printf ("   ");
        if ((count % 11)==10)
           printf("\n");
     }
   printf ("\n\n");
}


/* ********************  end POOL functions   ****************** */


/* ******************  RESOURCE ADT functions  *************** */


void InitResources (numres)
int numres;
{ /* sets up the resources with numres */
int count;
   for (count = 0; count < numres; count++)
      Res.Resources[count] = GetPool();
   Res.ResNum = numres;
}


void AddResource ()
{  /* adds a resource */
   Res.Resources[Res.ResNum] = GetPool();
   Res.ResNum++;
}


CARD GetResource ()
{  /* removes and returns a card from the resources */
  Res.ResNum--;
  return Res.Resources[Res.ResNum];
}


/* ******************  end RESOURCE functions  *************** */


/* ******************  ADT CASE functions   ****************** */


CARD RevealCase (casenum)
int casenum;
{  /* gets a card from the case and reveals it... */
int tempint;
   tempint = ++GameBoard.Cases[casenum].FactsShown;
   return (GameBoard.Cases[casenum].FactEvid[tempint]);
}


int FactsLeft (casenum)
int casenum;
{  /* returns the number of cards left in the case           *
    * if none left, can test for FactsLeft cause will be a 0 */
   return (GameBoard.Cases[casenum].CardsLeft -
                GameBoard.Cases[casenum].FactsShown);
}


int RemoveCase (casenum)
int casenum;
{  /* removes the last revealed card from the case, returns number  *
    * of cards left in that case (removes by replacing it with last *
```

```
     * card unrevealed (if any))                                    */
    if (GameBoard.Cases[casenum].CardsLeft -
                    GameBoard.Cases[casenum].FactsShown)
GameBoard.Cases[casenum].FactEvid[GameBoard.Cases[casenum].FactsShown] =
      GameBoard.Cases[casenum].FactEvid[GameBoard.Cases[casenum].CardsLeft];
    GameBoard.Cases[casenum].FactsShown--;
    GameBoard.Cases[casenum].CardsLeft--;
}


PrintCase (Cse)
CASES Cse;
{ /* prints out the case passed to it */
  /* shows only those cards revealed  */
CARD TempCard;
int i;
   for (i=0; i < Cse.FactsShown; i++)
     {
        PrintCard (GetCard(i,Cse));
        if (i != (Cse.FactsShown-1)) printf (",");
        if (i==7) printf ("\n\t\t\t\t");
     }
   printf(" (%d)",Cse.CardsLeft-Cse.FactsShown);
}


boolean AFact (Cd,num)
CARD Cd;
int num;
{  /* returns whether Cd is a fact of case number num */
int count;
   for (count = 1; count<=GameBoard.Cases[num].FactsShown; count++)
      if (EquivCard(Cd,GameBoard.Cases[num].FactEvid[count]))
        return 1;
   if (DEBUG)
     {
        PrintCard(Cd);
        printf(" was not found in case number %d.\n",num);
     }
   return 0;
}


boolean InDecisions (Cd,warn)
CARD Cd;
int warn;
{  /* returns whethere there is a case with decision Cd */
   /* If warn is set, then the function will only output a warning *
    * and return 1, letting the player do it manually (testing?)   */
int casenum;
boolean found;
  found = 0;
  for (casenum=0; casenum < 20; casenum++)
     if (EquivCard(Cd,GameBoard.Cases[casenum].FactEvid[0]))
        found = 1;
  if ((warn) && (!found))
     {
        printf("WARNING: No case with ");
        PrintCard(Cd);
        printf(" as the decision was found.\n");
        return 1;
     }
  if (!found)
     {
        printf("ILLEGAL: No case with the proper facts was found.\n");
        return 0;
     }
  return 1;
}


boolean InCases (Dec,Fact,warn)
NODE * Dec;
CARD Fact;
boolean warn;
{  /* reurns whether there is a case where all of the support of *
    * dec is in some case and Fact is in that case as well       */
   /* If warn is set, then the function will only output a warning *
    * and return 1, letting the player do it manually (testing?)   */
int casenum,supnum;
boolean notfound;
   for (casenum=0; casenum < 20; casenum++)
     {
        if (EquivCard(Dec->C,GameBoard.Cases[casenum].FactEvid[0]))
          {
             notfound = 0;
             supnum = 0;
             while ((supnum < Dec->numsupports) && (!notfound))
                {
                   if (!AFact(Dec->support[supnum],casenum))
                      notfound = 1;
                   supnum++;
                }
             if (!AFact(Fact,casenum)) notfound = 1;
             if (!notfound) casenum = 20;
          }
     }
  if ((warn) && (notfound))
    {
       printf("WARNING: No case with ");
       PrintCard(Fact);
       printf(" and all the other support of ");
       PrintCard(Dec->C);
       printf(" was found.\n");
       return 1;
    }
  if (notfound)
    {
       printf("ILLEGAL: No case with the proper facts was found.\n");
       return 0;
    }
  return 1;
}


boolean AntiEvid (Cd)
CARD Cd;
{  /* Checks to see if Cd is anti-Evid */
int count;
   for (count = 0; count <= 2; count++)
      if (OppCard(Cd,GameBoard.Evidence[count]))
        {
           PrintCard (Cd);
           printf(" is Anti-Evidence.  Removing it from the case.\n");
           return 1;
        }
  if (DEBUG)
    {
       PrintCard(Cd);
       printf(" is not Anti-Evidence.\n");
     }
   return 0;
}


boolean ContraCase (casenum, Cd)
int casenum;
CARD Cd;
{  /* Checks to see if Cd is a contradiction of a card revealed in *
    * case casenum                                                 */
int count;
   for (count=0; count<GameBoard.Cases[casenum].FactsShown; count++)
      if (OppCard(Cd,GameBoard.Cases[casenum].FactEvid[count]))
        {
           PrintCard (Cd);
           printf(" is contrary to case cards.  Removing it from the case.\n");
           return 1;
        }
   if (DEBUG)
     {
        PrintCard(Cd);
        printf(" is not contrary to a card in case %d.\n",casenum);
     }
   return 0;
}


boolean NotAlive (Cd)
CARD Cd;
{  /* Checks to see if Cd is not alive (not occurs in cases) */
int count;
   for (count = 0; count < 20; count++)
      if (EquivCard(Cd,GameBoard.Cases[count].FactEvid[0]))
        return 0;
   PrintCard(Cd);
   printf(" is not alive.  Removing it from the case.\n");
   return 1;
}


boolean CopyCaseCard (casenum, Cd)
int casenum;
CARD Cd;
{  /* Checks to see if Cd is a copy of another card revealed in *
    * case casenum (equivalent cards would be copy)             */
int count;
   for (count=0; count<GameBoard.Cases[casenum].FactsShown; count++)
      if (EquivCard(Cd,GameBoard.Cases[casenum].FactEvid[count]))
        {
           PrintCard (Cd);
           printf(" is already in the case.  Removing it from the case.\n");
           return 1;
        }
   if (DEBUG)
     {
        PrintCard(Cd);
        printf(" is not a copy of a card in case %d.\n",casenum);
     }
   return 0;
}


boolean CheckFlipped (casenum, Cd)
int casenum;
CARD Cd;
{  /* Checks to see if a flipped card should be thrown away (is *
    * anti-evidence, contradiction of another card in that case, *
    * or a copy of another card in that case)                    */
    if (CopyCaseCard(casenum,Cd) || ContraCase(casenum,Cd)
          || AntiEvid(Cd) || NotAlive(Cd))
      {
         RemoveCase(casenum);
         return 0;
      }
    if (TEMPDEBUG)
      {
         printf("You flip the valid support card ");
         PrintCard(Cd);
         printf(".\n");
      }
    return 1;
}


void InvisFlip (num)
{ /* Flips and returns a card on case num */
CARD revealed;
  if (FactsLeft(num))
     revealed = RevealCase(num);
  else printf("There are no cards left unturned in case %d.\n",num);
  CheckFlipped(num,revealed);
}


/* ********************  end CASE functions   ******************* */

/* ********************  ADT GAME functions   ******************* */


boolean CheckEvid (C)
CARD C;
{  /* checks to see if card C is evidence or anti-evidence - returns *
    * 1 if evidence, 2 if anti-evidence */
   /* the evid/anti-evid part will only work if Red/Black, not suits */
int count;
      for (count = 0;count <=2; count++) {
      if (C.Rank == GameBoard.Evidence[count].Rank) {
         if (C.Suit == GameBoard.Evidence[count].Suit) {
            if (DEBUG) printf ("Evidence card drawn - returning to pool.\n");
            return 1;   /* evidence */
            }
         else  {
            if (DEBUG) printf ("Anti-Evidence card drawn - returning to pool.\n"
            return 2; /* anti-evid */
            }
         }
      }
   return 0;
}


void PrintBoard()
{  /* prints out the game board as it currently is... */
int count,numfacts,numshown;
int ten;
int linerevealed;
   printf("\n\nCurrent status of the board is as follows:\n");
   printf("EVIDENCE:  ");
   for (count = 0; count <= 2; count ++) {
        PrintCard(GameBoard.Evidence[count]);
        printf("   ");
```

```
    }
  if (DEBUG)
    for (count = 0; count <= 19; count++)
      {
        printf("Case number %d has %d revealed",count,
        GameBoard.Cases[count].FactsShown);
        printf(" and %d unrevealed facts.\n",
                    GameBoard.Cases[count].CardsLeft-
                    GameBoard.Cases[count].FactsShown);
        PrintCase(GameBoard.Cases[count]);
      }
  printf("\n\nSHORT CASES:\n");
  for (count = 0; count <= 9;)
    {
      printf("%d: ",count);
      for (numshown = 0;numshown<=GameBoard.Cases[count].FactsShown;)
        {
          if (GameBoard.Cases[count].FactEvid[numshown].Rank == TEN)
            ten = 1;
          else ten = 0;
          PrintCard(GameBoard.Cases[count].FactEvid[numshown]);
          if ((numshown++)<GameBoard.Cases[count].FactsShown)
            printf(",");
        }
      if (!ten) printf(" ");
      printf("(%d)    ",FactsLeft(count));
      count ++;
      if (count % 2) printf ("\t\t");
      else printf ("\n");
    }
  printf("\nLONG CASES:\n");
  linerevealed = GameBoard.Cases[10].FactsShown +
                 GameBoard.Cases[11].FactsShown;
  for (count = 10; count <= 19;)
    {
      printf("%d: ",count);
      for (numshown = 0;numshown<=GameBoard.Cases[count].FactsShown;)
        {
          printf (" ");
          if (GameBoard.Cases[count].FactEvid[numshown].Rank == TEN)
            ten = 1;
          else ten = 0;
          PrintCard(GameBoard.Cases[count].FactEvid[numshown]);
          if ((numshown++)<GameBoard.Cases[count].FactsShown)
            printf(",");
        }
      if (!ten) printf(" ");
      printf("(%d)    ",FactsLeft(count));
      count ++;
      if (count % 2)
        {
          if (linerevealed > 6) printf ("\n");
          else printf ("\t");
        }
      else
        {
          if (count < 19)
            linerevealed = GameBoard.Cases[(count)].FactsShown +
                           GameBoard.Cases[(count + 1)].FactsShown;
          printf("\n");
        }
    }
  printf("\n\n");
}
```

```
}


void RemoveArgSupt(loc)
NODE * loc;
{ /* clears the support of an argument */
int n;
   for (n = 0; n < loc->numsupports; n++)
     {
       if (TEMPDEBUG)
         {
           printf("Removing the support of ");
           PrintCard(loc->support[n]->C);
           printf(".\n");
         }
       RemoveArgSupt(loc->support[n]);
printf("here.\n");
       loc->numsupports = 0;
     }
   loc->numsupports = 0;
}


void RemoveArgCard (loc)
NODE * loc;
{ /* removes the card from the arg, updates the arg */
NODE * pappy;
int n;
   for (n = 0; n < Arg[GoAhead].numheads; n++)
     {
       if (loc == Arg[GoAhead].HeadCards[n])
         {
           Arg[GoAhead].HeadCards[n]->numsupports = 0;
           Arg[GoAhead].HeadCards[n] = NULL;
           printf ("Argument number %d removed",n);
           if (n != (Arg[GoAhead].numheads-1))
             {
               Arg[GoAhead].HeadCards[n] =
                  Arg[GoAhead].HeadCards[(Arg[GoAhead].numheads-1)];
               printf(" and replaced by the last argument");
             }
           printf(".\n");
           Arg[GoAhead].numheads--;
           return;
         }
     }
   if (loc->numsupports) RemoveArgSupt(loc);
printf("BACK.\n");
   pappy = loc->parent;
   for (n = 0; n <= pappy->numsupports; n++)
     if (pappy->support[n] == loc)
       {
         if (TEMPDEBUG)
           {
             PrintCard(pappy->support[n]->C);
             printf(" removed and replaced by the last support, ");
             PrintCard(pappy->support[pappy->numsupports-1]);
             printf(".\n");
           }
         pappy->support[n] = pappy->support[pappy->numsupports-1];
         pappy->numsupports--;
       }
}


void ClearArgs(num)
```

```
void NewGame ()
{ /* sets up the initial game board - all go into pool first *
   * (done in NewPool) and then remove the cases and evid.   */
int count,numfacts;
CARD newcard;
   srand();
   NewPool();
   if (DEBUG) PrintPool();
   Shuffle();
   GameBoard.Evidence[0].Rank = JOKER;
   GameBoard.Evidence[1].Rank = JOKER;
   GameBoard.Evidence[2].Rank = JOKER;
   if (DEBUG) PrintPool();
   if (DEBUG) printf("Laying Out Board!!!\n\n");
   for (count = 0; count <= 2; count ++) {
       newcard = GetPool();
       if (CheckEvid(newcard)) {
          ReturnPool(newcard);
          count--;
          }
       else
          GameBoard.Evidence[count] = newcard;
       }
  for (count = 0; count < 10; count ++)
     for (numfacts = 0; numfacts <= 2; numfacts++)
       {
         GameBoard.Cases[count].FactEvid[numfacts] = GetPool();
         if (!numfacts)
            if (CheckEvid(GameBoard.Cases[count].FactEvid[numfacts])) {
               ReturnPool(GameBoard.Cases[count].FactEvid[numfacts]);
               numfacts--;
               }
         GameBoard.Cases[count].CardsLeft = numfacts;
         GameBoard.Cases[count].FactsShown = 0;
       }
  for (count = 10; count < 20; count ++)
     for (numfacts = 0; numfacts <= 10; numfacts++)
       {
         GameBoard.Cases[count].FactEvid[numfacts] = GetPool();
         if (!numfacts)
            if (CheckEvid(GameBoard.Cases[count].FactEvid[numfacts]))
              {
                ReturnPool(GameBoard.Cases[count].FactEvid[numfacts]);
                numfacts--;
              }
         GameBoard.Cases[count].CardsLeft = numfacts;
         GameBoard.Cases[count].FactsShown = 0;
       }
  if (DEBUG) PrintPool();
}


int GoAhead;  /* the player whose turn it is */

CARD goal;    /* the card to be proven */

/* ********************  end GAME functions   ****************** */

/* ******************   ADT ARGUMENT functions   *************** */


void InitArgs ()
{  /* initializes the arguments (simple enuf, huh??) */
   Ordered = 0;
   Arg[0].numheads = 0;   Arg[1].numheads = 0;
   Arg[0].burden = 1;   Arg[1].burden = 0;
```

```
int num;
{  /* clears a player's argument... */
int count;
   for (count = 0;count < Arg[GoAhead].numheads; count++)
      RemoveArgCard(Arg[GoAhead].HeadCards[num]);
   Arg[GoAhead].numheads = 0;
}



NODE * AddHead (plyr,Crd)
int plyr; /* the player whose argument is being added to */
CARD Crd;     /* the card being added */
{  /* adds an argument structure to the head (ie: a 'new' argument) */
   if (!InDecisions(Crd,0)) return NULL;
   Arg[plyr].HeadCards[Arg[plyr].numheads] =
                (NODE *) malloc (sizeof (NODE));
   Arg[plyr].HeadCards[Arg[plyr].numheads]->C = Crd;
   Arg[plyr].HeadCards[Arg[plyr].numheads]->depth = 0;
   Arg[plyr].HeadCards[Arg[plyr].numheads]->numsupports = 0;
   Arg[plyr].HeadCards[Arg[plyr].numheads]->argnumber = Arg[plyr].numheads;
   Arg[plyr].numheads=Arg[plyr].numheads + 1;
   if (DEBUG) {
        printf("Added ");
        PrintCard(Arg[plyr].HeadCards[(Arg[plyr].numheads - 1)]->C);
        printf(" as card %d in the head of player %d.",
                        Arg[plyr].numheads - 1,plyr);
        printf("  It has %d supports.\n",
                Arg[plyr].HeadCards[Arg[plyr].numheads - 1]->numsupports);
      }
   return (Arg[plyr].HeadCards[Arg[plyr].numheads-1]);
}


NODE * AddSupport (loc, Cd)
NODE * loc;  /* the location of the card being supported */
CARD Cd;     /* the card being added */
{  /* Adds a support card to loc; returns ptr to that support */
int count;
   if (!InCases (loc,Cd,0)) return NULL;
   for (count = 0; count < loc->numsupports; count++)
      if (EquivCard(Cd,loc->support[count])) return NULL;
   loc->challnode = 0;
   loc->support[loc->numsupports] = (NODE *) malloc (sizeof(NODE));
   loc->support[loc->numsupports]->C = Cd;
   loc->support[loc->numsupports]->numsupports = 0;
   loc->support[loc->numsupports]->parent = loc;
   loc->support[loc->numsupports]->depth = loc->depth + 1;
   loc->support[loc->numsupports]->argnumber = loc->argnumber;
   if (Arg[GoAhead].Challenged[loc->argnumber])
     {
       printf("That may sufficiently deal with the challenge on ");
       PrintCard(Cd);
       printf(".\n");
       Arg[GoAhead].Challenged[loc->argnumber] = NULL;
     }
  loc->numsupports++;
  if (DEBUG) {
       printf("Added ");
       PrintCard (Cd);
       printf(" as support number %d of card ",loc->numsupports);
       PrintCard (loc->C);
       printf(".\n");
     }
  return (loc->support[loc->numsupports-1]);
}
```

```
void PrintSupport (supt)
NODE * supt;  /* pointer to the supporting node */
{  /* prints out the supporting cards (recursively) of a node */
int countsupports;
   countsupports = supt->numsupports;
   PrintCard (supt->C);
   if (countsupports != 0)
      switch (supt->depth)
        {
           case 0: printf(" ("); break;
           case 1: printf(" ["); break;
           case 2: printf(" {"); break;
           case 3: printf(" <"); break;
           case 4: printf(" (*"); break;
           case 5: printf(" [*"); break;
           case 6: printf(" {*"); break;
           case 7: printf(" <*"); break;
           default: printf(" (");
        }
   for (countsupports=0; countsupports<=(supt->numsupports - 1);
                         countsupports++)
     {
        PrintSupport(supt->support[countsupports]);
        if ((supt->numsupports-countsupports)-1) printf (", ");
     }
   if (countsupports != 0)
      switch (supt->depth)
        {
           case 0: printf(")"); break;
           case 1: printf("]"); break;
           case 2: printf("}"); break;
           case 3: printf(">"); break;
           case 4: printf("*)"); break;
           case 5: printf("*]"); break;
           case 6: printf("*}"); break;
           case 7: printf("*>"); break;
           default: printf(")");
        }
}



void PrintArg (player)
int player;
{  /* prints out the head cards and uses PrintSupport to print out the *
    * rest of the argument for a player */
int countheads;
   for (countheads = 0;
          countheads <= (Arg[player].numheads - 1); countheads ++)
     {
        printf("Argument %d:   ",countheads);
        PrintSupport (Arg[player].HeadCards[countheads]);
        printf("\n");
     }
   printf("\n");
   if (DEBUG) printf ("Done Printing Argument for player %d\n",player);
}

void HiLitPrintSupport (supt,loc)
NODE * supt;  /* pointer to the supporting node */
NODE * loc;   /* pointer to the card to be hilit */
{  /* prints out the supporting cards (recursively) of a node *
    * and hilights the one pointed to by loc                   */
int countsupports;
   countsupports = supt->numsupports;
   if (supt == loc) HiLitPrintCard(supt->C);
   else PrintCard (supt->C);
   if (countsupports != 0) printf ("(");
   for (countsupports=0; countsupports<=(supt->numsupports - 1);
                         countsupports++)
     {
        HiLitPrintSupport(supt->support[countsupports],loc);
        if ((supt->numsupports-countsupports)-1) printf (", ");
     }
   if (countsupports != 0) printf (")");
}


void HiLitPrintArg (player,loc)
int player;
NODE * loc;
{  /* HiLitPrintSupport to print out the rest of the arg for a player, *
    * hilighting loc                                                   */
int countheads;
   for (countheads = 0;
          countheads <= (Arg[player].numheads - 1); countheads ++)
     {
        printf("  Argument %d:   ",countheads);
        HiLitPrintSupport (Arg[player].HeadCards[countheads],loc);
        printf("\n");
     }
   printf("\n");
   if (DEBUG) printf ("Done Printing Argument for player %d\n",player);
}


void ArgumentDriver()
{  /* This lets you make a specific argument structure - for use *
    when testing specific argument forms (ie for specificity)  */
NODE * loc1;
NODE * loc2;
  if (DEBUG) printf("In Arg Driver II.\n");
  loc1 = AddHead(0,MakeTheCard("Q","Sp"));
  AddSupport (loc1,MakeTheCard("J","Sp"));
  loc2 = AddSupport (loc1,MakeTheCard("K","Sp"));
  AddSupport(loc2,MakeTheCard("4","Sp"));
  AddSupport(loc2,MakeTheCard("5","Sp"));
  loc2 = AddSupport(loc1,MakeTheCard("9","Sp"));
  AddSupport(loc2,MakeTheCard("K","Cl"));
  AddHead(0,MakeTheCard("J","Di"));
  loc1 = AddHead(0,MakeTheCard("Q","Sp"));
  AddSupport (loc1,MakeTheCard("K","Sp"));
  AddSupport (loc1,MakeTheCard("4","Cl"));
  loc1 = AddHead(0,MakeTheCard("K","Sp"));
  AddSupport (loc1,MakeTheCard("3","Di"));
  loc2 = AddSupport (loc1,MakeTheCard("5","Di"));
  AddSupport (loc2,MakeTheCard("4","Cl"));
  AddSupport (loc2,MakeTheCard("4","Di"));
  AddSupport (loc1,MakeTheCard("7","He"));
  loc1 = AddHead(1,MakeTheCard("Q","He"));
  AddSupport (loc1,MakeTheCard("K","Sp"));
  loc2 = AddSupport (loc1,MakeTheCard("J","Sp"));
  AddSupport(loc2,MakeTheCard("4","Sp"));
  AddSupport(loc2,MakeTheCard("5","Sp"));
  AddHead(1,MakeTheCard("A","Di"));
}

/* ******************   end ARGUMENT functions   *************** */
```

```
/* ********************   GAME functions   ******************** */


/* ***** Specificity functions ***** */

typedef struct {
        CARD Activators [11];
        int numacts;
} ACTS; /* the activators for an argument's specificity check */

ACTS Activs; /* the activators variable */


boolean InActivators(Crd)
CARD Crd;
{ /* this function checks to see if the passed parameter is one *
   * of the activators */
int count;
   for (count = 0; count < Activs.numacts; count++)
     {
        if (DEBUG) printf ("Comparing ");
        if (DEBUG) PrintCard(Crd);
        if (DEBUG) printf(" with ");
        if (DEBUG) PrintCard(Activs.Activators[count]);
        if (DEBUG) printf(".\n");
        if ((Activs.Activators[count].Rank == Crd.Rank) &&
                (Activs.Activators[count].Color == Crd.Color))
          return 1;
     }
  return 0;
}


void GetActivators (loc)
NODE * loc;
{ /* gets all of the possible activators */
CARD tempcard;
int supporters;
int evidadd;
int evidnum1,evidnum2;
int count;
   if (DEBUG)
     {
        printf("Getting Activators for support of ");
        PrintCard(loc->C);
        printf(" which has %d supports.\n",loc->numsupports);
     }
   Activs.numacts = 0;
   evidadd = 0;
   for (supporters = 0;supporters < loc->numsupports;supporters++)
     {
        tempcard = loc->support[supporters]->C;
        Activs.Activators[Activs.numacts] = tempcard;
        Activs.numacts++;
        if ((Ordered) && (CheckEvid(tempcard) == 1))
          for (evidnum1 = 0;evidnum1 < 1;evidnum1++)
            if ((tempcard.Rank == GameBoard.Evidence[evidnum1].Rank) &&
                    (tempcard.Suit == GameBoard.Evidence[evidnum1].Suit))
              for (evidnum2 = (evidnum1+1);evidnum2 <= 2;
                                           evidnum2++)
                Activs.Activators[Activs.numacts++] =
                                           GameBoard.Evidence[evidnum2];
     }  /* this for statement gets all of the possible activators */
  if (DEBUG)
     {
        printf ("The %d Activators are:",Activs.numacts);
        for (count = 0;count<=Activs.numacts-1;count++)
          {
             PrintCard(Activs.Activators[count]);
             if ((Activs.numacts-count)-1)
                printf (", ");
          }
        printf("\n");
     }
}


boolean RecActivates (currnode)
NODE * currnode;
{ /* recursively checks to see if a given node is activated by *
   * the Activators in Activs */
int count;
boolean deeper;
   deeper = 1;
   if (InActivators(currnode->C)) return 1;
   if (!currnode->numsupports) return 0;
   for (count = 0; count<currnode->numsupports; count++)
     {
        if (DEBUG) printf("Going deeper on ");
        if (DEBUG) PrintCard(currnode->C);
        if (DEBUG) printf("'s supports.\n");
        deeper = ((deeper) &&
                  (RecActivates(currnode->support[count])));
     }
   return deeper;
}



boolean Activates(loc1,loc2)
NODE * loc1;
NODE * loc2;
{ /* An argument activates another argument if its supporting  *
   * cards (along with the less important evidence, if ordered *
   * evidence is used) fully cover the supporting cards of the *
   * other argument (with a card being covered possibly at any *
   * level of the argument)                                    */
boolean yesactivs;
int countsupp;
   yesactivs = 1;
   countsupp = 0;
   GetActivators(loc1);
   while ((yesactivs) && (countsupp < loc2->numsupports))
      yesactivs = RecActivates(loc2->support[countsupp++]);
   return yesactivs;
}


int MoreSpecific (loc1,loc2)
NODE * loc1;
NODE * loc2;
{  /* uses function Activate to tell whether the argument headed  *
    * by loc1 (returns 1), loc2 (returns 2) or neither argument   *
    * are more specific                                           */
/* An argument is more specific than another it fully Activate[s] *
 * the other argument without the reverse being true              */
int result;
   result = 0;
   if (Activates (loc1,loc2))
      result++;
```

```
    if (Activates (loc2,loc1))
       result = result + 2;
    if ((TEMPDEBUG) && (result == 3))
       printf ("Both Are Equally Specific.\n");
    else if ((TEMPDEBUG) && (!result))
       printf ("Neither is More Specific.\n");
    else if (TEMPDEBUG) printf("Argument %d is More Specific.\n",result);
    return result;
 }


 void SpecTest ()
 { /* my specificity driver test */
 int mspec;
    mspec = MoreSpecific(Arg[0].HeadCards[0],Arg[1].HeadCards[0]);
    if (!mspec)
       printf("Neither argument is more specific.\n");
    else if (mspec == 3)
       printf("Neither argument is more specific.\n");
    else printf("Argument %d is more specific.\n",mspec);
 }

 /* ***** end Specificity functions ***** */

 /* ***** Find functions ***** */

 typedef struct {
         NODE * locations [7];
         NODE * parent [7];
         int numfound;
 } REPS; /* the activators for an argument's specificity check */

 REPS Repetitions; /* the number of repetitions and pointers to the cards */


 void AddRept (loc,par)
 NODE * loc;
 NODE * par;
 { /* simply adds a repetition */
    Repetitions.parent[Repetitions.numfound] = par;
    Repetitions.locations[Repetitions.numfound] = loc;
    Repetitions.numfound++;
 }


 int RecFindRepts (loc,Cd,par)
 NODE * loc;
 CARD Cd;
 NODE * par;
 { /* traverses the argument under (and including) loc to find *
    * out the number of times card Cd occurs                    */
 int count;
    if (SameCard (loc->C,Cd))
       AddRept (loc,par);
    for (count = 0;count<=(loc->numsupports-1);count++)
          RecFindRepts (loc->support[count],Cd,loc);
    return Repetitions.numfound;
 }


 int RecFindEquiv (loc,Cd,par)
 NODE * loc;
 CARD Cd;
 NODE * par;
 { /* traverses the argument under (and including) loc to find *
    * out the number of times cards with color of Cd occur     */
 int count;
    if (EquivCard (loc->C,Cd))
       AddRept (loc,par);
    for (count = 0;count<=(loc->numsupports-1);count++)
          RecFindRepts (loc->support[count],Cd,loc);
    return Repetitions.numfound;
 }

 int GetAll (C)
 CARD C;
 {  /* This function finds ALL of the occurrances of a card C in     *
     * BOTH players arguments.  It stores them in the REPS structure *
     * and returns number found.                                     */
 int count;
 int plyr;
 int ctheads;
    plyr = GoAhead;
    Repetitions.numfound = 0;
    for (ctheads = 0;ctheads < Arg[plyr].numheads; ctheads++)
       {
          if (EquivCard(Arg[plyr].HeadCards[ctheads]->C,C))
            AddRept(Arg[plyr].HeadCards[ctheads],NULL);
          for (count = 0;count<Arg[plyr].HeadCards[ctheads]->numsupports;
                       count++)
             RecFindEquiv(Arg[plyr].HeadCards[ctheads]->support[count],C,
                       Arg[plyr].HeadCards[ctheads]);
       }
    if (GoAhead) plyr = 0;
    else plyr = 1;
    for (ctheads = 0;ctheads < Arg[plyr].numheads; ctheads++)
       {
          if (EquivCard(Arg[plyr].HeadCards[ctheads]->C,C))
            AddRept(Arg[plyr].HeadCards[ctheads],NULL);
          for (count = 0;count<Arg[plyr].HeadCards[ctheads]->numsupports;
                       count++)
             RecFindRepts(Arg[plyr].HeadCards[ctheads]->support[count],C,
                       Arg[plyr].HeadCards[ctheads]);
       }
    return (Repetitions.numfound);
 }



 NODE * FindCard (plyr,argnum,rank,suit)
 int plyr;
 int argnum;
 char * suit;
 char * rank;
 {  /* This function finds returns an (if unique) occurrance of a *
     * card with suit suit and rank rank in player plyr's argument*
     * argnum.  If no such card occurs, it returns 0.  If there  *
     * are > 1 such occurrances, each are listed, and the player  *
     * decides which to use.                                      */
 CARD C;
 word temp;
 int count;
 int numwanted;
 int s;
    Repetitions.numfound = 0;
    C = MakeTheCard(rank,suit);
    if (SameCard(Arg[plyr].HeadCards[argnum]->C,C))
       AddRept(Arg[plyr].HeadCards[argnum],NULL);
    for (count = 0;count<Arg[plyr].HeadCards[argnum]->numsupports;
                       count++)
       RecFindRepts(Arg[plyr].HeadCards[argnum]->support[count],C,
                       Arg[plyr].HeadCards[argnum]);
    if (!Repetitions.numfound) return 0;
    if (Repetitions.numfound == 1)
       return (Repetitions.locations[0]);
    else
       {
         numwanted = 99;
         while ((numwanted > Repetitions.numfound) && (numwanted >= 0))
           {
              printf ("More than one instance of ");
              PrintCard(C);
              printf(" occurrs in the argument.\nWhich of the following ");
              printf("arguments did you want?\n");
              for (count = 0; count < Repetitions.numfound; count++)
                 {
                    printf("   %d:  ",count);
                    HiLitPrintSupport(Arg[GoAhead].HeadCards[argnum],
                                      Repetitions.locations[count]);
                    printf("\n");
                 }
              printf(" > ");
              scanf("%d",&numwanted);
              gets(temp);  /* just for the stupidity of c input */
           }
         return (Repetitions.locations[numwanted]);
       }
 }


 NODE * FindAll(plyr,rank,suit)
 int plyr;
 char * suit;
 char * rank;
 {  /* This function finds returns an (if unique) occurrance of a *
     * card with suit suit and rank rank in player plyr's argu-   *
     * ment.  If no such card occurs, it returns 0.  If there     *
     * are > 1 such occurrances, each are listed, and the player  *
     * decides which to use.                                      */
 CARD C;
 word temp;
 int count;
 int ctheads;
 int wanted;
   Repetitions.numfound = 0;
   C = MakeTheCard(rank,suit);
   for (ctheads = 0;ctheads < Arg[plyr].numheads; ctheads++)
      {
         if (SameCard(Arg[plyr].HeadCards[ctheads]->C,C))
           AddRept(Arg[plyr].HeadCards[ctheads],NULL);
         for (count = 0;count<Arg[plyr].HeadCards[ctheads]->numsupports;
                       count++)
            RecFindRepts(Arg[plyr].HeadCards[ctheads]->support[count],C,
                       Arg[plyr].HeadCards[ctheads]);
      }
   if (!Repetitions.numfound) return 0;
   if (Repetitions.numfound == 1)
      return (Repetitions.locations[0]);
   wanted = 99;
   while (wanted >= Repetitions.numfound)
      {
           printf ("More than one instance of ");
           PrintCard(C);
           printf(" occurrs in the argument.\nWhich of the following ");
           printf("instances did you want?\n");
           for (count = 0; count < Repetitions.numfound; count++)
              {
                 printf("Instance %d:  \n",count);
                 HiLitPrintArg(plyr,Repetitions.locations[count]);
              }
           printf(" > ");
           scanf("%d",&wanted);
           gets(temp);  /* must for the stupidity of c input */
           if ((wanted >= 0) && (wanted < Repetitions.numfound))
              return (Repetitions.locations[wanted]);
      }
 }



 /* ***** end Find functions ***** */


 /* ***** Argument Manipulation functions ***** */

 void RecAddArgSupport(loc,addloc)
 NODE * loc;     /* the location adding to */
 NODE * addloc; /* the card to add... */
 {  /* Recursively adds support to loc */
 int count;
 NODE * addedsupt;
    addedsupt = AddSupport(loc,addloc->C);
    if (addedsupt == NULL)
      {
         printf("As ");
         PrintCardCol(addloc->C);
         printf(" is already in the support list, it is not added.\n");
         return;
      }
    if (addloc->numsupports)
       for (count = 0; count < addloc->numsupports; count++)
          RecAddArgSupport(addedsupt,addloc->support[count]);
 }


 NODE * AddArgSupport()
 {  /* Adds a support argument to loc; returns ptr to that support */
 NODE * loc;
 NODE * addloc;
 int count;
 int num;
 word cinput;
 CARD addcard;
 CARD tocard;
 int argnum;
 word temprank;
 word tempsuit;
    if (!Arg[GoAhead].numheads)
      {
         printf("You have no current arguments to add.\n");
         return NULL;
      }
    printf("\nYour current arguments:\n");
    PrintArg(GoAhead);
    printf("What argument is being added to (negative is none)?  ");
    scanf("%d",&argnum);
    gets(cinput);  /* just for the stupidity of c input */
    if (argnum < 0) return NULL;
    if (argnum > Arg[GoAhead].numheads)
      {
         printf("Invalid argument.  Must be between 0 and %d.\n",
                    Arg[GoAhead].numheads);
         return NULL;
      }
```

```
    printf("What card is being added to (form: J Cl)?  ");
    gets(cinput);
    sscanf(cinput,"%s %s",temprank,tempsuit);
    tocard = MakeTheCard(temprank,tempsuit);
    if (SameCard(tocard,(MakeTheCard("x","x"))))
      {
         printf("Invalid card to add to.\n");
         return;
      }
    loc = FindCard(GoAhead,argnum,temprank,tempsuit);
    if (!loc)
      {
         printf("That card was not found in your current");
         printf(" argument number %d.\n",argnum);
         return;
      }
    printf("What argument would you like to add?  ");
    scanf("%d",&num);
    gets(cinput);  /* just for the stupidity of c input */
    if ((num<0) || (num > Arg[GoAhead].numheads))
      {
         printf("Invalid argument.  Must be between 0 and %d.\n",
                Arg[GoAhead].numheads);
         return;
      }
    if (!(Arg[GoAhead].HeadCards[num]->numsupports))
      {
         printf("Argument %d has only a head card.\n",num);
         return;
      }
    if (!(EquivCard(loc->C,Arg[GoAhead].HeadCards[num]->C)))
      {
         printf("Invalid support attempt.\nThe head card of argument");
         printf(" %d is not the same as ",num);
         PrintCard(loc->C);
         printf(".\n");
         return;
      }
    if (loc->numsupports)
      {
         printf("NOTE: That is quite possibly an invalid move.  Since ");
         printf("this is\na testing phase and I trust you, I will allow");
         printf(" it.\n");
         /*return;*/
      }
    addloc = Arg[GoAhead].HeadCards[num];
    for (count = 0; count < addloc->numsupports; count ++)
       RecAddArgSupport(loc,addloc->support[count]);
    printf("\nYour argument is now:\n");
    PrintArg(GoAhead);
    return (loc->support[loc->numsupports-1]);
}


/* ***** end Argument Manipulation functions ***** */


/* ********************   ADT DIALOGUE functions   ***************** */


void InitDials()
{  /* initializes the dialogues */
   Dial[0].type = -1; Dial[1].type = -1;
   Dial[0].numargs = 0; Dial[1].numargs = 0;
}
```

```
void MakeDialogue(num,ArgList)
int num;
int ArgList [10];
{  /* constructs the dialogue state for the player using ArgList */
int count;
   for (count = 0; count < num; count++)
      Dial[GoAhead].Arg[count] = Arg[GoAhead].HeadCards[ArgList[count]];
   Dial[GoAhead].numargs = num;
   Dial[GoAhead].object = NULL;
}


boolean RecCheckArgForChal(loc)
NODE * loc;
{  /* used by below to recursively check for challenged cards */
int count;
boolean challenged;
   if (loc->challnode)
     {
        if (TEMPDEBUG)
          {
             printf("The card ");
             PrintCard(loc->C);
             printf(" in argument %d is still challenged or refuted.\n",
                loc->argnumber);
          }
        return 1;
     }
   challenged = 0;
   count = 0;
   while ((!challenged) && (count < loc->numsupports))
     {
        if (loc->challnode)
          {
             if (TEMPDEBUG)
               {
                  printf("The card ");
                  PrintCard(loc->C);
                  printf(" in argument %d is still challenged or refuted.\n",
                      loc->argnumber);
               }
             challenged = 1;
          }
        challenged = RecCheckArgForChal(loc->support[count]);
        count++;
     }
   return challenged;
}



boolean CheckArgForChal(argnum)
int argnum;
{  /* Checks to see if a argument can be used in a dialogue or if it *
    * has unmet challenges or refutations against it...               */
   if (Arg[GoAhead].HeadCards[argnum]->challnode)
     {
        if (TEMPDEBUG)
           printf("The head card is still under challenge or refutation.\n");
        return 1;
     }
   return RecCheckArgForChal(Arg[GoAhead].HeadCards[argnum]);
}
```

```
void PrintDialogue(plyr)
int plyr;
{  /* displays the dialogue, silly... */
int count;
   if (Dial[plyr].type == -1)
     {
        if (GoAhead == plyr)
           printf("You have ");
        else printf("Your opponent has ");
        printf("no current dialogue.\n");
        return;
     }
   if (Dial[plyr].type == 0)
     {
        if (plyr == GoAhead)
           printf("Challenging ");
        else printf("Your opponent challenged ");
        PrintCard(Dial[plyr].object->C);
        printf(".\n");
        return;
     }
   if (Dial[plyr].type == 1)
     {
        if (plyr == GoAhead)
           printf("Refuting ");
        else printf("Your opponent refuted ");
        PrintCard(Dial[plyr].object->C);
        printf(" with:  ");
        PrintSupport(Dial[plyr].Arg[0]);
        printf(".\n");
        return;
     }
   if (plyr == GoAhead)
      printf("You are using the following arguments:\n");
   else
      printf("Your opponent used the following arguments:\n");
   for (count = 0; count < Dial[plyr].numargs; count++)
     {
        printf("%d:  ",count);
        PrintSupport(Dial[plyr].Arg[count]);
        printf("\n");
     }
   printf("\n");
}


void ArgumentBuilder()
{  /* This lets you make an argument structure - makes sure it *
    * is a valid argument being built, too...                  */
NODE * loc1;
NODE * loc2;
CARD Cd;
   Cd = GameBoard.Cases[17].FactEvid[0];
   loc1 = AddHead(0,Cd);
   Cd = GameBoard.Cases[18].FactEvid[0];
   loc1 = AddHead(0,Cd);
   Cd = GameBoard.Cases[19].FactEvid[0];
   loc1 = AddHead(0,Cd);
   Cd = GameBoard.Cases[10].FactEvid[0];
   loc1 = AddHead(1,Cd);
   Cd = GameBoard.Cases[11].FactEvid[0];
   loc1 = AddHead(1,Cd);
   Cd = GameBoard.Cases[12].FactEvid[0];
   loc1 = AddHead(1,Cd);
}
```

```
boolean PresentDialogue()
{  /* Returns 1 if new dialogue given, updates Dialogue */
int ArgList [10]; /* array of argument numbers */
int numargs;      /* number of argument numbers */
int nextarg;   /* the next arg in the arg list */
boolean done;  /* done parsing? */
word cinput;
int count;
boolean nope;
   Dial[GoAhead].type = 2;
   done = 0;
   numargs = 0;
   if (!Arg[GoAhead].numheads)
     {
        printf("You have no current arguments to use.\n");
        return 0;
     }
   printf("Please input the argument(s) you would like to use");
   printf(" (negative will\nend inputing).");
   printf("  > ");
   while (!done)
     {
        scanf ("%d",&nextarg);
        gets(cinput);
        if (nextarg < 0) done = 1;
        else if (nextarg >= Arg[GoAhead].numheads)
           printf("Sorry, but you only have arguments 0 to %d.\n  > ",
                   Arg[GoAhead].numheads-1);
        else
          {
             nope = 0;
             if (CheckArgForChal(nextarg))
               {
                  printf("That argument still contains challenged/refuted ");
                  printf("cards and can't be used.\n");
                  nope = 1;
               }
             for (count = 0; count < numargs; count++)
               if (nextarg == ArgList[count])
                 {
                    printf("You are already using that argument.\n");
                    nope = 1;
                 }
             if (!nope)
               {
                  ArgList[numargs] = nextarg;
                  numargs ++;
               }
             printf("  > ");
          }
     }
   MakeDialogue(numargs,ArgList);
   PrintDialogue(GoAhead);
   return 1;
}



boolean PresentDialogueTry()
{  /* Returns 1 if new dialogue given, updates Dialogue */
int ArgList [10]; /* array of argument numbers */
```

```
int numargs;       /* number of argument numbers */
char *cinput;    /* the (being parsed?) string */
int nextarg;    /* the next arg in the arg list */
int a,b,c,d,e,f,g,h,i,j;
boolean done;  /* done parsing? */
boolean error; /* cinput not proper list */
   done = 0;
   error = 0;
   a = -1; b = -1; c = -1; d = -1; e = -1; f = -1; g = -1; h = -1;
   i = -1; j = -1;
   printf("Please input the argument(s) you would like to use, ");
   printf("separated\nby commas (negative will return to options).");
   printf("  > ");
   gets(cinput);  /* just for the stupidity of c input */
   numargs = sscanf(cinput,"%d, %d, %d, %d, %d, %d, %d, %d, %d, %d",
                    a,b,c,d,e,f,g,h,i,j);
printf("Gotten.\n");
printf("numargs = %d, nums are %d, %d, %d, %d, %d, %d, %d, %d, %d, %d",
        numargs,a,b,c,d,e,f,g,h,i,j);
printf("\nPrinted.\n");
   while (!done)
     {
        if (sscanf(cinput,"%d; %s",nextarg,cinput) != 2) error = 1;
        else error = 0;
printf("here.\n");
printf("Argument gotten: %d, rest is: %s, error: %d.\n",nextarg,cinput,error);
        if (error)
          error = (!sscanf(cinput,"%d",nextarg));
        if (error)
          {
             printf("Inproper argument list format.\n");
             return PresentDialogue();
          }
        if (nextarg < 0) return 0;
     }
   MakeDialogue(numargs,ArgList);
   return 1;
}


void HiLitPrintDial (player,loc)
int player;
NODE * loc;
{  /* HiLitPrintSupport to print out the rest of the arg for a player, *
    * hilighting loc                                                   */
int countheads;
   for (countheads = 0;countheads < Dial[player].numargs;countheads ++)
     {
        printf("  Dialogue %d:   ",countheads);
        HiLitPrintSupport (Dial[player].Arg[countheads],loc);
        printf("\n");
     }
   printf("\n");
   if (DEBUG) printf ("Done Printing Dialogue for player %d\n",player);
}



NODE * FindDial(plyr,rank,suit)
int plyr;
char * suit;
char * rank;
{  /* This function finds returns an (if unique) occurrance of a *
    * card with suit suit and rank rank in player plyr's dial-    *
    * ogue.  If no such card occurrs, it returns 0.  If there     *
    * are > 1 such occurrances, each are listed, and the player   *
    * decides which to use.                                       */
CARD C;
word temp;
int count;
int ctheads;
int wanted;
  Repetitions.numfound = 0;
  C = MakeTheCard(rank,suit);
  for (ctheads = 0;ctheads < Dial[plyr].numargs; ctheads++)
     {
        if (SameCard(Dial[plyr].Arg[ctheads]->C,C))
          AddRept(Dial[plyr].Arg[ctheads],NULL);
        for (count = 0;count<Dial[plyr].Arg[ctheads]->numsupports;count++)
           RecFindRepts(Dial[plyr].Arg[ctheads]->support[count],C,
                  Dial[plyr].Arg[ctheads]);
     }
  if (!Repetitions.numfound) return NULL;
  if (Repetitions.numfound == 1)
    return (Repetitions.locations[0]);
  wanted = 99;
  while (wanted >= Repetitions.numfound)
     {
         printf ("More than one instance of ");
         PrintCard(C);
         printf(" occurrs in the argument.\nWhich of the following ");
         printf("instances did you want?\n");
         for (count = 0; count < Repetitions.numfound; count++)
            {
               printf("Instance %d:  \n",count);
               HiLitPrintDial(plyr,Repetitions.locations[count]);
            }
         printf(" > ");
         scanf("%d",&wanted);
         gets(temp);  /* must for the stupidity of c input */
         if ((wanted >= 0) && (wanted < Repetitions.numfound))
            return (Repetitions.locations[wanted]);
     }
}


boolean FindAllDial(plyr,C)
int plyr;
CARD C;
{  /* This function finds returns all of the ooccurrance(s) of a *
    * card C in player plyr's dialogue.                           *
    * If no such card occurrs, it returns 0.  If there are *
    * > 1 such occurrances, each are returned through the  *
    * repetitions structure.                                      */
word temp;
int count;
int ctheads;
int wanted;
  Repetitions.numfound = 0;
  for (ctheads = 0;ctheads < Dial[plyr].numargs; ctheads++)
     {
        if (EquivCard(Dial[plyr].Arg[ctheads]->C,C))
          AddRept(Dial[plyr].Arg[ctheads],NULL);
        for (count = 0;count<Dial[plyr].Arg[ctheads]->numsupports;count++)
           RecFindRepts(Dial[plyr].Arg[ctheads]->support[count],C,
                  Dial[plyr].Arg[ctheads]);
     }
  if (!Repetitions.numfound) return NULL;
  if (Repetitions.numfound == 1) return 1;
  else return 2;  /* more than 1, so why not use the fact that bool is int */
}
```

```
NODE * Refute()
{  /* Returns the loc of the card being refuted, updates the Dialogue */
NODE * loc;
CARD refuted;
int iinput;
int plyr;
word cinput;
word temprank;
word tempsuit;
   if (GoAhead) plyr = 0;
   else plyr = 1;
   Dial[GoAhead].type = 1;
   if ((Dial[plyr].type == 0) || (Dial[plyr].type == -1))
     {
        printf("Invalid.  You must respond to your opponent's last respon");
        printf("se:\n");
        PrintDialogue(plyr);
        return NULL;
     }
   if (!Arg[GoAhead].numheads)
     {
        printf("You cannot must make an argument against a card to refute");
        printf(" it.\n");
        return NULL;
     }
   printf("What card in your opponent's argument are you trying to ref");
   printf("ute?  > ");
   gets(cinput);
   sscanf(cinput,"%s %s",temprank,tempsuit);
   refuted = MakeTheCard(temprank,tempsuit);
   if (SameCard(refuted,(MakeTheCard("x","x"))))
     {
        printf("Invalid card.\n");
        return NULL;
     }
   loc = FindDial(plyr,temprank,tempsuit);
   if (!loc)
     {
        printf("That card was not found in your opponent's dialogue.\n");
        return NULL;
     }
   iinput = -1;
   while ((iinput < 0) || (iinput >= Arg[GoAhead].numheads))
     {
        printf("And what argument of yours are you using to refute?  > ");
        scanf("%d",&iinput);
        gets(cinput);  /* just for the stupidity of c input */
        if (iinput < 0) return NULL;
        if (iinput > Arg[GoAhead].numheads-1)
           printf("Sorry, but you only have %d arguments.\n  > ",
                  Arg[GoAhead].numheads);
        else if (!OppCard(Arg[GoAhead].HeadCards[iinput]->C,loc->C))
          {
             printf("You must refute your opponent's card, ");
             PrintCard(loc->C);
             printf(" with an argument starting with\n   an opposite card.\n");
             return NULL;
          }
        else
          {
             Arg[GoAhead].Challenged[loc->argnumber] = loc;
             loc->challnode = 1;
             Dial[GoAhead].Arg[0] =
                  Arg[GoAhead].HeadCards[iinput];
             Dial[GoAhead].numargs = 1;

             Dial[GoAhead].type = 1;
          }
     }
   return loc;
}


NODE * Challenge()
{  /* Returns ptr to card challenged, updates Dialogue */
NODE * loc;
CARD challenged;
word cinput;
word temprank;
word tempsuit;
int plyr;
  if (GoAhead) plyr = 0;
     else plyr = 1;
   Dial[GoAhead].type = 0;
   if ((Dial[plyr].type == 0) || (Dial[plyr].type == -1))
     {
        printf("Invalid.  You must respond to your opponent's last respon");
        printf("se:\n");
        PrintDialogue(plyr);
        return NULL;
     }
   printf("What card in your opponent's argument are you trying to chal");
   printf("leng?  > ");
   gets(cinput);
   sscanf(cinput,"%s %s",temprank,tempsuit);
   challenged = MakeTheCard(temprank,tempsuit);
   if (SameCard(challenged,(MakeTheCard("x","x"))))
     {
        printf("Invalid card.\n");
        return NULL;
     }
   loc = FindDial(plyr,temprank,tempsuit);
   if (!loc)
     {
        printf("That card was not found in your opponent's dialogue.\n");
        return NULL;
     }
   if (loc->numsupports)
     {
        printf("The ");
        PrintCard(loc->C);
        printf(" is not challengable, as it has valid supports already.\n");
        printf("You may only challenge unsupported cards.\n");
        return NULL;
     }
   Dial[GoAhead].numargs = 0;
   Arg[GoAhead].Challenged[loc->argnumber] = loc;
   loc->challnode = 1;
   if (TEMPDEBUG)
     {
        PrintCard(Arg[GoAhead].Challenged[loc->argnumber]);
        printf(" in argument number %d is now under challenge.\n",
                        loc->argnumber);
     }
   return loc;
}


boolean SuccRefute()
{  /* Returns whether a refutation attempt is (if burdened) more specific *
    * or (if not burdened) at least not less specific                      */
int specres;
```

```
    specres = MoreSpecific(Dial[GoAhead].Arg[0],Dial[GoAhead].object);
    if (Arg[GoAhead].burden)
       if (specres == 1)
          return 1;
    if (!Arg[GoAhead].burden)
       if (specres < 2)
          return 1;
    return 0;
}

boolean CheckDialHeads()
{  /* Returns whether one of the heads in the dialogue is the goal */
int i;
   for (i = 0; i < Dial[GoAhead].numargs; i++
      if (Arg[i]->C == goal) return 1;
   return 0;
}


boolean CheckForSpec(loc)
NODE * loc;
{  /* Used by RecCheckDialSpec - returns if the opponent has any *
    * arguments more specific than loc in their dialogue         */
int i;
int plyr;
   if (GoAhead) plyr = 0;
      else plyr = 1;
   for (i = 0; i < Dial[plyr].numhargs; i++)
      if


boolean RecCheckDialSpec (loc)
NODE * loc;
{  /* Recursive function that checks node loc to see if the opponent has *
    * any arguments more specific than loc in their dialogue using       *
    * CheckForSpec                                                       */


boolean CheckDialSpec(burdn)
int burdn;
{  /* Returns whether there are ANY arguments in the opponent's dialogue *
    * which are not less specific (if burdn'ed) or more specific (if not *
    * burdn'ed) than one of the cards in your argument, using            *
    * RecCheckDialSpec                                                   */
int i;
int specfound;
   specfound = 0;
   if (GoAhead) plyr = 0;
      else plyr = 1;
   for (i = 0; i < Dial[GoAhead].numargs; i++)
      specfound = specfound + RecCheckDialSpec(Dial[GoAhead].Arg[i]);
   if (TEMPDEBUG)
     {
       printf("There were %d instances of more/not less specific card ");
       printf("usages found.\n",specfound);
     }
   return specfound;
}


{  /* Returns whether the current response is a sufficient response */
int i;
int plyr;
  if (Dial[GoAhead].type == 0)
    {
      if (Dial[GoAhead].object == NULL)
```

```
int numargs;    /* the number of args cited */
word cinput;    /* the input of the player  -  a string */
int iinput;    /* the input of the player  -  an int */
word typedial;  /* 0 = challenge, 1 = refute, 2 = present dialogue */
boolean suff;
   iinput = -1;
   while ((iinput < 0) || (iinput > 3))
     {
        printf("You can:  \n\t(0) challenge an argument in your opponent's");
        printf(" current dialogue,\n\t(1) refute a card in your opponent's");
        printf(" current dialogue,\n\t(2) present a (new) dialogue, ");
        printf("or \n\t(3) go back to options?\n   > ");
        scanf("%d",&iinput);
        gets(cinput);  /* just for the stupidity of c input */
     }
   switch (iinput)
     {
        case 0: Dial[GoAhead].object = Challenge(); break;
        case 1: Dial[GoAhead].object = Refute(); break;
        case 2: PresentDialogue(); break;
     }
   if ((iinput < 0) || (iinput > 2)) return 0;
   return Sufficient();
}



/* ****************   end DIALOGUE definitions   *************** */


/* ***** Two Player functions ***** */


void ResetStr(resetme)
word resetme;
{ /* just sets a string to dummy nil... for words... */
int count;
   resetme[0] = 'n';
   resetme[1] = 'u';
   resetme[2] = 'l';
   resetme[3] = 'l';
   for (count = 4;count < 20; count++)  resetme[count] = NULL;
}


void Possibilities()
{ /* prints out the possibilities */
   printf("During each turn, the valid responses are:\n");
   printf("\td[isplay]a[rguments] -> displays the current arguments.\n");
   printf("\td[isplay]b[oard] -> displays the current board status.\n");
   printf("\ta[dd] -> adds a card to one of your arguments, parameters:\n");
   printf("\t\targument number, card to add, card to add under.\n");
   printf("\tr[emove]a[rgument] -> removes an argument.\n");
   printf("\tr[estate] -> restate arguments - allows you to add or remove");
   printf("\n\t\targuments to your current argument list.\n");
   printf("\tf[lip]r[esource] -> flips a resource card.\n");
   printf("\tf[lip]c[ase] -> flips a card under a case, parameters are:\n");
   printf("\t\tl/s (long or short case), case number.\n");
   printf("\te[nd] -> ends turn.\n");
   printf("\tq[uit] -> ends game.\n");
   printf("Pretty boring, eh?\n\n\n");
}


void Intro()
```

```
        {
           printf("You formed no valid challenge.\n");
           return 0;
        }
      printf("Your response has been determined to be sufficient, as");
      printf(" it challenges an\nunsupported card in your opponent's");
      printf(" dialogue arguments.\n");
      return 1;
   }
 if (Dial[GoAhead].type == 1)
   {
      if (Dial[GoAhead].object == NULL)
        {
           printf("You formed no valid refutation.\n");
           return 0;
        }
      if (SuccRefute())
        {
           printf("Your response has been determined to be sufficient, as");
           printf(" it refutes a card\nin your opponent's dialogue argum");
           printf("ents with ");
           if (!Arg[GoAhead].burden)
              printf("an argument that is not less specific.\n");
           else printf("a more specific argument.\n");
           return 1;
        }
      else
        {
           printf("Your refutation is not sufficient, as it was ");
           if (Arg[GoAhead].burden)
              printf("not more specific than\n");
           else
              printf("less specific than ");
           printf("your opponent's.\n");
           return 0;
        }
   }
 else
   {
      if (!Dial[GoAhead].numargs)
        {
           printf("You inputed no arguments to use as a dialogue.\n");
           return 0;
        }
      if (GoAhead) plyr = 0;
         else plyr = 1;
      if (!GoAhead)
        {
           if (!CheckDialHeads())
             {
                printf("As you are trying to prove ");
                PrintCard(goal);
                printf(", that card should be the head\nof one of your");
                printf(" arguments.\n");
                return 0;
             }
        }
      CheckDialSpec(Arg[GoAhead].burden);
   }
}


boolean Dialogue()
{  /* Gets a dialogue response from the player, creates the Dial[] */
   /* Returns whether the dialogue was a sufficient response */
int Args [10];  /* the array of arguments cited */
```

```
{ /* gives the introduction, sets up the global(ish) variables */
  printf("Welcome to the Two-player version of An Argument Game.\n");
  printf("\t\tR. P. Loui, W. Chen, patent pending.\n\n\n");
  Possibilities();
}


boolean Add(lastadd)
boolean lastadd; /* for if there is to be multiple adds */
{ /* adds a card to the player's argument */
NODE * loc;
CARD addcard;
CARD tocard;
int iinput;
word cinput;
int argnum;
boolean head;
boolean tempread;
word temprank;
word tempsuit;
   tempread = 0;
   ResetStr(tempsuit);
   ResetStr(temprank);
   if (!Arg[GoAhead].numheads)
     {
        argnum = 0;
        head = 1;
        tempread = 1;
     }
   else while (!tempread)
     {
        printf("What argument would you like to add to?  > ");
        scanf ("%d",&iinput);
        gets(cinput);  /* just for the stupidity of c input */
        if ((!lastadd) && (iinput < 0)) return 1;
        if (DEBUG)
           printf ("Adding to player %d's argument %d out of %d\n",
                    GoAhead ,iinput,Arg[GoAhead].numheads);
        if ((Arg[GoAhead].numheads > 0) && (iinput >= 0) &&
              (iinput <= (Arg[GoAhead].numheads-1)))
          {
             if (DEBUG) printf("Adding a support card.\n");
             head = 0;
             argnum = iinput;
             tempread = 1;
          }
        else if ( (((!iinput) && (!Arg[GoAhead].numheads)) ||
                  ((iinput == (Arg[GoAhead].numheads)) &&
                   (iinput)) )
          {
             if (DEBUG) printf("Adding a head card.\n");
             head = 1;
             argnum = iinput;
             tempread = 1;
          }
        else if (iinput < 0)
           printf("Invalid argument number, Player %d.\n",GoAhead);
        else printf("Player %d, you only have %d headcards!\n",
                    GoAhead,Arg[GoAhead].numheads);
     }
   tempread = 0;
   while (!tempread)
     {
        printf("What card is being added (form: J Cl)?  ");
        gets(cinput);
        sscanf(cinput,"%s %s",temprank,tempsuit);
```

```
        addcard = MakeTheCard(temprank,tempsuit);
        ResetStr(tempsuit);
        ResetStr(temprank);
        if (DEBUG)
          {
            printf("Adding made card ");
            PrintCard(addcard);
            printf(".\n");
          }
        if (!SameCard(addcard,(MakeTheCard("x","x")))) tempread = 1;
      }
    tempread = 0;
    if (!head) while (!tempread)
     {
        printf("What card is being added to (form: J Cl)?  ");
        gets(cinput);
        sscanf(cinput,"%s %s",temprank,tempsuit);
        tocard = MakeTheCard(temprank,tempsuit);
        if (DEBUG)
          {
            printf("Add to made card ");
            PrintCard(tocard);
            printf(".\n");
          }
        if (!SameCard(tocard,(MakeTheCard("x","x")))) tempread = 1;
     }
    if (head) AddHead(GoAhead,addcard);
    else
      {
        loc = FindCard(GoAhead,argnum,temprank,tempsuit);
        if (loc) AddSupport(loc,addcard);
        else
          {
            printf("That card was not found in your current");
            printf(" argument number %d.\n",argnum);
          }
      }
    ResetStr(tempsuit);
    ResetStr(temprank);
    if (lastadd) printf("\nYour argument is now:\n");
    if (lastadd) PrintArg(GoAhead);
    return 0;
}


void DisplayArg ()
{ /* Allows a player to display the arguments */
   printf("\nYour current arguments are:\n");
   PrintArg(GoAhead);
   printf("Your opponent's current arguments are:\n");
   if (GoAhead) PrintArg(0);
   else PrintArg(1);
}


void RemoveArg()
{ /* Allows a player to remove an argument from his/her arg list */
int removeme;
word cinput;
   printf("What argument would you like to remove (negative removes");
   printf(" none)?\n\t> ");
   scanf ("%d",&removeme);
   gets(cinput);  /* just for the stupidity of c input */
   if (removeme < 0) return;
   else if (removeme < Arg[GoAhead].numheads)
     {
        Arg[GoAhead].HeadCards[removeme]->numsupports = 0;
        Arg[GoAhead].HeadCards[removeme] = NULL;
     }
   printf ("Argument number %d removed",removeme);
   if (removeme != (Arg[GoAhead].numheads-1))
     {
       Arg[GoAhead].HeadCards[removeme] =
          Arg[GoAhead].HeadCards[(Arg[GoAhead].numheads-1)];
       printf(" and replaced by the last argument");
     }
   printf(".\n");
   Arg[GoAhead].numheads--;
}


void Restate ()
{ /* Allows a player to restate his/her argument */
NODE * tempptr;
CARD deleted;
CARD tempcard;
int temp;
boolean done;
word cinput;
word temprank;
word tempsuit;
  printf("\nYour current arguments:\n");
  PrintArg(GoAhead);
  printf("First, lets delete the cards you dont' want to use.\n");
  printf("Name cards to delete - if a card with apport is named,");
  printf("all of its\n  support will be deleted as well.\n");
  printf("Inputing x x on a line will finish deleting.\n\n");
  done = 0;
  ResetStr(tempsuit);
  ResetStr(temprank);
  while (!done)
    {
       printf("What card is being deleted (form: J Cl)?   > ");
       gets(cinput);
       sscanf(cinput,"%s %s",temprank,tempsuit);
       if ((temprank[0] == 'x') && (tempsuit[0] == 'x'))
          done = 1;
       if (!done) deleted = MakeTheCard(temprank,tempsuit);
       if ((!done) &&(!SameCard(deleted,MakeTheCard("x","x"))))
         {
           tempptr = FindAll(GoAhead,temprank,tempsuit);
           if (!tempptr) printf("Card not found.\n");
           else
              {
                 if (DEBUG)
                   {
                      printf("Have found ");
                      PrintCard(tempptr);
                      printf(" to delete.\n");
                   }
                 RemoveArgCard(tempptr);
                 printf("\nYour current arguments:\n");
                 PrintArg(GoAhead);
              }
         }
       ResetStr(temprank);
       ResetStr(tempsuit);
    }
done = 0;
printf("Now it is time to add cards.\nTo stop adding, simply ");
printf("input a negative number for the argument number.\n");
while (!done)
    {
       done = Add(0);
       printf("\nYour current arguments:\n");
       PrintArg(GoAhead);
    }
}



void FlipRes ()
{ /* Allows a player to flip a resource card */
  printf("The card flipped is ");
  PrintCard(GetResource());
  printf(", leaving you with %d resources.\n",Res.ResNum);
}


void FlipCase ()
{ /* Allows a player to flip on a case */
CARD Cd;
int flipcase;
int numleft;
word temp;
boolean gotcard;
int i;
int reps;
CARD FlippedOn;
  printf("What case would you like to flip on, using the short cases");
  printf(" as cases\n\t0 through 9 and the long ones as 10 through 19.\n");
  printf("   > ");
  scanf("%d",&flipcase);
  gets(temp);  /* just for the stupidity of c input */
  if ((flipcase < 0) || (flipcase > 19))
    {
       printf("Invalid case number.  Cases are 0-19.\n");
       return;
    }
  if (FactsLeft(flipcase))
    {
       FlippedOn = GameBoard.Cases[flipcase].FactEvid[0];
       printf("The card revealed is ");
       Cd = RevealCase(flipcase);
       PrintCard(Cd);
       printf(" leaving %d facts left unturned in that case.\n",
              FactsLeft(flipcase));
       gotcard = CheckFlipped(flipcase,Cd);
    }
  else printf("There are no cards left unturned in that case.\n");
  if ((AUTOADD) && (gotcard))
    {
       reps = GetAll(FlippedOn);
       if (reps)
         {
            printf("Attempting to add the card to the %d occurance",reps);
            if (reps>1) printf("s");
            printf(" in your argument");
            if (Arg[GoAhead].numheads>1) printf("s");
            printf(".\n");
         }
       for (i=0; i < reps; i++)
           AddSupport(Repetitions.locations[i],Cd);
    }
}


int End ()
{ /* Allows a player to end his turn */
int next;
   if (!GoAhead) next = 1;
   else next = 0;
   printf ("\nPlayer %d, it is your turn.\n",next);
   printf ("Your current arguments:\n");
   PrintArg(next);
   printf ("Your opponent's current arguments:\n");
   PrintArg(GoAhead);
   printf("\n");
   return next;
}



void TwoPlayers ()
{ /* This procedure runs the game for two players */
word cinput;
int iinput;
boolean tempread;
boolean done;
word temprank;
word tempsuit;
   Intro();
   InitDials();
   tempread = 0;
   ResetStr(tempsuit);
   ResetStr(temprank);
   PrintBoard();
   if (TEMPDEBUG)
     {
        GoAhead = 0;
        goal = GameBoard.Cases[19].FactEvid[0];
        printf("For ease of input, setting player 0 trying to argue for ");
        PrintCard(goal);
        printf("\n\twith 20 resource cards.\n\n");
        InitResources(20);
     }
   if (!TEMPDEBUG) while (!tempread)
     {
        printf("What card is being proven(form: J Cl)?   > ");
        gets(cinput);
        sscanf(cinput,"%s %s",temprank,tempsuit);
        goal = MakeTheCard(temprank,tempsuit);
        if (!SameCard(goal,(MakeTheCard("x","x")))) tempread = 1;
        ResetStr(tempsuit);
        ResetStr(temprank);
     }
   done = 0;
   printf ("The current list of arguments for player 0 is:\n");
   PrintArg(0);
   printf ("The current list of arguments for player 1 is:\n");
   PrintArg(1);
   printf("\n\n");
   while (!done)
     {
        printf("Player %d's turn ",GoAhead);
        printf("(da/db/a/aa/ra/r/fc/fr/d/e/q/?)  > ");
        gets(cinput);
        if ((cinput[0] == 'a') && (cinput[1] == 'a'))
              AddArgSupport();
        else if (cinput[0] == 'a') Add(1);
        else if ((cinput[0] == 'r') && (cinput[1] == 'a'))
              RemoveArg();
```

```
        else if (cinput[0] == 'r') Restate();
        else if (cinput[0] == 'e') GoAhead = End();
        else if (cinput[0] == 'q') return;
        else if ((cinput[0] == 'd') && (cinput[1] == 'a'))
                DisplayArg();
        else if ((cinput[0] == 'd') && (cinput[1] == 'b'))
               PrintBoard();
        else if (cinput[0] == 'd')
                { if (Dialogue()) GoAhead = End(); }
        else if ((cinput[0] == 'f') && (cinput[1] == 'r'))
                 FlipRes();
        else if ((cinput[0] == 'f') && (cinput[1] == 'c'))
                 FlipCase();
        else
            Possibilities();
    }
}



/* ************************************************************ *
 * *********************  MAIN PROGRAM TIME  ****************** *
 * ************************************************************ */


main()
{
   printf("\n");
   NewGame();
   InitArgs();
   ArgumentBuilder();
   TwoPlayers();
   printf("\n\nHope you had fun - at least as much as I had coding this");
   printf(" Bad-Boy...\n\n");
}
```

References

Allen, L. (1966) "WFF 'N PROOF: The game of modern logic," Autotelic Instructional Materials Publishers, New Haven.

Carlson, E. (1969), *Learning Through Games: A New Approach to Problem Solving,* Public Affairs Press, Washington, D.C.

Ellington, H I, Addinall, E, and Percival, F (1981), *Games and Simulations in Science Education*, Kogan Page, London.

Gibbs, G. I. (1974a), *APLET Yearbook of Educational and Instructional Technology 1975/75*, Kogan Page, London.

Gibbs, G. I. (1974b), *Handbook of Games and Simulation Exercises*, E & F N Spon Ltd, London.

Gibbs, G. I., Howe, A. (1974), *Academic Gaming and Simulation in Education and Training*, Kogan Page, London.

Loui, R. (1987) "Defeat among arguments." *Computational Intelligence 4.*

Loui, R., Chen, W. (1992), *An Argument Game*, W.U.C.S. TR 92-47, 1992.

Loui, R., Norman, J., Stiefvater, K., Olson, J., and Costello, A. (1992), "Computing specificity," W.U.C.S. TR 92-47.

Mulac, M. E. (1971), *Educational Games for Fun*, Harper and Row Publishers, New York.

Poole, D. (1985) "Preferring the most specific theory," *Proceedings of the International Conference on Artifical Intelligence (IJCAI).*

Pollack, J. (1987) "Defeasible reasoning," *Cognitive Science.*

Twelker, P A (1981), "Simulation and media," in Tansey, P J (ed) *Educational Aspects of Simulation*, McGraw-Hill, London.