The Crème Language:

by: Anthony Chan, Cheryl Lau, James Leslie, Josh Mackler

Programming Languages and Translators – Spring 2003

Contents

1 INTRODUCTION ..... - 4 -
1.1 Crème Overview ..... - 4 -
1.2 Crème Functionality ..... - 5 -
1.3 Conclusion ..... - 6 -
2 TUTORIAL ..... - 7 -
2.1 Necessary Elements ..... - 7 -
2.2 Step By Step - Solitaire ..... - 7 -
2.2.1 Card Types ..... - 7 -
2.2.2 Locations for Cards ..... - 8 -
2.2.3 Turn System ..... - 9 -
2.2.4 Rules ..... - 10 -
3 LANGUAGE REFERENCE MANUAL ..... - 12 -
3.1 Introduction ..... - 12 -
3.2 Lexical structure ..... - 12 -
3.2.1 Comments ..... - 12 -
3.2.2 Identifiers (Names) ..... - 12 -
3.2.3 Keywords ..... - 13 -
3.2.4 Primitive Data Types ..... - 13 -
3.2.5 Strings ..... - 13 -
3.2.6 Separators ..... - 13 -
3.2.7 Operators ..... - 13 -
3.2.8 White Space ..... - 14 -
3.2.9 Semicolon Usage ..... - 14 -
3.3 Syntax notation ..... - 14 -
3.4 Expressions ..... - 14 -
3.4.1 Hierarchy ..... - 14 -
3.4.1.1 Sample Code ..... - 14 -
3.4.1.2 Meaning ..... - 14 -
3.4.2 Operator notation ..... - 15 -
3.4.3 Deck Instantiation ..... - 16 -
3.5 Test Conditionals ..... - 16 -
3.5.1 Purpose ..... - 16 -
3.5.2 atDepth() ..... - 16 -
3.5.3 contains() ..... - 16 -
3.5.4 isEmpty() ..... - 17 -
3.5.5 isTotallyEmpty() ..... - 17 -
3.5.6 upToDepth() ..... - 17 -
3.6 Ranksys ..... - 17 -
3.6.1 Purpose ..... - 17 -
3.6.2 Sample Code ..... - 17 -
3.6.3 Meaning ..... - 18 -
3.7 Grid ..... - 19 -
3.7.1 Purpose ..... - 19 -

3.7.2 Sample Code - 19 -
3.7.3 Meaning - 19 -
3.8 Deal - 19 -
3.8.1 Purpose - 19 -
3.8.2 Sample Code - 20 -
3.8.3 Meaning - 20 -
3.9 Rules - 20 -
3.9.1 Purpose - 20 -
3.9.2 Sample Code - 21 -
3.9.3 Meaning - 21 -
3.10 Take and Place and Action - 21 -
3.10.1 Purpose - 21 -
3.10.2 Sample Code - 22 -
3.10.3 Meaning - 22 -
3.11 Move - 22 -
3.11.1 Purpose - 22 -
3.11.2 Sample Code - 22 -
3.11.3 Meaning - 23 -
3.12 Turns - 23 -
3.12.1 Purpose - 23 -
3.12.2 Sample Code: - 23 -
3.12.3 Meaning - 23 -
4 Project Plan - 24 -
4.1 Team Responsibilities - 24 -
4.2 Project Timeline and Log - 24 -
4.3 Software Development Environment - 25 -
4.4 Programming Style Guide - 25 -
5 ARCHITECTURAL DESIGN - 27 -
6 TEST PLAN - 28 -
7 LESSONS LEARNED - 33 -
8 APPENDIX - 35 -
8.1 Crème Grammar - 35 -
8.2 Crème Java Code - 36 -
8.2.1 Deck.java - 36 -
8.2.2 CardStack.java - 41 -
8.2.3 Card.java - 43 -
8.2.4 RankElement.java - 43 -
8.2.5 Grid.java - 44 -
8.2.6 Turn.java - 55 -
8.2.7 RuleGrid.java - 56 -
8.2.8 ActionGrid.java - 58 -
8.2.9 RuleType.java - 59 -
8.2.10 Rule.java - 60 -
8.2.11 FlipRule.java - 66 -
8.2.12 Conditional.java - 67 -
8.2.13 AtDepth.java - 68 -

8.2.14 NotAtDepth.java - 70 -
8.2.15 Contains.java - 71 -
8.2.16 NotContains.java - 72 -
8.2.17 IsEmpty.java - 74 -
8.2.18 NotIsEmpty.java - 75 -
8.2.19 IsTotallyEmpty.java - 76 -
8.2.20 NotIsTotallyEmpty.java - 77 -
8.2.21 UpToDepth.java - 78 -
8.2.22 NotUpToDepth.java - 80 -
8.2.23 ActionConditional.java - 81 -
8.2.24 Action.java - 82 -
8.2.25 Crème.g - 84 -
8.2.26 Map.Java - 101 -

1 INTRODUCTION

Crème's purpose is to integrate different types of card games into one scheme. The language is designed to facilitate the quick and simple development of classic single player card games as well as variations on classic single card games. By creating domain specific code, we avoid the tedious and mundane task of writing low level code.

The language allows the programmer to focus on developing the game rather than thinking about how the code should be implemented. Instead of wondering what structures would be needed to implement the code, the programmer can answer more high level questions: How is the game played? What are the rules of the game?

1.1 Crème Overview

Crème: A simple, powerful, scalable, graphical and architecture neutral language.

Taking a page from the Java white paper, we are also using buzzwords to describe the feature set of our language. Since Crème code is compiled into Java, it naturally inherits many of the same attributes, most importantly, being architecture neutral. However, Crème also possesses features that are exclusively helpful to the specific language domain. The following explanations on the characteristics of Crème will show how we have addressed the issues of card-game programming.

Simple When designing a card game, it is most important to focus on the game itself, instead of having to deal with complicated nested loops and overwhelming conditional statements. Alternatively, attention should be paid to making the game enjoyable, addictive, and challenging. Crème allows the programmer to develop on a higher plane of abstraction, dealing only with ideas that are intrinsic to card games such as decks, cards and rules. The programmer can focus on the rules of the card game without needing to worry about how those rules should be implemented. Also, there is no messy GUI code to deal with, simply define how you would like the game to look, and our graphical engine can do the rest.

Powerful Through the functions provided by Crème, a lot can be accomplished with just a few lines of code. Point made.

Scalable Crème was designed as an easy to use language that allows maximum scalability without bogging the programmer down. When simple, classic games are constructed, alternative versions of those games can be easily constructed by simply changing a few rules and adding a few lines of code. For instance, FreeCell can easily be created just by using the code that the programmer has already designed for other related games like Solitaire or Spider Solitaire.

Graphical Since Crème is compiled into Java code, the transition into a graphical user interface, GUI, is seamless. No extra code is necessary for the programmer to write a fully functioning graphical application other than the rules of the game he wishes to create. The compiler knows where to place the various stacks of cards defined and creates mouse action listeners according to the layout of the game. The programmer can not directly affect the graphics, since his/her job is only to specify the rules. Crème does the rest.

Architecture Neutral Crème code is compiled into Java byte code which makes it architecture neutral. Crème code can essentially be developed on all environments for which there is a Crème compiler, and then executed on any system equipped with a Java Virtual Machine. This "write once, run everywhere" paradigm means that your Crème games will enjoy compatibility across multiple platforms which will make the games more popular more quickly.

1.2 Crème Functionality

Graphics Crème offers a graphical representation of cards. The player interacts with and views the game through the GUI. The graphical interface provides a way for a player to play, see the cards that have been dealt to the playing field and easily interact with them.

Card Movement Cards or sets of cards can be passed from location to location on the playing field. The program can automatically move cards based on certain conditions or the programmer can give that functionality to the player for added game control.

Ranking Cards can be compared to each other based on a ranking system. The ranking system determines whether a card has a higher precedence over another card or has the same rank as the other card. A card can be defined to match another card when both cards have the same rank. For example, a 4 card may have a higher rank over a 3 card. It is also sometimes useful for a programmer to define his own ranking system.

1.3 Conclusion

In conclusion, we are the Crème de la Crème. Keep in mind, Crème stands for Cards Really Excite ME! Crème simplifies the making of card games by allowing programmers to design with a high level perspective. Crème supports basic actions of card games which can be invoked with single statements. Variations on previously made games can be made with ease. A card game operates on a graphical user interface, providing a way of input from the game player and a visual display for the game player. Since Crème is compiled into Java, Crème code can be run on any machine that has a Java Virtual Machine.

2 TUTORIAL

Card games are created with Crème. As in all traditional card games, cards, locations, turns and rules are the main elements involved in Crème programs. In this tutorial, we will slowly build up to a fully functional version of Solitaire, explaining each step in the process.

2.1 Necessary Elements

As in many languages, certain distinct features of Crème code need to be defined in every program for it to be properly read in and interpreted by the compiler. As described above, these entail what types of cards are being used, where these cards can be placed on the playing field, what the turn system governing the progression of play is and what rules direct the game play.

2.2 Step By Step - Solitaire

2.2.1 Card Types

For a program to run, it needs to have a basic card structure. This is accomplished using Decks and Ranksys. The Deck is the data structure that holds the individual cards before they are dealt out to the playing field. Before they are actually created and placed in the data structure, they must be given values. This is done by Ranksys calls. The following call instantiates both the Deck data structure and the Ranksys value system:

```
Deck.Ranksys = 3;
```

This call creates a Deck with 3 value systems associated with it. In the next few lines of code, we give the program more information as to what the values for each card are:

```
Deck.Ranksys.1 = (King > Queen > Jack > Ten > Nine > Eight > Seven > Six > Five > Four > Three > Two > Ace);
Deck.Ranksys.2 = (SPADES.CLUBS > HEARTS.DIAMONDS);
Deck.Ranksys.3 = (BLACK [2. (SPADES, CLUBS)].(RED [2. (HEARTS, DIAMONDS)]);
```

Calls to Ranksys are interpreted in a way that is easy for the programmer to manipulate. The code declaring Ranksys.1 gives the cards their first values. After Ranksys.1 is called, there are thirteen cards

that have been created (their names have been explicitly defined, for example: Jack).

The code declaring Ranksys.2 creates more cards by giving each of the values declared in Ranksys.1 another property, namely a suite. So, after Ranksys.2 is called, 52 cards have been created, as each of the cards from Ranksys.1 now has turned into four new cards, with the original value from Ranksys.1 and a new value (the suite) added to it. Now instead of there only being one Jack card, there are four Jack cards: Jack SPADES, Jack CLUBS, Jack HEARTS, and Jack DIAMONDS.

The final Ranksys call does not create any new cards. It has been specially declared to only add an attribute to cards that have already been created. This attribute is the color of the card (either BLACK or RED as in a normal deck of cards). Similar to a traditional deck of cards, only the Spades and Clubs are black, while only the Hearts and Diamonds are red. When the value BLACK is given, it is followed by brackets containing two fields. The first field is the Ranksys that this Ranksys is modifying. Here, Ranksys.3 is modifying Ranksys.2. The second field is the values of the Ranksys being modified that will receive this particular attribute. In this case, all SPADES and CLUBS are given the BLACK attribute while all HEARTS and DIAMONDS are given the RED attribute. Now instead of there being four colorless Jacks, there are: Jack SPADES BLACK, Jack CLUBS BLACK, Jack HEARTS RED and Jack DIAMONDS RED. In four lines of code, an entire 52 card deck has been completely described and understood by the compiler.

2.2.2 Locations for Cards

Since Crème creates GUI based games, it is imperative that the compiler know where the programmer would like the cards to be placed visually, so the user can easily interact with them. Again, in only a few lines of code, this is done completely.

```
Grid=[7,2];
```

In this simple call, the entire playing field has been defined. It has created a grid of seven columns by two rows. The cells of this grid are accessed in the same manner as n-sized arrays are accessed in Java, with 0 being the first cell and n-1 being the last cell. Once the playing field has been created, cards need to be dealt to these cells. This is done by a series of deal commands that take cards from the Deck and pass them out to the cells. The following calls deal out the cards for Solitaire:

```
Deck.deal ~ Grid[1-6, 1](0,1);  //1st Deck.deal call
Deck.deal ~ Grid[2-6, 1](0,1);  //2nd Deck.deal call
Deck.deal ~ Grid[3-6, 1](0,1);  //  .
Deck.deal ~ Grid[4-6, 1](0,1);  //  .
Deck.deal ~ Grid[5-6, 1](0,1);  //  .
```

```
Deck.deal ~ Grid[ 6 , 1](0,1);  //6th Deck.deal call
Deck.deal ~ Grid[0-6, 1](1,0);  //7th Deck.deal call

Deck.deal ~ Grid[0 , 0](0, 50);  //8th Deck.deal call
```

The `Deck.deal` call takes cards from the deck and deals them to the desired cells in the grid. The destination cells are defined after the tilde ("~") and follow the same convention as a double array in Java (i.e. – the first seven calls are dealing to the second row, denoted by the 1). Both of the fields of the Grid location can take a single argument (`1` in the second field of the first call) or a range of arguments (`1-6` in the first field of the first call). The numbers in parentheses after the Grid location of the just-dealt cards defines the visibility of those cards to the player and how many should be dealt. In the first six calls, there is a 0 in the first field (meaning zero cards should be dealt that are facing the player or "face up") and a 1 in the second field (one card should be dealt that is not facing the player or "face down"). This means that a single card that is "face down" is dealt to each of the specified Grid spaces for each of the first six Deck.deal calls. The seventh Deck.deal card is similar to the previous ones with one notable exception. Instead of dealing a card face down to the specified Grid spaces, it deals a card face up. Thus, the top card (most recently added) of each Grid space is facing the player, while all cards beneath that one are not.

2.2.3 Turn System

Now that the cards have been dealt, the next step is defining how the game is played and when the game should end. This is determined by the turn loop:

```
turn.1(Grid[3-6,0].atDepth(1)[King,*,*]) {
```

This line of code establishes when this "turn" (in Solitaire there is only one turn, as explained later) is up. At that time, the program moves to the end of the loop, which we will find is the end of the program. So, when this turn is completed the game is over. In Solitaire, the game has been won when all of the cards in the Deck have been placed in their respective place in the top right of the Grid space. By "respective place" I mean that they are ordered from `Ace` to `King` according to Ranksys.1, for each suit that there is, according to Ranksys.2. The argument that turn.1 takes in this example states this in Crème. It says that the turn should not end until the following has occurred: each of the four top right Grid spaces (`Grid[3-6,0]`) have Kings of any suite (`[King,*,*]`) in their top location (`.atDepth(1)`). Now that the end result has been defined, the rules of the game can be declared.

2.2.4 Rules

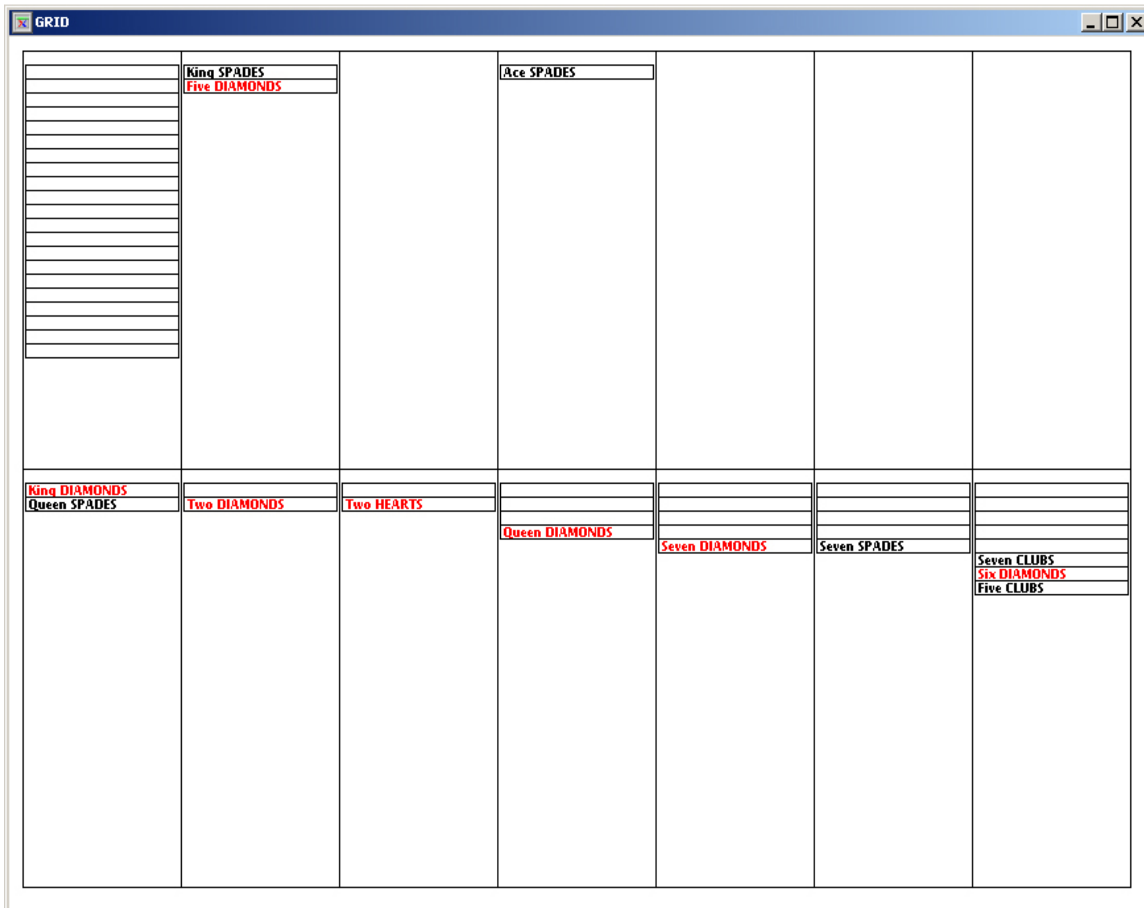
Rules turn out to be the most important aspect of Crème games. Without these, players would just be moving cards around, without knowing what to do. Rules are declared on the specific Grid spots that they affect. And last for the entire turn. If there is more than one turn, the Rules must be declared for each turn individually. For Solitaire, the following rules govern play:

```
Grid[0-6, 1].rules ~ if isTotallyEmpty then {  place[King,*,*];
                             else {  place[(-1,+1,*)(-1,-1,*)]; }}
Grid[3-6, 0].rules ~ if isTotallyEmpty then {  place[Ace,*,*); }
                             else {  place(+1,2,*); }}
Grid[0,0].rules ~ if isTotallyEmpty then
                     (action(Grid[1,0].move(*,*)~Grid[0,0](0,*););
                             else
                     {  action(Grid[0,0].move(0,3)~Grid[1,0](3,0);); }}
Grid[0,1].rules ~ if isEmpty then
                     {action(Grid[0,1].move(0,1)~Grid[0,1](1,0);); }
Grid[1,1].rules ~ if isEmpty then
                     {action(Grid[1,1].move(0,1)~Grid[1,1](1,0);); }
Grid[2,1].rules ~ if isEmpty then
                     {action(Grid[2,1].move(0,1)~Grid[2,1](1,0);); }
Grid[3,1].rules ~ if isEmpty then
                     {action(Grid[3,1].move(0,1)~Grid[3,1](1,0);); }
Grid[4,1].rules ~ if isEmpty then
                     {action(Grid[4,1].move(0,1)~Grid[4,1](1,0);); }
Grid[5,1].rules ~ if isEmpty then
                     {action(Grid[5,1].move(0,1)~Grid[5,1](1,0);); }
Grid[6,1].rules ~ if isEmpty then
                     {action(Grid[6,1].move(0,1)~Grid[6,1](1,0);); }
```

These calls completely define the rules of Solitaire. The first rule describes the bottom portion of the Grid. Whenever there are no cards on a Grid space in that area (`isTotallyEmpty`), a player can place a King of any suite there. If there are already cards there, then only an alternating color in descending Ranksys.1 order can be placed there. This would be equivalent to placing a `Jack HEARTS RED` on a `Queen CLUBS BLACK`. The next rule describes the top right portion of the Grid. This is where cards are placed to eventually win the game (this is the area that the turn conditional is monitoring). This rule states that if there are no cards in this area, only an Ace of any suite can be placed there. If a card is already on this Grid space, then only the card in that suit (`place(+1,2,*)`) that has the next highest Ranksys.1 over the card presently there (`place(+1,2,*)`) can be placed on top of it. The next rule is for going through the stack of available cards that the user can freely move. This is done by moving

three cards from this Grid space (that are NOVIEW) to the space next to it (that is ALLVIEW). This continues until the Grid space becomes empty. This is all defined in the else statement of the rule. The if portion defines what should happen when there are no more cards in this Grid space. When this happens, all of the cards that have been moved to the space to the right of this Grid space are immediately moved back and flipped back over. Very easily, the remaining cards of the deck can be iterated through for Solitaire. The rest of the rules in this section are again for the bottom portion of the Grid. Here is where cards that are not viewable but on the stack in this Grid space can be flipped over for use. This is done using an action (meaning the user has clicked and released in this Grid space) that tells the program to take the top card that is NOVIEW and turn it into an ALLVIEW card on this same stack.

The following is the GUI that appears when the program is run. The game has been started, and cards have been moved to all of the different parts of the Grid space:



That simply, the entire game of Solitaire has been defined and is ready for use. This completes the Tutorial and now it's your turn to start creating some games!

3 LANGUAGE REFERENCE MANUAL

3.1 Introduction

Crème is a language based on Java that allows programmers to quickly and intuitively create their own GUI based card games. The benefits of a Java based language are clear as the "write once, run everywhere" idea is very attractive to the types of card games that Crème can produce, since it is logical that these will be broadcast over the internet for mass consumption. While the present focus of the language is on single player games and functionality, an important feature of Crème is its extensibility which, for example, easily allows for more players to be added through the same interface or through networking protocols, among other things. Another benefit of a Java based language is the graphical capabilities that are associated with that language. While the programmer can specifically create a custom GUI layout for their particular game in Crème, a default layout is available that places the cards so that they can be easily seen on the screen throughout the game.

3.2 Lexical structure

There are eight types of tokens that are used in Crème.

3.2.1 Comments

The characters // introduce a comment which continues for the rest of the line. Multiple line comments are delimited by the /* and */ characters.

3.2.2 Identifiers (Names)

An identifier is a sequence of alphanumeric characters, starting with a letter. The underscore ("_") counts as a letter. Upper and lower case letters are considered to be distinct.

3.2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

Objects:

```
Deck        Grid
```

Declarers:

```
Ranksys
```

Conditionals:

```
if        else       then            atDepth
contains  isEmpty    isTotallyEmpty  upToDepth
```

Associated to Objects:

```
deal        place
rules       take
```

Associated to Rules:

```
action      depth
top
```

3.2.4 Primitive Data Types

All primitive data types (integer, character, etc.) are used as they are in Java.

3.2.5 Strings

Strings are used in the same manner as in Java. For Crème in particular, their function is to name cards, players, etc.

3.2.6 Separators

The following ASCII characters are separators (punctuators):

```
( )  { }  [ ]
```

3.2.7 Operators

The following ASCII character sequences are operators:

```
=    >    !    ~    .
+    -    ^    *
```

3.2.8 White Space

White space is considered to be independent in Crème.

3.2.9 Semicolon Usage

Semicolons are used to end certain calls in Crème code. All Deck calls must terminate with a semicolon (these include Ranksys declarations, and deals from the deck). When the Grid is first defined, the call must end with a semicolon. Grid calls in Grid rule descriptions (those appearing on the right side of the tilde inside the conditional) must be terminated with a semicolon. Also, all action calls in Grid rule descriptions must end with a semicolon. The Grid rule itself needs no semicolon.

3.3 Syntax notation

In the syntax notation used in this manual literal words and characters are represented in `gothic`.

3.4 Expressions

3.4.1 Hierarchy

3.4.1.1 Sample Code

```
... (K.W.J.10>9) ...
```

3.4.1.2 Meaning

When two values are separated by “.” it means that the values to the immediate left and the immediate right of the “.” are equivalent. When two values are separated by “>” it means that all values to the left of the “>” have a greater value than all values to the right. Also, card hierarchies are determined by the order of the Ranksys calls. Lower number Ranksys calls have precedence over higher number Ranksys calls.

3.4.2 Operator notation

The following is a summary of what the operators in Crème mean:

"^" -> number of cards to make
example: A^4[2.(CLUBS,DIAMONDS)] -> create 4 cards with the "A" value and the CLUBS and DIAMONDS values from Ranksys.2.

"!" -> mapping onto all except
example: J[2!(SPADES)] -> create a set of cards with the "J" value and all values of Ranksys.2 except for the SPADES value.

"*" -> any valid value of the specified Ranksys.
Note: "*" cannot be used with specifying actual locations on the Grid

"X=" where X would normally be a mapping -> rank equivalent match needed from this Ranksys.
example:
```
Deck.Ranksys.1 = (K.Q.J>10>9);
Deck.Ranksys.2 = (SPADES,HEARTS);
…place[(2,*)(2=,*)];…
```
means that for (2=,*), K SPADES K HEARTS and K SPADES Q HEARTS are both valid while for (2,*), K SPADES K HEARTS is valid but K SPADES Q HEARTS is not.

"~" -> do the action described to the left of the "~" to the object described to the right of the "~."
example: see sections 3.9.

"X" where X would normally be a mapping -> this number of exact matches needed from this Ranksys.
example:
```
Deck.Ranksys.1 = (K.Q.J>10>9);
Deck.Ranksys.2 = (SPADES,HEARTS);
…place[(2,*)(2=,*)];…
```
means that for (2=,*), K SPADES K HEARTS and K SPADES Q HEARTS are both valid while for (2,*), K SPADES K HEARTS is valid but K SPADES Q HEARTS is not.

"+X" or "-X" where X is an integer -> only the card this many levels of precedence higher or lower than present will be accepted.

example: `…place (+1,-1)…`

means that only the cards that are one higher than the present value in the first Ranksys and one less than the present value in the second Ranksys will be accepted.

3.4.3 Deck Instantiation

The Deck is an object of cards. An automatic instantiation in all Crème programs is that of the Deck that will be used in the program. This deck is referred to simply as Deck.

3.5 Test Conditionals

3.5.1 Purpose

The purpose of test conditionals is so that rules and turns can be tested and based on the results of those tests, actions can take place or code can be executed. There are five types of conditionals and they are joined by their "not" counterparts. "Not conditional" is coded as "!conditional" where "conditional" is one of the following test conditionals:

3.5.2 atDepth()

```
turn.1(Grid[2,0].atDepth(7)[King,SPADES]) { code }
```

If the specified card (`[King, SPADES]`) is at the specified depth (7) at this Grid space (`Grid[2,0]`) then the turn has completed and the next block of code after the end parenthesis for this turn is executed.

3.5.3 contains()

```
Grid[1-2,2].rules ~ if contains[*,HEARTS] then { actions }
```

If the specified card (`[*,HEARTS]`) is located anywhere within the viewable stack of cards for the given Grid space (`Grid[1-2,2]`) then the actions specified are executed and the program moves to the next line.

3.5.4 isEmpty()

```
Grid[5,3].rules ~ if isEmpty then {   actions  }
```

If the specified Grid space (`Grid[1-2,2]`) does not contain any `VIEWABLE` cards (the space can contain cards that are face down) then the actions specified are executed and the program moves to the next line.

3.5.5 isTotallyEmpty()

```
Grid[3,12].rules ~ if isTotallyEmpty then {  actions  }
```

If the specified Grid space (`Grid[3,12]`) does not contain ANY cards (no `ALLVIEW` and no `NOVIEW`) then the actions specified are executed and the program moves to the next line.

3.5.6 upToDepth()

```
turn.1(Grid[3,5].upToDepth(4)[*,HEARTS]) {  code  }
```

If the specified card (`[*, SPADES]`) can be found anywhere in the stack of cards on this Grid space (`Grid[3,5]`) up to and including the specified depth (`4`) then the turn has completed and the next block of code after the end parenthesis for this turn is executed.

3.6 Ranksys

3.6.1 Purpose

Ranksys defines the system by which cards in the deck are ranked. In the beginning of all Creme programs, the size of Ranksys must be explicitly declared, as well as the values that are associated with each of the Ranksys

3.6.2 Sample Code

```
Deck.Ranksys = 3;
Deck.Ranksys.1 = (A>K.Q.J.10>9>8>7>6>5>4>3>2); //Ordering of card
//values
Deck.Ranksys.2 = (SPADES>HEARTS>CLUBS>DIAMONDS); //Ordering of suites
Deck.Ranksys.3 = (BLACK[2.SPADES,CLUBS].RED[2.(HEARTS,DIAMONDS)]);
```

3.6.3 Meaning

In the code above, there are four separate Ranksys calls. The first declares the size of Ranksys. The rest of the calls describe how the cards will be created. The second describes the values that cards will take. The highest value is "A," and the lowest is "2." The values of "K" "Q" "J" and "10" are all equivalent and are higher than all other values except "A." The third describes the suites that cards can take. The highest suite is SPADES and the lowest is DIAMONDS. The last describes the colors that will be associated with the given cards when they are created in the GUI. All of the SPADES and CLUBS cards will be BLACK while all of the HEARTS and DIAMONDS will be RED. These two values of the third Ranksys are equivalent and have no precedence over each other. After these four declarations are made, a complete deck is made by default at compile time by creating individual cards that have as many characteristics as there are Ranksys definitions (in this case three) with each card having a single value taken from each Ranksys. In the sample code, this would mean that each card has a characteristic from the first Ranksys, a characteristic from the second Ranksys and a characteristic of the third Ranksys. For example, after the declarations, the cards 4 CLUBS BLACK and J HEARTS RED are created.
The ordering of the Ranksys calls determines the precedence of the cards after they have been made. For instance, in the calls above, the following precedence is created:

```
3 HEARTS RED > 3 DIAMONDS RED > 2 HEARTS RED
```

since the values of cards are declared before the suites of cards when Ranksys is declared. The creation of specific cards can also be accomplished through Ranksys declarations. For instance, if the code was changed to:

```
Deck.Ranksys.1 =
(A^4[2.(SPADES,HEARTCLUBS)]>K.Q.J[2.(CLUBS)]>2[2!(SPADES)]);
Deck.Ranksys.2 = (SPADES>HEARTCLUBS>CLUBS);
```

where HEARTCLUBS is another type of suite, it would mean that the following cards are created (no color Ranksys was defined, so these cards are the default):

```
A SPADES, A SPADES, A SPADES, A SPADES, A HEARTCLUBS, A HEARTCLUBS, A
HEARTCLUBS, A HEARTCLUBS, K SPADES, K HEARTCLUBS, K CLUBS, Q SPADES, Q
HEARTCLUBS, Q CLUBS, J CLUBS, 2 HEARTCLUBS, 2 CLUBS
```

3.7 Grid

3.7.1 Purpose

The Grid is an object that is represented as an area in the GUI of the program where cards are dealt to when they are out of the Deck.

3.7.2 Sample Code

```
Grid = [3,3]; //Creates a two dimensional grid of size 9

Deck.deal ~ Grid[0,1](5,2); //Deal to [0,1] on the grid with the player
//able to view 5 of them and not able to view 2 of them.
```

3.7.3 Meaning

In the code above, a Grid is instantiated and cards are dealt to a specific area of that Grid with different permissions associated with the extent to which each of the cards can be viewed by the player (scope). In creating the Grid, the notation is [row, column] where both the rows and the columns start at cell 0. An important note to keep in mind is that many games in Crème require a kind of "temporary" card holder, where cards are placed while other cards that were previously below them are moved to other Grid spots. The programmer must decide how many of these temporary spots their program requires and they must implement them themselves.

3.8 Deal

3.8.1 Purpose

Deal is a method that can be used on the Deck object to distribute cards randomly to different places in the Grid.

3.8.2 Sample Code

```
Deck.deal ~ Grid[1-6, 1](0,1);  //One card from the Deck is dealt to
//each of the 2nd through 7th columns of the 2nd row and it is not visible
//to the player
```

3.8.3 Meaning

In the code above, cards are dealt from Deck to the Grid locations [1-6, 1] using the "~" operator. The two numbers that proceed the location to which the cards are dealt to represent the extent to which the cards are viewable (since cards that are dealt are random, it doesn't matter which specific cards that are being dealt get which scope). The breakdown of this representation is as follows:

```
(ALLVIEW,NOVIEW)
```

ALLVIEW means that the player can see the values of the cards. NOVIEW means that the player cannot see the values of the cards. When the deck is being represented in the GUI, all of the cards must be set to NOVIEW as no one should be able to see which cards are still in the deck.

3.9 Rules

3.9.1 Purpose

Rules is a method that can be used on Grid objects so that only specified procedures can be performed at any set time on a cell. Rules must be placed inside of turns. Care must be taken when declaring rules, as Crème is evaluated line by line. If line 10 in the code has a rule that makes a certain Grid location empty, all rules in that turn following line 10 that perform an action when this particular Grid location is empty will be executed. This is often not the programmer's intent, so forethought is needed in the placement of rules inside of turns.

3.9.2 Sample Code

```
Grid[0-1,0].rules ~ place [(*,+1)(2,+2)[K,SPADES]]; //Only allow these
//procedures to be performed on the specified cells
```

3.9.3 Meaning

In the code above, specific types of cards are allowed to be placed on the current top card (or empty space) in the first three cells of the first column. The following explicitly describes what is allowable using the Ranksys system described in section 7.3:

(*,+1) -> any values of Ranksys.1 and the next highest value in Ranksys.2.
(2,+2) -> two of the same value in Ranksys.one and an exact match of the value that is two greater than the top card presently in the cell in Ranksys.two.
[K,SPADES] -> an exact match of the card K SPADES.
[*,SPADES] -> any SPADES cards allowed

A special case is shown in the last two examples of this sample. When one or more specific cards can be placed in a rule (the cards have at least some definitive aspect, be it a specific value in Ranksys.1 or Ranksys.2 or both, and no ambiguity like "*" or "+2" involved) it is placed in brackets ("[" and "]") to make it obvious and distinct.

3.10 Take and Place and Action

3.10.1 Purpose

Take, place and action are used to denote which moves are allowable to and from certain Grid cells. Take is evaluated when the mouse button is pushed down. Place is evaluated when the mouse button is released over a cell that is not the cell where the mouse button was pushed down, while action is evaluated when the mouse button is released over the cell that originally received the mouse click.

3.10.2 Sample Code

```
Grid[0-6,1].rules ~ if isTotallyEmpty then { place [[K,*,*]; };
//Place only Kings on empty cells
Grid[0,0].rules ~ if isTotallyEmpty then
{action(Grid[1,0].move(*,*)~Grid[0,0](0,0,*););}
//If the top left cell is clicked and there are no cards available
//move all cards from the cell next to it to this cell
```

3.10.3 Meaning

In the code above, a conditional is used to show the place rule, which is that if any of the first 6 cells of the second column are empty, the only allowable move is to place a "K" of any suite there. A point to note is the use of `[[K,*] (2,+2)]`. As explained above, the hard brackets are used for unambiguous card calls (there must be a K of any suit) while all other calls are placed in parentheses. When two such card calls are together, they are grouped by hard brackets. If only one card call exists, it is only surrounded by whatever end symbols it would normally have (i.e. – `[K,*]` or `(2,+2)`, NOT `[[K,*]]` or `[(2,+2)]`). In the second part of the code, a conditional is again used to show the action rule, which is that if no cards are in the top left Grid space and the player clicks in that space, all of the cards from the Grid space next to it are moved over to this Grid space.

3.11 Move

3.11.1 Purpose

`Move` is a method that can be used on `Grid` objects to shift the top card of either the `ALLVIEW` or `NOVIEW` portions of each cell of the `Grid` to another cell of the `Grid` where the scope of the card in the new cell is explicitly specified.

3.11.2 Sample Code

```
Grid[0,0].move(1,0) ~ Grid[1,0](1,0); //Move top (one) ALLVIEW card from
//cell [0,0] to cell [1,0] with scope remaining ALLVIEW
```

3.11.3 Meaning

In the code above, the top card with ALLVIEW scope in cell [0,0] is moved to cell [1,0] and the scope remains ALLVIEW. A "flip" command (where a card goes from face value showing to the back of the card showing and vice versa) could easily be done by simply using a move command from a cell to itself and changing the scope from ALLVIEW to NOVIEW or vice versa as follows:

```
Grid[1,0](1,0).move ~ Grid[1,0](0,1);
```

3.12 Turns

3.12.1 Purpose

The conditionals used in Crème are used in a similar manner as in Java, with the addition of turn which functions in a way similar to a while statement, where the rules specified inside the turn statement are in effect as long as the condition that turn is looking for has not been met yet.

3.12.2 Sample Code:

```
turn.1(Grid[1,1].atDepth(1)[K,*])
       { specified rules }
```

3.12.3 Meaning

As long as the top ALLVIEW card in the first cell of the first column is not a value equivalent to a "K," then specified rules will be in effect. As soon as that cell contains a card with a value of "K" then the turn is over.
Another important note regarding conditionals in Crème is that open brackets and parentheses take the closest match (closed brackets and parentheses) as their terminus.

4 Project Plan

4.1 Team Responsibilities

We divided the work to suit each teammate's strengths. This involved giving Joshua the task of completing the Lexer, the Parser and the rest of the front-end, Cheryl and Anthony the tasks of programming in Java (both code generation and GUI implementation) and the rest of the back-end, and James the task of writing and updating the Language Reference Manual and the Final Project paper so everyone knew the language changes. We were all involved in deciding the necessary functionality of Crème, the syntax of the code and in writing the final Crème code for our games.

4.2 Project Timeline and Log

We used the following as a rough timeline and log for when different aspects of the project needed to get done by and other important dates in the history of Crème:

January 28, 2003	First Team Project Meeting.
February 11, 2003	Crème is decided to be project language.
February 16, 2003	White Paper completed.
February 20, 2003	Code conventions formed.
February 26, 2003	Lexer/Parser started.
March 7, 2003	Java code generation started.
March 26, 2003	Language Reference Manual completed.
April 12, 2003	Graphical User Interface started.
April 20, 2003	Testing of Java and Crème code started.
May 3, 2003	Lexer/Parser completed.
May 6, 2003	Java code generation completed.
May 7, 2003	Graphical User Interface completed.
May 9, 2003	Testing of Java and Crème functionality completed.
May 11, 2003	Final Project completed.
May 12, 2003	Final Project presentation planned.

This is by no means an exhaustive list of the accomplishments made throughout the semester. The team spent countless hours (pretty much

wherever we went) discussing the different parts of the language, deciding what still needed to be implemented and what could be removed.

4.3 Software Development Environment

Java and Crème code were written and compiled on CUNIX and CLIC lab computers using emacs and ejava. The ANTLR code was written using ANTLR 2.7.2.

4.4 Programming Style Guide

The following is a general guide to the programming style that is used in Crème:

-**Comments** are place at the top of the source code describing what the game is (the name of the game, the programmer's name, the object of the game, etc.).

-**Blank lines** separate each block of code. The following are types of code blocks in Crème:

-Deck and Ranksys instantiation
-Grid instantiation
-Deck deals to common areas on the Grid
-turn calls
-rules for each conditional type (i.e. – isEmpty should be in a different block than !isTotallyEmpty)
-in text comments

-**turn** calls are followed by a space and their opening brace. The closing brace occupies its own line at the end of the loop.

Example:
```
turn.1(Grid[0, 0].atDepth(3)[Ten, HEARTS, RED] {

//code

}  //closing brace
```

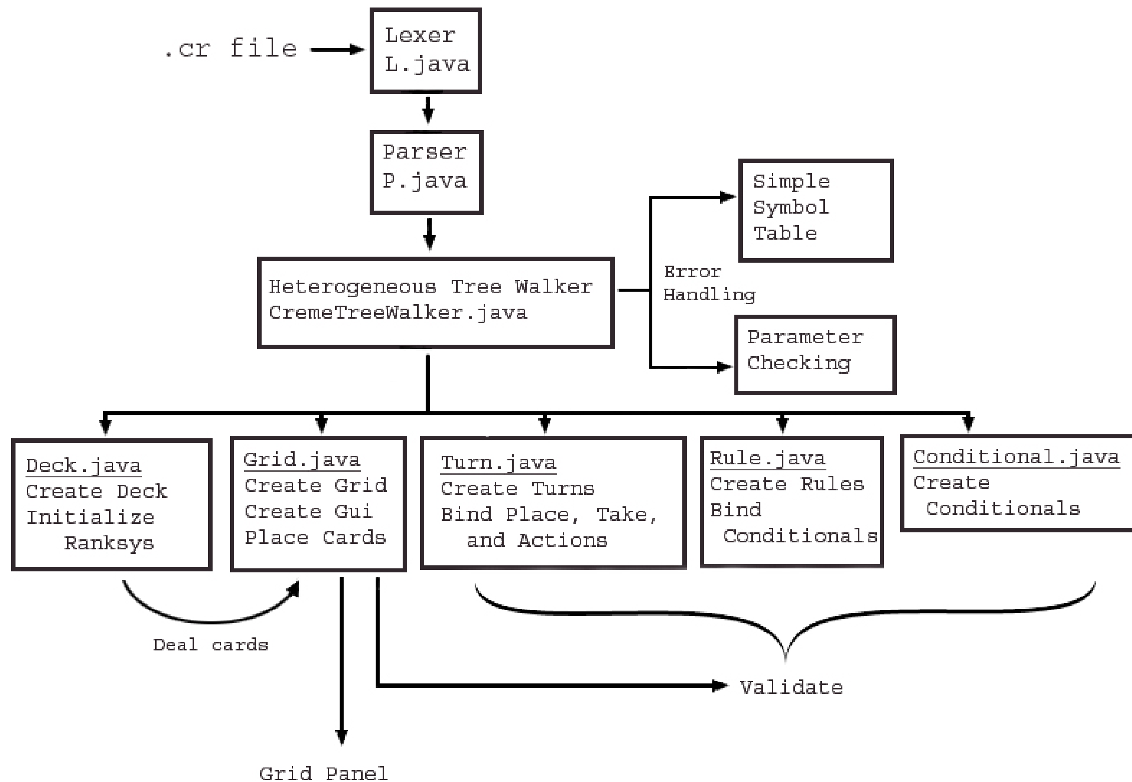
-**Spaces** are reserved for separating conditionals and their brackets and Grid rule instantiations and declarations.

Example:
```
Grid[1-15,2].rules ~ if !isEmpty then { place(+1,1,*); }
```
Optionally, spaces may be placed inside of brackets to separate numbers and wildcards [i.e. – `[1-15, 2]` (space included) is acceptable as well as `[1-15,2]` (space not included)]. For inline commenting, two spaces separate the code from the "`//`" starting the comment.

These conventions make it easy for any Crème programmer to look at a piece of code and instantly get a feel for what is going on it it.

5 ARCHITECTURAL DESIGN

In general, Crème is built like most other compilers. The main components are the Lexer, the Parser, the Tree Walker, Symbol Table, Parameter Checker and Code Generator. When the compiler is invoked, the Crème source code (a .cr file) is sent to the Lexer where it is stripped of white space and comments. This raw code is then sent to the Parser, where the different tokens are broken up. From there, the Tree Walker is given the tokenized Parser output and it walks the tree that results. It analyzes the code and creates a Symbol Table of the different labels defined in Ranksys. As it walks the tree, it checks referenced labels against the Symbol Table and returns errors and exits the program if they are not found. Also, the Tree Walker checks the different calls that are made from things like Grid spaces and Ranksys to make sure that they are valid. Again, if errors are encountered, an error message is printed and the compiler exits. As the Tree Walker goes through the tree, it calls the appropriate Java classes bypassing an intermediate code format. This is done after the static semantic checks have been made. From there, the Java code opens the GUI with all parsed information present (methods, variables, etc.).



6 TEST PLAN

The testing of Crème proved to be one of the most strenuous, yet satisfying, portions of developing the language. Instead of dealing with scripts and automatic testing suites, we decided that it was easier and more effective to create our own Java test file that we could continually pass to the compiler and check the output. This way, we did not waste time figuring out the scripts and starting a "mini-project" just to be able to test our code. Since we're all familiar with debugging normal Java programs, it made perfect sense to transfer that tendency to this project. The output that we checked was generated by a series of System.out.println() calls in the Java code that let us know where the compiler was in reading the methods. All of the individual Java classes were tested and as more functionality was added to the classes, they were tested in the compiler. This way, we knew that everything before the recent additions worked and that any errors were localized to the new parts of the code. Also, we ensured that the back end was working properly by ensuring that all calls that would be necessary in Solitaire (our main testing program) were in the Java test file. Once we knew that the back end was working properly, we set out to ensure that the compiler was interpreting the code properly. The biggest part of the testing process actually came after we successfully brought up the GUI and the game was semi-functional. At this point we spent most of the time constantly testing different aspects of the language to make sure that cards could not go where they were not supposed to, but went exactly to where they were supposed to be. A big help in finding out if the functionality was correct was the print statements that the Java code was spewing out as the game was played. This allowed us to find out what calls were being made on what objects and what was changing in the program. Essentially the statements gave us a firsthand look into what was going on in the compiler.

The following is the Java file that was used in error testing. It contains all of the necessary functions for Solitaire, which allowed us to ensure that the entire game was complete at the end:

```
import java.util.*;

public class Solitaire{

    public static void main(String[] args){


        Vector[] ranksysArray = new Vector[3];
        ranksysArray[0] = new Vector();
        ranksysArray[1] = new Vector();
        ranksysArray[2] = new Vector();

        ranksysArray[0].addElement("King>");
        ranksysArray[0].addElement("Queen>");
        ranksysArray[0].addElement("Jack>");
        ranksysArray[0].addElement("Ten>");
```

```
ranksysArray[0].addElement("Nine>");
ranksysArray[0].addElement("Eight>");
ranksysArray[0].addElement("Seven>");
ranksysArray[0].addElement("Six>");
ranksysArray[0].addElement("Five>");
ranksysArray[0].addElement("Four>");
ranksysArray[0].addElement("Three>");
ranksysArray[0].addElement("Two>");
ranksysArray[0].addElement("Ace.");
ranksysArray[1].addElement("SPADES.");
ranksysArray[1].addElement("CLUBS>");
ranksysArray[1].addElement("HEARTS.");
ranksysArray[1].addElement("DIAMONDS.");
ranksysArray[2].addElement("RED.");
ranksysArray[2].addElement("BLACK.");


Deck myDeck = new Deck(ranksysArray);
Grid myGrid=new Grid(7, 2);

int[] theInts=new int[1];
String[] theLabels=new String[1];
theInts[0]=2;
theLabels[0]="BLACK";
myDeck.darwinize(theInts, theLabels, 1, "SPADES", true);
myDeck.darwinize(theInts, theLabels, 1, "CLUBS", true);
theInts[0]=2;
theLabels[0]="RED";
myDeck.darwinize(theInts, theLabels, 1, "HEARTS", true);
myDeck.darwinize(theInts, theLabels, 1, "DIAMONDS", true);
myDeck.createCardStack();
myDeck.printDeck();

RuleGrid myTakeGrid = new RuleGrid(7, 2, myDeck);
RuleGrid myPlaceGrid = new RuleGrid(7,2, myDeck);
ActionGrid myActionGrid = new ActionGrid(7,2, myGrid);

Rule rule1 = new Rule(3);
rule1.setMinus(0, 1);  //descending
rule1.setMinus(1, 1);  //alternating black -> red
Rule rule2 = new Rule(3);
rule2.setMinus(0, 1);  //descending
rule2.setPlus(1, 1);  //alternating red -> black
myPlaceGrid.setRule(rule1, 0, 1);
myPlaceGrid.setRule(rule1, 1, 1);
myPlaceGrid.setRule(rule1, 2, 1);
myPlaceGrid.setRule(rule1, 3, 1);
myPlaceGrid.setRule(rule1, 4, 1);
myPlaceGrid.setRule(rule1, 5, 1);
myPlaceGrid.setRule(rule1, 6, 1);
myPlaceGrid.setRule(rule2, 0, 1);
myPlaceGrid.setRule(rule2, 1, 1);
myPlaceGrid.setRule(rule2, 2, 1);
myPlaceGrid.setRule(rule2, 3, 1);
myPlaceGrid.setRule(rule2, 4, 1);
myPlaceGrid.setRule(rule2, 5, 1);
myPlaceGrid.setRule(rule2, 6, 1);

//only kings can go into empty grid spots
IsTotallyEmpty cond0=new IsTotallyEmpty(myGrid, 0, 1);
IsTotallyEmpty cond1=new IsTotallyEmpty(myGrid, 1, 1);
IsTotallyEmpty cond2=new IsTotallyEmpty(myGrid, 2, 1);
IsTotallyEmpty cond3=new IsTotallyEmpty(myGrid, 3, 1);
IsTotallyEmpty cond4=new IsTotallyEmpty(myGrid, 4, 1);
IsTotallyEmpty cond5=new IsTotallyEmpty(myGrid, 5, 1);
IsTotallyEmpty cond6=new IsTotallyEmpty(myGrid, 6, 1);
Rule tEmptyRule0=new Rule(3, cond0);
Rule tEmptyRule1=new Rule(3, cond1);
Rule tEmptyRule2=new Rule(3, cond2);
Rule tEmptyRule3=new Rule(3, cond3);
Rule tEmptyRule4=new Rule(3, cond4);
```

```
Rule tEmptyRule5=new Rule(3, cond5);
Rule tEmptyRule6=new Rule(3, cond6);
tEmptyRule0.setExactValue(0, "King");
tEmptyRule1.setExactValue(0, "King");
tEmptyRule2.setExactValue(0, "King");
tEmptyRule3.setExactValue(0, "King");
tEmptyRule4.setExactValue(0, "King");
tEmptyRule5.setExactValue(0, "King");
tEmptyRule6.setExactValue(0, "King");
myPlaceGrid.setRule(tEmptyRule0, 0, 1);
myPlaceGrid.setRule(tEmptyRule1, 1, 1);
myPlaceGrid.setRule(tEmptyRule2, 2, 1);
myPlaceGrid.setRule(tEmptyRule3, 3, 1);
myPlaceGrid.setRule(tEmptyRule4, 4, 1);
myPlaceGrid.setRule(tEmptyRule5, 5, 1);
myPlaceGrid.setRule(tEmptyRule6, 6, 1);

//increasing, same suit, in the 4 top right grid boxes
//start with aces
Rule rule3=new Rule(3);
rule3.setPlus(0, 1);  //increasing
rule3.setMultipleMatch(1, 2);  //same suit
myPlaceGrid.setRule(rule3, 6, 0);
myPlaceGrid.setRule(rule3, 5, 0);
myPlaceGrid.setRule(rule3, 4, 0);
myPlaceGrid.setRule(rule3, 3, 0);
//can initially start with ace
IsTotallyEmpty acesCond6=new IsTotallyEmpty(myGrid, 6, 0);
IsTotallyEmpty acesCond5=new IsTotallyEmpty(myGrid, 5, 0);
IsTotallyEmpty acesCond4=new IsTotallyEmpty(myGrid, 4, 0);
IsTotallyEmpty acesCond3=new IsTotallyEmpty(myGrid, 3, 0);
Rule aces6=new Rule(3, acesCond6);
Rule aces5=new Rule(3, acesCond5);
Rule aces4=new Rule(3, acesCond4);
Rule aces3=new Rule(3, acesCond3);
aces6.setExactValue(0, "Ace");
aces5.setExactValue(0, "Ace");
aces4.setExactValue(0, "Ace");
aces3.setExactValue(0, "Ace");
myPlaceGrid.setRule(aces6, 6, 0);
myPlaceGrid.setRule(aces5, 5, 0);
myPlaceGrid.setRule(aces4, 4, 0);
myPlaceGrid.setRule(aces3, 3, 0);

//tell grid about the rule grids for take and place
//      myGrid.setTakeAndPlace(myTakeGrid, myPlaceGrid);

// Overturn a NOVIEW card on the top of the stack
IsEmpty emptyCond0=new IsEmpty(myGrid, 0, 1);
IsEmpty emptyCond1=new IsEmpty(myGrid, 1, 1);
IsEmpty emptyCond2=new IsEmpty(myGrid, 2, 1);
IsEmpty emptyCond3=new IsEmpty(myGrid, 3, 1);
IsEmpty emptyCond4=new IsEmpty(myGrid, 4, 1);
IsEmpty emptyCond5=new IsEmpty(myGrid, 5, 1);
IsEmpty emptyCond6=new IsEmpty(myGrid, 6, 1);
Action turnOver0 = new Action("MOVE", 0,1,1,false,0,1,1,0);
Action turnOver1 = new Action("MOVE", 1,1,1,false,1,1,1,0);
Action turnOver2 = new Action("MOVE", 2,1,1,false,2,1,1,0);
Action turnOver3 = new Action("MOVE", 3,1,1,false,3,1,1,0);
Action turnOver4 = new Action("MOVE", 4,1,1,false,4,1,1,0);
Action turnOver5 = new Action("MOVE", 5,1,1,false,5,1,1,0);
Action turnOver6 = new Action("MOVE", 6,1,1,false,6,1,1,0);
ActionConditional ac0=new ActionConditional(emptyCond0, turnOver0);
ActionConditional ac1=new ActionConditional(emptyCond1, turnOver1);
ActionConditional ac2=new ActionConditional(emptyCond2, turnOver2);
ActionConditional ac3=new ActionConditional(emptyCond3, turnOver3);
ActionConditional ac4=new ActionConditional(emptyCond4, turnOver4);
ActionConditional ac5=new ActionConditional(emptyCond5, turnOver5);
ActionConditional ac6=new ActionConditional(emptyCond6, turnOver6);
myActionGrid.setActionConditional(ac0, 0, 1);
myActionGrid.setActionConditional(ac1, 1, 1);
```

```
        myActionGrid.setActionConditional(ac2, 2, 1);
        myActionGrid.setActionConditional(ac3, 3, 1);
        myActionGrid.setActionConditional(ac4, 4, 1);
        myActionGrid.setActionConditional(ac5, 5, 1);
        myActionGrid.setActionConditional(ac6, 6, 1);

        //grid box (0, 0) "deal" out cards
        //action1 - if not empty, move 3 cards over
        //action2 - if empty, reverse (meaning, move cards back)
        NotIsTotallyEmpty topCond1=new NotIsTotallyEmpty(myGrid, 0, 0);
        IsTotallyEmpty topCond2=new IsTotallyEmpty(myGrid, 0, 0);
        Action move3 = new Action("MOVE", 0, 0, 3, false, 1, 0, 3, 0);
        Action moveAll = new Action("MOVESTAR", 1, 0, true, true, 0, 0,
false, true);
        ActionConditional topAC1=new ActionConditional(topCond1, move3);
        ActionConditional topAC2=new ActionConditional(topCond2, moveAll);
        myActionGrid.setActionConditional(topAC2, 0, 0);
        myActionGrid.setActionConditional(topAC1, 0, 0);

        //tell grid about action grid
        //      myGrid.setAction(myActionGrid);

        //tell grid about turns and their associated rule/action grids
        Rule kingRule=new Rule(3);
        kingRule.setExactValue(0, "King");
        Contains kingCond3=new Contains(myGrid, 3, 0, kingRule, myDeck);
        Contains kingCond4=new Contains(myGrid, 4, 0, kingRule, myDeck);
        AtDepth kingCond5=new AtDepth(myGrid, 5, 0, kingRule, 1, myDeck);
        AtDepth kingCond6=new AtDepth(myGrid, 6, 0, kingRule, 1, myDeck);
        Rule kingCondRule3=new Rule(3, kingCond3);
        Rule kingCondRule4=new Rule(3, kingCond4);
        Rule kingCondRule5=new Rule(3, kingCond5);
        Rule kingCondRule6=new Rule(3, kingCond6);
        Vector kingRuleVector=new Vector();
        kingRuleVector.add(kingCondRule3);
        kingRuleVector.add(kingCondRule4);
        kingRuleVector.add(kingCondRule5);
        kingRuleVector.add(kingCondRule6);
        Turn turn1=new Turn(kingRuleVector, myTakeGrid, myPlaceGrid,
myActionGrid);
        Vector turnVector=new Vector();
        turnVector.add(turn1);
        myGrid.setTurn(turnVector);

        myGrid.put(0,1, myDeck.deal(), true);

        myGrid.put(1,1, myDeck.deal(), true);
        myGrid.put(1,1, myDeck.deal(), false);

        myGrid.put(2,1, myDeck.deal(), true);
        myGrid.put(2,1, myDeck.deal(), false);
        myGrid.put(2,1, myDeck.deal(), false);

        myGrid.put(3,1, myDeck.deal(), true);
        myGrid.put(3,1, myDeck.deal(), false);
        myGrid.put(3,1, myDeck.deal(), false);
        myGrid.put(3,1, myDeck.deal(), false);

        myGrid.put(4,1, myDeck.deal(), true);
        myGrid.put(4,1, myDeck.deal(), false);
        myGrid.put(4,1, myDeck.deal(), false);
        myGrid.put(4,1, myDeck.deal(), false);
        myGrid.put(4,1, myDeck.deal(), false);

        myGrid.put(5,1, myDeck.deal(), true);
        myGrid.put(5,1, myDeck.deal(), false);
        myGrid.put(5,1, myDeck.deal(), false);
        myGrid.put(5,1, myDeck.deal(), false);
        myGrid.put(5,1, myDeck.deal(), false);
        myGrid.put(5,1, myDeck.deal(), false);
```

```
        myGrid.put(6,1, myDeck.deal(), true);
        myGrid.put(6,1, myDeck.deal(), false);
        myGrid.put(6,1, myDeck.deal(), false);
        myGrid.put(6,1, myDeck.deal(), false);
        myGrid.put(6,1, myDeck.deal(), false);
        myGrid.put(6,1, myDeck.deal(), false);
        myGrid.put(6,1, myDeck.deal(), false);

        myGrid.put(0,0, myDeck, false);

        myGrid.display();

    }
}
```

Finally, this is Freecell.cr, which is another game that we wrote in Crème:

```
//Freecell in Creme

Deck.Ranksys = 3;
Deck.Ranksys.1=(King>Queen>Jack>Ten>Nine>Eight>Seven
              >Six>Five>Four>Three>Two>Ace);
Deck.Ranksys.2=(SPADES.CLUBS>HEARTS.DIAMONDS);
Deck.Ranksys.3=(BLACK[2.(SPADES,CLUBS)].RED[2.(HEARTS,DIAMONDS)]);

Grid=[8,2];

Deck.deal ~ Grid[0-3, 1](7,0);
Deck.deal ~ Grid[4-7, 1](6,0);
Grid[0-7,0-1].setDepth(1);

turn.1(Grid[4-7,0].atDepth(1)[King,*,*]){

// If the bottom grids are not empty, only put descending alternating

Grid[0-7,1].rules ~ if !isTotallyEmpty then { place[(-1,+1,*)(-1,-
1,*)]; }

// If the top is empty, place anything

Grid[0-3,0].rules ~ if isTotallyEmpty then { place(*,*,*);
                     else { place(+1,*,*); }}

// if the top is not empty, place nothing
//Grid[0-3,0].rules ~ if !isTotallyEmpty then { place(1,1,1); }

// Place only Aces on the empty top right spots, and if they're
// not empty, only acsending cards of the same suit.
Grid[4-7,0].rules ~ if isTotallyEmpty then { place[Ace,*,*];
                     else { place(+1,1,*); }}
Grid[4-7,0].rules ~ if !isTotallyEmpty then { place(+1,1,*); }

}
```

7 LESSONS LEARNED

In doing this project (the first large scale group programming project most of our group has done) we've each come away with our own thoughts and comments about the project as a whole. We all learned a lot about teamwork and responsibility; since all of the pieces that we did individually had to come together collectively, it was imperative that we communicate with each other and help each other so no slipups would slow down the project. The following are real life testimonials by our team describing the trials and tribulations we each faced and the lessons that we are leaving this project glad to have learned.

Anthony Chan:

> One of the things I learned during the course of this project was that whenever a group consisting of more than one person is assigned to accomplish a task, there is an amazing tendency to find the most complex method imaginable to solve a particular task. Part of what we had to learn, was when to add things to the language, and when to start cutting out functionality that was clearly outside of the scope of what we had time to implement. There was a lot of time wasted in the beginning on neat features that never really came to fruition and in retrospect, it would probably have been more efficient to start the project off as simple as possible. I also learned that communication with teammates is very important for building a cohesive and productive development environment. I was very grateful that two of my teammates lived literally next door as it made having quick chats about problems or new ideas possible.

Cheryl Lau:

> I learned that a language doesn't have to be as complex as Java or C. A language can be small and specific with a directed purpose. Going into this project at the beginning of the semester, I was scared out of my mind about creating a language and a compiler. I thought to myself, I have to create an entire programming language and a compiler!! That's crazy! I didn't know where to begin. As the semester went on, the task became clearer and our goal became more visible. (Although the stress level never really dropped for me.) We took on the problem one step at a time. Again, I have come to realize that the way to handle a big project is to tackle it in pieces. First you have to clearly define your goal. Then you have to come up with what steps you should do next. Then, you have to take each piece at a time and build up the project little by little. When I actually got to the coding part and I found some direction, the project became fun. I would definitely like to thank my teammates for being so chill and calm. Thank you for putting up with me as a big ball of stress, me on no sleep, and me in general.

James Leslie:

> This project taught me many things about myself and my capabilities as a "team player." I learned that simple communication is the fastest way to get

things done when working with people. When the project is separated up into smaller components, it can be much easier to get things done while still focusing on the big picture. This is because everyone knows what role they play in the group and they don't have to keep asking for work or waste time by trying to figure something out that someone else did and didn't tell anyone about. As we progressed, we regularly kept each other up to date on what was going on with our respective portions of the assignment. And as things moved along, everything became easier, as we knew what had been done, what was being done, and what still needed to be done. With this knowledge, we could dynamically give pressing work to teammates who were working on less intensive things, to ensure that the group as a whole kept rolling on. I also realized that some "team projects" are undeserving of their negative connotation. As long as everyone does their job and does it on time, things go smoothly and problems can be avoided quickly. I was lucky to work with such a great team, and I'm immensely pleased with our results.

Joshua Mackler:

Developing Creme provided many opportunities to learn, especially about the process of design, and the resulting implementation of design in a group structure. Many brainstorming sessions resulted in the gathering of extensive information on design, which was helpful to a point, but ended up being harmful as well. Since the design extended began to exceed the necessary scope of our final language, cut-backs were made. As a result, the language never had strong structure until we examined test cases of card games and decided what we needed, and what we could do without. The end result was a robust program without unnecessary abilities, such as card hands, various players, multiple decks, or even the need to declare the name of the deck.

The group dynamic was informative as well, since it forced the four of us to coordinate and work together. I took it upon myself to make sure each person understood their part and how to relate it to the project, as well as making flexible deadlines. One important note is the fact that, to work as a group, each person needed to know "just enough" information about what each other person was doing so that person could perform his own job well. This forced each person to work closely with every other. Each person was able to work on the part of the project they were most suited for, and most willing to do, which enabled for better productivity, and a better final product.

8 APPENDIX

8.1 Crème Grammar

Run → *StartRule*+ EOF!

StartRule → *DeckCall* | *Turn* |'**Grid**' (*ObjectCall* | '**=**' '**[**' *Digit* '**,**' *Digit* '**]**' '**;**')

DeckCall → '**Deck**' '**.**' (*RankSys* | *Deal*)

Deal → '**deal**' '**~**' '**Grid**' *Grid* *Vis* '**;**'

ObjectCall → (*Grid* '**.**' (*Move* | *Depth*))

Depth → '**setDepth**' '**(**' *Digit* '**)**' '**;**'

Move → '**move**' *Vis* '**~**' '**Grid**' *Grid* *Vis* '**;**'

Turn → '**turn**' '**.**' *TurnCond* '**[**' *Conditional** '**]**'

TurnCond → *Digit* '**(**' ('**Grid**' *Grid* *BooleanCheck* ('**,**')?)* '**)**'

Grid → '**[**' *Digit* ('**-**' *Digit*)? '**,**' *Digit* ('**-**' *Digit*)? '**]**'

Rules → ('**place**' |'**take**') (*Card* | '**(**' *CardRule* '**)**'
| '**[**' ('**(**' *CardRule* '**)**' | *Card*)+ '**]**')) '**;**'

Conditional → '**if**' *BooleanCheck* '**then**' '{' (*Conditional* | *Action*) '**else**' '**{**' (*Conditional* | *Action*) '**}**')? '**}**'
| '**Grid**' *Grid* '**.**' '**rules**' '**~**' *Conditional*
| *Rules*

BooleanCheck → ('**!**' |'**.**')? ('**isEmpty**' | '**isTotallyEmpty**' | '**contains**' *Card* | ('**atDepth**' |'**upToDepth**') '**(**' *Digit* '**)**' *Card*)

Action → '**action**' '**(**' *TheAction** '**)**' '**;**'

TheAction → '**Grid**' *Grid* '**.**' (*Move* | ('**shuffle**' | '**setDepth**' *Digit* |'**reverse**' (*Digit* |'*****') '**,**' (*Digit* |'*****')) '**;**')

CardRule → *RuleSpec* ('**,**' *RuleSpec*)*

RuleSpec → (('**-**' |'**+**')? *Digit* ('**=**')?) |'*****'

```
Vis →  '('  (Digit | '*')  ','  (Digit | '*')  ')'

RankSys →  'Ranksys' ('.'  Digit  '='  InRank |  '='  Digit  ';')

InRank →  '('  DefRank (  ('>'  |  '.'  ) DefRank) *  ')'  ';'

DefRank →   Id  ('^'  Digit) ?  ('['  Map ( ','  Map) *  ']') ?

Map →  Digit  ('.' | '!')  '('  Id  (','  Id) *  ')'

Seq →  ('seq' | 'cseq')  '^'  Digit

Card →  '['  (Id  ('=') ?  | '*') (',' (Id  ('=') ?  | '*')  ) *  ']'

Digit  →  ('0'..'9') +

Id  →  (  'a'..'z' | 'A'..'Z')
(  'a'..'z' | '_' | 'A'..'Z' | '0'..'9') *
```

8.2 Crème Java Code

8.2.1 Deck.java

```
Deck.java
import java.util.*;
import java.awt.Color;

/* LOG
 *================================
 * Last Modified: 5.04.03 by Ant.
 *
 * 5.05 - Made shuffle call mandatory upon first deal.
 *
 * 5.04 - Merged Destroy and Preserve into Darwinize.
 *
 * 4.29 - Implemented preserve().
 *
 * 4.27 - Implemented destroy(), printDeck(), createCardVector(), createStackVector()
 *        cardsVector is no longer a vector of String[] labels, but a vector of Cards.
 *
 * 4.24 - Implemented compare(), parseRank(), printRanks(), rankOf()
 *        Ranksys structure complete with ranking and comparitive abilities.
 *
 * 4.17 - Implemented Shuffle() to randomize card ordering.
 */

/* Grid Methods
 *
 *    private String parseRank(String theLabel, int theDimension)
 *
 *    public void printRanks()
 *
 *    public void printDeck()
 *
 *    public void createCardVector(Vector theLabelVector)
 *
 *    public void createCardStack()
```

```
 *
 *    public int compare(Card card1, Card card2)
 *
 *    public int rankof(String label, int dimension)
 *
 *    public void darwinize(int[] dSpecifier, String[] lSpecifier, int dLock, String
lLock, boolean preserve)
 *
 *    private Vector initiateMap(Vector[] ranksys)
 *
 *    private Vector map(Vector v1, Vector v2, int labelIndex)
 *
 *    public Card deal(){
 *
 *    public void shuffle(){
 */

/**
 * Deck is a class that describes a deck of cards. No creme program or game can run
without
 * a deck.  This class takes in the labels of the cards gathered from the Ranksys
statements
 * and creates cards by mapping the first dimension onto each successive dimension. Ranks
are
 * then parsed and assigned to each label according to the dimension. Deck is also home
to the
 * Darwinizing engine, a method we use to destroy or preserve the cards specified.
 */

public class Deck{

    CardStack theDeck = new CardStack();
    Vector cardsVector = new Vector();
    int numCards = 0;
    int numDimensions=0;
    boolean toShuffle = true;
    boolean colorize = true;
    int colorIndex = -1;

    // Used to keep track of the current rank number in each dimension.
    int rankNum = 0;
    Vector[] rankedArray;

    private String parseRank(String theLabel, int theDimension){
        String parsedString;

        if(theLabel.endsWith(">")){
            parsedString = theLabel.substring(0, theLabel.length()-1);
            rankedArray[theDimension].add(new RankElement(parsedString, rankNum));
            rankNum++;
        }
        else if(theLabel.endsWith(".")){
            parsedString = theLabel.substring(0, theLabel.length()-1);
            rankedArray[theDimension].add(new RankElement(parsedString, rankNum));
        }
        else{
            parsedString = theLabel;
            rankedArray[theDimension].add(new RankElement(theLabel, rankNum));
        }
        return parsedString;
    }

    public void printRanks(){
        for(int i=0; i<rankedArray.length; i++){
            System.out.println("\nRanksys."+(i+1));
            for(int j=0; j<rankedArray[i].size(); j++){
                RankElement rElement = (RankElement)rankedArray[i].elementAt(j);
                    System.out.println(rElement.rank+"  "+rElement.label);
            }
        }
    }
```

```
    public void printDeck(){
        for(int i=0; i<cardsVector.size(); i++){
            System.out.println("Card#"+(i+1)+" -> "+(Card)cardsVector.elementAt(i));
        }
    }

    public void createCardVector(Vector theLabelVector){
        cardsVector.clear();
        for(int i=0; i<theLabelVector.size(); i++){
            Card theCard = new Card((String[])theLabelVector.elementAt(i));
            cardsVector.add(theCard);
        }
    }

    public void createCardStack(){
        theDeck.makeEmpty();
        for(int i=0; i<cardsVector.size(); i++){
            theDeck.push((Card)cardsVector.elementAt(i));
        }
    }

    public int compare(Card card1, Card card2){
        boolean distinct = false;
        int result = 0;
        int i=0;
        while( !distinct && (i<(card1.rankLabels).length) ){
            // If card1 is ranked more than card2, return 1
            if( rankof(card1.rankLabels[i], i)<rankof(card2.rankLabels[i], i) ){
                distinct=true;
                result = 1;
            }
            // If card1 is ranked less than card2, return -1
            else if( rankof(card1.rankLabels[i], i)>rankof(card2.rankLabels[i], i) ){
                distinct=true;
                result = -1;
            }
            // No distinction yet, proceed to next dimension for comparison
            else if( rankof(card1.rankLabels[i], i)==rankof(card2.rankLabels[i], i) ){
                i++;
            }
        }
        return result;
    }

    public int rankof(String label, int dimension){
        for(int i=0; i<=rankedArray[dimension].size(); i++){
            RankElement rElement = (RankElement)rankedArray[dimension].elementAt(i);
            if( label.equals(rElement.label) )
                return rElement.rank;
        }
        // If static semantics did not save the day.
        System.out.println("VERY BAD ERROR: Card Label ["+label+"] not found in RankSys
dimension "+dimension+"!!!!");
        return -1;
    }


    // Usage: Pass two arrays, first with dimension numbers, second with labels for that
dimension
    //        specifying cards we want to keep. dLock and lLock specify the dimension and
label
    //        at which these exclusions take place. Set PRESERVE = True to preserve the
    //        specified cards. Set PRESERVE = False to destroy the same cards.
    // NOTE: After destroying or preserving a card, the cardsVector and the stack of
cards, theDeck will
    // be out of sync. For performance issues, it is imperative that a call to
    // createCardStack is made after all the destroy calls.
    public void darwinize(int[] dSpecifier, String[] lSpecifier, int dLock, String lLock,
boolean preserve){
        if( preserve ){
```

```
            System.out.println("Darwinizer invoked in Preservation mode");
        }
        else{
            System.out.println("Darwinizer invoked in Destroy mode");
        }
        boolean match;
        for( int i=0; i<cardsVector.size(); i++){
            match = true;
            Card currentCard = (Card)cardsVector.elementAt(i);
            if( lLock.equals((String)(currentCard.rankLabels[dLock])) ){
                for( int j=0; (j<dSpecifier.length&&match); j++){
                    int indexCheck = (int)dSpecifier[j];

if( ((currentCard.rankLabels[indexCheck]).equals((String)lSpecifier[j]))){
                        match=!preserve;
                    }
                    // Else, just exit the loop.
                    else{
                        match=preserve;
                    }
                }
                if( match ){
                    System.out.println("Removing card " + cardsVector.elementAt(i));
                    cardsVector.remove(i);
                    i--;
                }
            }
        }
    }

    private Vector initiateMap(Vector[] ranksys){
        numDimensions = ranksys.length;
        rankedArray = new Vector[numDimensions];
        rankedArray[0] = new Vector();
        Vector cardVector=new Vector();
        //initialize to first dimension
        for (int i=0; i<ranksys[0].size(); i++)
            {
                String[] cardLabel= new String[numDimensions];
                //Parse the Rank data, assign a numRank to label for the particular
dimension
                //Then return the string stripped of rank data.
                String parsedLabel = parseRank((String)ranksys[0].elementAt(i), 0);
                cardLabel[0]=parsedLabel;
                cardVector.add(i, cardLabel);

            }

        //iterate through the dimensions, adding on one dim at a time
        for(int i=1;i<numDimensions;i++)
            {
                colorize=true;
                rankedArray[i] = new Vector();
                cardVector=map(cardVector, ranksys[i], i);
            }


        System.out.println("Initial Mapping complete. "+cardVector.size()+
                                " Cards created, "+numDimensions+" dimensions each.");
        return cardVector;
    }

    //v1 cardVector, vector of arrays
    //v2 next dimension to add, vector of strings
    private Vector map(Vector v1, Vector v2, int labelIndex)
    {
        numCards=0;
        rankNum=0;
        Vector newCardVector=new Vector();
        for(int i=0;i<v1.size();i++)
            {
```

```
            for(int j=0;j<v2.size();j++)
                {
                    String[] cardLabel = new String[numDimensions];

                    for(int k=0; k<labelIndex; k++){
                        cardLabel[k] = ((String[])v1.elementAt(i))[k];
                    }
                    String parsedLabel;
                    if(i==0)
                        parsedLabel = parseRank((String)v2.elementAt(j), labelIndex);

                    else
                        parsedLabel = ((String)v2.elementAt(j)).substring(0,

    ((String)v2.elementAt(j)).length()-1);

                    cardLabel[labelIndex]=parsedLabel;
                    newCardVector.addElement(cardLabel);
                    numCards++;

                }
            }
        return newCardVector;
    }

    public Deck(Vector[] ranksys){

        Vector labelVector = initiateMap(ranksys);
        createCardVector(labelVector);
        createCardStack();

        printRanks();
    }

    public Card deal(){
        if( toShuffle ){
            shuffle();
            toShuffle = false;
        }
        if( !theDeck.isEmpty() )
            return theDeck.topAndPop();
        else{
            return null;
        }
    }

    //shuffle the stack of cards
    public void shuffle()
    {
        Vector v=new Vector(); //to temporarily hold cards

        //put cards in vector
        while(!theDeck.isEmpty())
            {
                Card c=theDeck.topAndPop();
                v.add(c);  //add card to vector
            }

        //put cards back into deck
        while(!v.isEmpty())
            {
                int rand=(int)(Math.random()*v.size());

                Card c=(Card)v.remove(rand);  //randomly pick card from vector
                theDeck.push(c);  //push card back in deck
            }

    }

}
```

8.2.2 CardStack.java

CardStack.java

```
import java.util.Vector;
/* LOG
 *================================
 * Last Modified: 4.17.03 by Ant.
 *
 * 4.20 Changed standard linked list implementation
 *      to Vector based implementation with vector
 *      properties. Basically a "transparent stack"
 *
 * 4.17 Copied stack code from Weiss data structures book
 *      adapted for a CardStack.
 *
 */

/** AtDepth Methods
    // StackVector class
    //
    // CONSTRUCTION: with no initializer
    //
    // ******************PUBLIC OPERATIONS*********************
    // void push( x )         --> Insert x
    // void pop( )            --> Remove most recently inserted item
    // Object top( )          --> Return most recently inserted item
    // Object topAndPop( )    --> Return and remove most recent item
    // boolean isEmpty( )     --> Return true if empty; else false
    // boolean isFull( )      --> Always returns false
    // void makeEmpty( )      --> Remove all items
    // ******************ERRORS********************************
    // pop on empty stack
*/
    /** Adapted from:
     * List-based implementation of the stack.
     * @author Mark Allen Weiss
     */
    public class CardStack
    {
        /**
         * Construct the stack.
         */
        public CardStack( )
        {
           cardStackVector = new Vector();
        }

        /**
         * Test if the stack is logically full.
         * @return false always, in this implementation.
         */
        public boolean isFull( )
        {
            return false;
        }

        /**
         * Test if the stack is logically empty.
         * @return true if empty, false otherwise.
         */
        public boolean isEmpty( )
        {
            return cardStackVector.isEmpty();
        }

        /**
         * Make the stack logically empty.
         */
        public void makeEmpty( )
```

```java
    {
        cardStackVector.clear();
    }

    /**
     * Get the most recently inserted item in the stack.
     * Does not alter the stack.
     * @return the most recently inserted item in the stack, or null, if empty.
     */
    public Card top( )
    {
        if( isEmpty( ) )
            return null;
        return ((Card)cardStackVector.lastElement());
    }

    /**
     * Remove the most recently inserted item from the stack.
     * @exception Underflow if the stack is empty.
     */
    public void pop( ) throws Underflow
    {
        if( isEmpty( ) )
            throw new Underflow( );
        cardStackVector.removeElementAt(cardStackVector.size()-1);
    }

    /**
     * Return and remove the most recently inserted item from the stack.
     * @return the most recently inserted item in the stack, or null, if empty.
     */
    public Card topAndPop( )
    {
        if( isEmpty( ) )
            return null;

        Card topItem = (Card)cardStackVector.lastElement();
       cardStackVector.removeElementAt(cardStackVector.size()-1);
        return topItem;
    }

    /**
     * Insert a new item into the stack.
     * @param x the item to insert.
     */
    public void push( Card x )
    {
        cardStackVector.addElement(x);
    }

           /**
            * Get element at underlying vector from stack.
            * @param x the index of the item to get.
            */
           public Card elementAt(int x)
           {
             return ((Card)cardStackVector.elementAt(x));
           }

           /**
            * Return the size of the underlying vector.
            */
           public int size()
           {
               return cardStackVector.size();
           }

private CardNode topOfStack;
    private Vector cardStackVector;
}
```

8.2.3 Card.java

```
Card.java
import java.util.*;
/* LOG
 *================================
 * Last Modified: 4.16.03 by Ant
 *
 * 4.16 Created the class
 */

/*
Card
-------
A basic card object.

Rules in Conditionals can only be of the [] case because Contains, AtDepth,
 and UpToDepth only test an exact match of one card.
 */
public class Card
{

    // Size of the rankLabels array denotes the number of dimensions,
    // or how many labels are needed to uniquely identify the card.
    public String[] rankLabels;
    public boolean VISIBLE = true;


    public Card(String[] labels){

        rankLabels = labels;

    }

    public String toString(){
        String theString = new String();
        for(int i=0; i<rankLabels.length; i++){
            theString = theString+ " ["+rankLabels[i]+"]";
        }
        return theString;
    }
}
```

8.2.4 RankElement.java

```
RankElement.java
import java.util.*;

public class RankElement{

    // A class to store labels and thier corresponding ranks.
    // Each rankElement is stored in the rankedArray in the Deck class.
    // rankedArray is a 2D array where each dimension corresponds to
    // the ranksys dimension and replaces ">" and "." information
    // with rank integers.
    public String label;
    public int rank;


    public RankElement(String theLabel, int theRank){

        label = theLabel;
        rank = theRank;

    }

}
```

8.2.5 Grid.java

```
Grid.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

/* LOG
 *================================
 * Last Modified: 5.06.03 by Ant.
 *
 * 5.07 - Reverse() fully tested. Implemented setDepth() and shuffle(). Added
 *        put() for Stacks instead of just cards.
 *
 * 5.06 - Made card sizes smaller. Made font proportional to the card size.
 *        Enabled detection of Action events. Implemented ACTIONS
 *
 * 5.05 - Implemented Colorizing the last dimension, and linking to Rules via the
 *        isValid() function. Implemented recursive rule checking. Included
 *        josh's gridCheck function.
 *
 * 5.03 - Tested move, implemented basic mouse listeners, drawing of the
 *        grid with all card stacks, and gui movement of cards.
 *
 * 4.27 - Implemented  private void drawStackAt(int x, int y, Graphics page);
 *        in class GridPanel
 *
 * 4.23 - Implemented Move().
 */

    /* Grid Methods
     *
     * public Grid(int x, int y)
     *   Constructor where x and y specify the number of cells in the
     *   x and y direction. These are multiplied by the cellWidth and
     *   the cellHeight to get the pixel size of the Grid.
     *
     * public void display()
     *   Creates a new GridPanel (an extension of the JPanel class) and displays all the
     *   visible cards on the screen. Call this everytime you want to update the screen.
     *
     * public void put(Card theCard, int x, int y)
     *   Given a card, and a position on the grid, this method adds the card to the
     *   cardStack at that grid spot.
     *
     * public void put(int destX, int destY, Deck theDeck, boolean visible)
     *   Given a deck, and a position on the grid, this method adds the card to the
     *   cardStack at that grid spot.
     *
     * public Card get(int x, int y)
     *   Given a position on the grid, this method will pop the top card on the stack
     *   at that location and return it.
     *
     * public Card peek(int x, int y)
     *   Given a position on the grid, this method will return the top card on the stack
     *   at that location but not pop it from the stack.
     *
     * public void move(int srcX, int srcY, int srcNum, boolean srcVIEW, int destX, int
destY, int destVIEW, int destNOVIEW)
     *   Moves a specified number of cards from a grid location either from ALLVIEW or
NOVIEW to any other grid spot
     *   into any view.
     *
     * private void transferVisible(CardStack sourceStack, CardStack destStack);
     *   Private method used by move to take all visible cards and put them into a
tempStack
     *
     * public void setTakeAndPlace(RuleGrid tR, RuleGrid pR);
```

```
     *   Set the Take and Place rule objects to enable rule checking.
     *
     * public void gridCheck(int dim, int number);
     *   Josh's method to check if a number is greater than the width or length.
     *
     */

public class Grid implements MouseListener{

    CardStack theGrid[][];
    CardStack tempStack = new CardStack();
    int width;
    int height;
    int cellWidth=120;
    int cellHeight=300;
    int buffer=20;
    int cardDspSize= 10;
    // cardDspSize must ALWAYS divide into the cellHeight with no remainder
    // or else trouble ensues.
    JFrame frame = new JFrame("GRID");

    int liftCardX;
    int liftCardY;
    int liftCardIndex;
    boolean cardLifted;
    int dropCardX;
    int dropCardY;
    int dropCardIndex;
    CardStack recursiveStack = new CardStack();
    int undoPtr;
    boolean performUndo = false;

    RuleGrid takeRules;
    RuleGrid placeRules;
    ActionGrid  conditionalActions;
    Vector turnVector;
    int currentTurnIndex;
    Turn currentTurn;
    boolean gameover=false;
    int ruleDepth[][];

    GridPanel theGridPanel;

    public Grid(int x, int y){

        width = x;
        height= y;
        theGrid = new CardStack[width][height];
        ruleDepth = new int[width][height];

        for(int i=0; i<width; i++){
            for(int j=0; j<height; j++){
                theGrid[i][j]= new CardStack();
                ruleDepth[i][j]=-1;
            }
        }

       theGridPanel = new GridPanel(theGrid, width, height, cellWidth, cellHeight, buffer,
cardDspSize);
        theGridPanel.addMouseListener(this);
        frame.getContentPane().add(theGridPanel, BorderLayout.CENTER);
       frame.setSize(((width)*cellWidth)+cellWidth/2,  ((height)*cellHeight)+cellHeight/2);
        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
        frame.setVisible(true);

    }
```

```
    public void display(){
        frame.repaint();
    }

    public void put(int destX, int destY, Card theCard, boolean visible){
        if( theCard!=null ){
            if( visible ){
                theCard.VISIBLE=visible;
                theGrid[destX][destY].push(theCard);
            }
            else{
                transferVisible(theGrid[destX][destY], tempStack);
                theCard.VISIBLE=visible;
                theGrid[destX][destY].push(theCard);
                transferVisible(tempStack, theGrid[destX][destY]);
            }
        }
    }

    public void put(int destX, int destY, Deck theDeck, boolean visible){
        Card theCard = theDeck.deal();
        while( theCard != null ){
            if( visible ){
                theCard.VISIBLE=visible;
                theGrid[destX][destY].push(theCard);
            }
            else{
                transferVisible(theGrid[destX][destY], tempStack);
                theCard.VISIBLE=visible;
                theGrid[destX][destY].push(theCard);
                transferVisible(tempStack, theGrid[destX][destY]);
            }
        }
    }


    public Card get(int x, int y){
        return theGrid[x][y].topAndPop();
    }


    public Card peek(int x, int y){
        return theGrid[x][y].top();
    }

    public void move(int srcX, int srcY, int srcNum, boolean srcVIEW, int destX, int
destY, int destVIEW, int destNOVIEW){
        if( srcNum != destVIEW+destNOVIEW ){
            System.out.println("Grid.move() ERROR: The number cards being moved is not
consistent!");
        }
        else if( theGrid[srcX][srcY].isEmpty() ){
            System.out.println("Grid.move() ERROR: No cards present at the specified
location!");
        }
        else{
            // Moving VISIBLE source cards
            if( srcVIEW ){
                // There are no VISIBLE cards available
                if( !(theGrid[srcX][srcY].top()).VISIBLE ){
                    System.out.println("Grid.move() ERROR: No VISIBLE cards to move");
                }
                else{
                    int i=0;
                    // While, we're not done moving the number of VISIBLE DESTINATION
CARDS,
                    // or the top card is still visible, or the stack is not yet empty(no
not visible on stack).
                    // Keep moving.
                    while(i<destVIEW && (theGrid[srcX][srcY].top()).VISIBLE
                          && !((theGrid[srcX][srcY]).isEmpty()) ){
```

```
                        (theGrid[destX][destY]).push(theGrid[srcX][srcY].topAndPop());
                        i++;
                    }
                    // While, we're not done moving the number of NOT VISIBLE Dest. Cards,
                    // or there are still VISIBLE cards in the source stack.
                    // Keep moving.
                    i=0;
                    transferVisible(theGrid[destX][destY], tempStack);
                    while(i<destNOVIEW && (theGrid[srcX][srcY].top()).VISIBLE
                          && !((theGrid[srcX][srcY]).isEmpty()) ){
                        (theGrid[destX][destY]).push(((theGrid[srcX][srcY]).topAndPop()));
                        i++;
                        ((theGrid[destX][destY]).top()).VISIBLE=false;
                    }
                    transferVisible(tempStack, theGrid[destX][destY]);
                }
            }
            // Moving NOT VISIBLE source cards
            else{
                transferVisible(theGrid[srcX][srcY], tempStack);
                // There are no NOT VISIBLE cards available
                if( theGrid[srcX][srcY].isEmpty() ){
                    System.out.println("Grid.move() ERROR: No NOT VISIBLE cards to move");
                }
                else{
                    int i=0;
                    // While, we're not done moving the number of VISIBLE DESTINATION
CARDS,
                    //  or the stack is not yet empty.
                    // Keep moving.
                    while(i<destVIEW && !(theGrid[srcX][srcY].isEmpty())){
                        (theGrid[destX][destY]).push((theGrid[srcX][srcY]).topAndPop());
                        ((theGrid[destX][destY]).top()).VISIBLE=true;
                        i++;
                    }              // While, we're not done moving the number of NOT
VISIBLE Dest. Cards,
                    // or the stack is not yet empty.
                    // Keep moving.
                    i=0;
                    CardStack destTempStack = new CardStack();
                    transferVisible(theGrid[destX][destY], destTempStack);
                    while(i<destNOVIEW && !(theGrid[srcX][srcY].isEmpty())){
                        (theGrid[destX][destY]).push((theGrid[srcX][srcY]).topAndPop());
                        i++;
                    }
                    transferVisible(destTempStack, theGrid[destX][destY]);
                }
                // Done transfering NOVISIBLE source to all destinations
                // Return Visible source cards
                transferVisible(tempStack, theGrid[srcX][srcY]);
            }
        }
    }

    public void move(int srcX, int srcY, boolean srcVIEW, boolean srcNOVIEW, int dstX,
int dstY, boolean dstVIEW, boolean dstNOVIEW){

        // transfer all of the cards from one stack to another, preserving visibility
        if( srcVIEW && srcNOVIEW && dstVIEW && dstNOVIEW ){
            CardStack tempStack1 = new CardStack();
            CardStack tempStack2 = new CardStack();
            transferVisible( theGrid[srcX][srcY], tempStack1 );
            transferVisible( theGrid[dstX][dstY], tempStack2);
            while(!theGrid[srcX][srcY].isEmpty()){
                theGrid[dstX][dstY].push(theGrid[srcX][srcY].topAndPop());
            }
            transferVisible( tempStack2, theGrid[dstX][dstY] );
            transferVisible( tempStack1, theGrid[dstX][dstY] );
        }
        // transfer all of the cards from one stack to the dest making all visible
        else if( srcVIEW && srcNOVIEW && dstVIEW && !dstNOVIEW){
```

```java
            CardStack tempStack1 = new CardStack();
            while( !theGrid[srcX][srcY].isEmpty() ){
                tempStack1.push(theGrid[srcX][srcY].topAndPop());
            }
            while( !tempStack1.isEmpty() ){
                theGrid[dstX][dstY].push(tempStack1.topAndPop());
                theGrid[dstX][dstY].top().VISIBLE=true;
            }
        }
        // transfer all of the cards from one stack to the dest making all not visible
        else if( srcVIEW && srcNOVIEW && !dstVIEW && dstNOVIEW){
            CardStack tempStack1 = new CardStack();
            CardStack tempStack2 = new CardStack();
            transferVisible( theGrid[dstX][dstY], tempStack2);
            while( !theGrid[srcX][srcY].isEmpty() ){
                tempStack1.push(theGrid[srcX][srcY].topAndPop());
            }
            while( !tempStack1.isEmpty() ){
                theGrid[dstX][dstY].push(tempStack1.topAndPop());
                theGrid[dstX][dstY].top().VISIBLE=false;
            }
            transferVisible( tempStack2, theGrid[dstX][dstY] );
        }
        else{
            System.out.println("MOVESTAR called with invalid parameters: "
                        +srcVIEW+" "+srcNOVIEW+" "+dstVIEW+" "+dstNOVIEW+
                        " from Grid["+srcX+","+srcY+"] to Grid["+dstX+","+dstY+"]");
        }

    }

    private void transferVisible(CardStack sourceStack, CardStack destStack){

        // Transfering all the visible cards
        while( !sourceStack.isEmpty() && (sourceStack.top()).VISIBLE ){
            destStack.push( sourceStack.topAndPop() );
        }
    }

    public void setTakeAndPlace(RuleGrid tR, RuleGrid pR){
        takeRules = tR;
        placeRules = pR;
    }

    //must have at least one element in Vector
    public void setTurn(Vector tV)
    {
        turnVector = tV;

        currentTurnIndex=0;

        //get first turn's grids
        currentTurn=(Turn)turnVector.elementAt(0);
        takeRules = currentTurn.takeGrid;
        placeRules = currentTurn.placeGrid;
        conditionalActions = currentTurn.actionGrid;
    }

    public void setAction(ActionGrid aC){
        conditionalActions = aC;
    }

    public void gridCheck(int dim, int number){
         if (dim == 0){
            if (number >= width || number < 0){
                System.err.println("Grid parameter " + number +
                                 " not within the range of specified width " + width);
                System.exit(0);
            }}
        else{if (dim == 1){
            if (number >= height || number < 0){
```

```
                System.err.println("Grid parameter " + number +
                                   " not within the range of specified length " + height);
                System.exit(0);
            }
        }

        }
    }

    public void reverse(int x, int y, int VIEW, int NOVIEW){
        tempStack = new CardStack();
        if( VIEW < 0 ){
            transferVisible(theGrid[x][y], tempStack );
            for( int i=0; i<tempStack.size(); i++){
                theGrid[x][y].push((Card)tempStack.elementAt(i));
            }
        }
        if( NOVIEW < 0 ){
            transferVisible(theGrid[x][y], tempStack );
            CardStack newStack = new CardStack();
            while(!theGrid[x][y].isEmpty()){
                newStack.push( theGrid[x][y].topAndPop() );
            }
            for( int i=0; i<newStack.size(); i++){
                theGrid[x][y].push((Card)newStack.elementAt(i));
            }
            transferVisible(tempStack, theGrid[x][y]);
        }
        theGridPanel.repaint();
    }

    public void shuffle(int x, int y){
        Vector v=new Vector();
        while(!theGrid[x][y].isEmpty() ){
            v.add(theGrid[x][y].topAndPop());
        }
        while(!v.isEmpty())
            {
                int rand=(int)(Math.random()*v.size());
                Card c=(Card)v.remove(rand);
                theGrid[x][y].push(c);
            }
    }

    public void setDepth(int x, int y, int theDepth){
        ruleDepth[x][y] = theDepth;
    }

    /*
     *
     *  MOUSE LISTENER IMPLEMENTATION
     *
     */

    public void mousePressed(MouseEvent e){
        //outputConsole("Mouse pressed (# of clicks: "+e.getClickCount()+")", e);
        if( detectLiftCard(e.getX(), e.getY())&& !gameover ){
            int allowedIndex;
            if( ruleDepth[liftCardX][liftCardY]==-1)
                allowedIndex=0;
            else
                allowedIndex=(theGrid[liftCardX][liftCardY].size()-
ruleDepth[liftCardX][liftCardY]);

            recursiveStack = new CardStack();
            if( (liftCardIndex < theGrid[liftCardX][liftCardY].size() &&
                 liftCardIndex>=allowedIndex)
                && (theGrid[liftCardX][liftCardY].elementAt(liftCardIndex)).VISIBLE ){

                for(int i=theGrid[liftCardX][liftCardY].size()-1; i>=liftCardIndex; i--){
                    Vector oneCard = new Vector();
```

```
                    oneCard.addElement((Card)theGrid[liftCardX][liftCardY].elementAt(i));

                    if( takeRules.isValid(oneCard, liftCardX, liftCardY) ){
                        cardLifted = true;
                        System.out.println("Pushing "+theGrid[liftCardX][liftCardY].top()+"
to the RecursiveStack");
                        recursiveStack.push(theGrid[liftCardX][liftCardY].topAndPop());
                    }
                    else{
                        cardLifted = false;
                        i=liftCardIndex-1;
                    }
                }
                if( cardLifted )
                    theGridPanel.highlightCardAt(e.getX(), e.getY());
                else{
                    while( !recursiveStack.isEmpty() ){
                        theGrid[liftCardX][liftCardY].push( recursiveStack.topAndPop() );
                    }
                }
            }
            else
                cardLifted = false;
        }
        else
            cardLifted = false;
    }

    public void mouseReleased(MouseEvent e){
        //outputConsole("Mouse released (# of clicks: "+e.getClickCount()+")", e);
        // Differentiate between PLACE and ACTION by detecting the lift and drop stacks.
        detectDropStack(e.getX(), e.getY());
        if( !gameover){
            if( liftCardX==dropCardX && liftCardY==dropCardY ){
                // DO ACTION
                System.out.println("PERFORMING AN ACTION at
Grid["+liftCardX+","+liftCardY+
                                "]. Returning cards from Recursive Stack");
                while(!recursiveStack.isEmpty() ){
                    theGrid[liftCardX][liftCardY].push( recursiveStack.topAndPop());
                }
                conditionalActions.doValid(dropCardX, dropCardY);
            }

            else{
                if( cardLifted && detectDropStack(e.getX(), e.getY()) ){
                    undoPtr = theGrid[dropCardX][dropCardY].size();

                    for(int i=0; i<recursiveStack.size();){

                        Vector twoCards = new Vector();
                        System.out.println("Testing "+recursiveStack.top()+" ["+(i+1)+" of
"+recursiveStack.size()+"]");

                        if( undoPtr==0 && theGrid[dropCardX][dropCardY].isEmpty()){
                            twoCards.addElement(recursiveStack.top());
                        }
                        else{
                            twoCards.addElement(recursiveStack.top());

twoCards.addElement(theGrid[dropCardX][dropCardY].elementAt(theGrid[dropCardX][dropCardY]
.size()-1));
                        }
                        if( placeRules.isValid(twoCards ,dropCardX, dropCardY) ){
                            theGrid[dropCardX][dropCardY].push(recursiveStack.topAndPop());
                        }
                        else{
                            performUndo = true;
                            // Return all the cards from the destination back to the
recursive stack
```

```
                              for(int j=theGrid[dropCardX][dropCardY].size(); j>undoPtr; j--
){

        recursiveStack.push(theGrid[dropCardX][dropCardY].topAndPop());
                              }
                              i=recursiveStack.size();
                          }
                      }
                      // Return all the cards from the recursive stack back to thier source
                      if( performUndo ){
                          while( !recursiveStack.isEmpty() ){
                             theGrid[liftCardX][liftCardY].push( recursiveStack.topAndPop());
                          }
                      }
                  }

              }

              theGridPanel.unHighlightCard();
              performUndo = false;


              //check if turn is over
              if(currentTurn.checkIfTurnOver())
                  {
                      //last turn?
                      if(currentTurnIndex==turnVector.size()-1)
                          {
                              System.out.println("GAME OVER!!!");
                              JOptionPane.showMessageDialog(frame, "GAME OVER");
                              gameover=true;
                          }

                      else
                          {
                              //next turn
                              currentTurnIndex++;
                              currentTurn=((Turn)turnVector.elementAt(currentTurnIndex));

                              //reset rule and action grids
                              takeRules = currentTurn.takeGrid;
                              placeRules = currentTurn.placeGrid;
                              conditionalActions= currentTurn.actionGrid;
                          }
                  }
        }

    }

    public void mouseEntered(MouseEvent e){
        //outputConsole("Mouse entered",  e);
    }

    public void mouseExited(MouseEvent e){
        //outputConsole("Mouse exited",  e);
    }

    public void mouseClicked(MouseEvent e){
        //outputConsole("Mouse clicked (# of clicks: "+e.getClickCount()+")", e);
    }

    void outputConsole(String eventDescription, MouseEvent e){
        System.out.println(eventDescription+" detected on
"+e.getComponent().getClass().getName()+
                " at coordinates: "+e.getX()+", "+e.getY());
    }

    boolean detectLiftCard(int liftX, int liftY){
        liftCardX = (liftX-buffer/2)/cellWidth;
        liftCardY = (liftY-buffer/2)/cellHeight;
        liftCardIndex = ((liftY-(liftCardY*cellHeight+(buffer/2)))/cardDspSize)-1;
```

```
        try{
            System.out.println("Mouse Selection LIFT on card:
"+theGrid[liftCardX][liftCardY].elementAt(liftCardIndex));
            return true;
        }
        catch(Exception e){
            System.out.println(e+" while detecting LIFT card");
            return false;
        }
    }

    boolean detectDropStack(int dropX, int dropY){
        dropCardX = (dropX-buffer/2)/cellWidth;
        dropCardY = (dropY-buffer/2)/cellHeight;
        //dropCardIndex = ((dropY-(dropCardY*cellHeight+(buffer/2)))/15)-1;
        dropCardIndex = theGrid[dropCardX][dropCardY].size()-1;

        try{
            System.out.print("Mouse Selection DROP on stack: "+dropCardX+","+dropCardY);
            if( theGrid[dropCardX][dropCardY].isEmpty() ){
                System.out.print(" with Empty Stack");
            }
            else if( dropCardIndex < theGrid[dropCardX][dropCardY].size() ){
                System.out.print(" with top card
"+theGrid[dropCardX][dropCardY].elementAt(dropCardIndex));
            }
            System.out.println();
            return true;
        }
        catch(Exception e){
            System.out.println(e+" while detecting DROP card");
            return false;
        }
    }
}



class GridPanel extends JPanel{

    private CardStack theGrid[][];
    int cellWidth;
    int cellHeight;
    int width;
    int height;
    int buffer;
    Font cardFont;
    double xStart;
    double yStart;
    boolean highlight = false;
    int cardDspSize;

    public GridPanel(CardStack g[][], int x, int y, int cw, int ch, int b, int dsp) {
        theGrid = g;
        cellWidth=cw;
        cellHeight=ch;
        width = x;
        height= y;
        buffer= b;
        cardDspSize= dsp;
    }

    // The paintComponent method is called every time
    // that the panel needs to be displayed or refreshed.
    // Anything you want drawn on the panel should be drawn
    // in this method.

    public void paintComponent(Graphics page) {
         super.paintComponent(page);
        cardFont = new Font("Helvetica", Font.BOLD, cardDspSize-1);
```

```
        setFont(cardFont);
        setBackground(Color.white);

        page.setColor(Color.black);

        for(int i=0; i<=width; i++){
            page.drawLine((i*cellWidth)+buffer/2, buffer/2, (i*cellWidth)+buffer/2,
(height*cellHeight)+buffer/2 );
        }
        for(int i=0; i<=height; i++){
            page.drawLine(buffer/2, (i*cellHeight)+buffer/2, (width*cellWidth)+buffer/2,
(i*cellHeight)+buffer/2);
        }

        // Draw any stack that's not empty
        for(int i=0; i<width; i++){
            for(int j=0; j<height; j++){
                if( theGrid[i][j].top() != null ){
                    drawStackAt(i, j, page);
                }
            }
        }

        if( highlight )
            page.fill3DRect((int)xStart, (int)yStart, cellWidth-4, cardDspSize, true);

    }

    private void drawStackAt(int x, int y, Graphics page){

        //Draws rankLabels in a row
        for(int j=0; j<theGrid[x][y].size(); j++){
            Card drawCard = theGrid[x][y].elementAt(j);
            if( drawCard.VISIBLE){
                String cardID = new String();
                int upToDimension = 0;
                if( isColor(drawCard.rankLabels[drawCard.rankLabels.length-1]) )
                    upToDimension = drawCard.rankLabels.length-1;
                else
                    upToDimension = drawCard.rankLabels.length;
                for(int i=0; i<upToDimension; i++){
                    cardID += (" "+drawCard.rankLabels[i]);
                }

                page.setColor( getColor(drawCard.rankLabels[drawCard.rankLabels.length-
1]) );
                page.drawString(cardID, ((x*cellWidth)+2)+buffer/2,
(y*cellHeight)+((j+2)*cardDspSize)+(buffer/2)-1);
                page.setColor( Color.BLACK );
                if( j==theGrid[x][y].size()-1 ){
                    page.draw3DRect(((x*cellWidth))+(buffer/2+2),
(y*cellHeight)+((j+1)*cardDspSize)+buffer/2, cellWidth-4, cardDspSize, true);
                }
                else
                    page.draw3DRect(((x*cellWidth))+(buffer/2+2),
(y*cellHeight)+((j+1)*cardDspSize)+buffer/2, cellWidth-4, cardDspSize, true);
            }
            else
                page.draw3DRect(((x*cellWidth))+(buffer/2+2),
(y*cellHeight)+((j+1)*cardDspSize)+buffer/2, cellWidth-4, cardDspSize, true);
                //page.draw3DRect("******************", ((x*cellWidth)+2)+buffer/2,
(y*cellHeight)+((j+2)*10)+buffer/2);
        }
    }

    private Color getColor(String colorString){

        if( colorString.compareToIgnoreCase("BLUE")==0 )
            return Color.BLUE;
        else if( colorString.compareToIgnoreCase("CYAN")==0 )
            return Color.CYAN;
```

```
    else if( colorString.compareToIgnoreCase("DARK_GRAY")==0 )
        return Color.DARK_GRAY;
    else if( colorString.compareToIgnoreCase("DARK_GREY")==0 )
        return Color.DARK_GRAY;
    else if( colorString.compareToIgnoreCase("GRAY")==0 )
        return Color.GRAY;
    else if( colorString.compareToIgnoreCase("GREEN")==0 )
        return Color.GREEN;
    else if( colorString.compareToIgnoreCase("LIGHT_GRAY")==0 )
        return Color.DARK_GRAY;
    else if( colorString.compareToIgnoreCase("LIGHT_GREY")==0 )
        return Color.DARK_GRAY;
    else if( colorString.compareToIgnoreCase("ORANGE")==0 )
        return Color.ORANGE;
    else if( colorString.compareToIgnoreCase("MAGENTA")==0 )
        return Color.MAGENTA;
    else if( colorString.compareToIgnoreCase("PINK")==0 )
        return Color.PINK;
    else if( colorString.compareToIgnoreCase("RED")==0 )
        return Color.RED;
    else if( colorString.compareToIgnoreCase("WHITE")==0 )
        return Color.WHITE;
    else if( colorString.compareToIgnoreCase("YELLOW")==0 )
        return Color.YELLOW;
    else
        return Color.BLACK;
}

private boolean isColor(String colorString){

    if( colorString.compareToIgnoreCase("BLUE")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("CYAN")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("DARK_GRAY")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("DARK_GREY")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("GRAY")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("GREEN")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("LIGHT_GRAY")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("LIGHT_GREY")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("ORANGE")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("MAGENTA")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("PINK")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("RED")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("WHITE")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("YELLOW")==0 )
        return true;
    else if( colorString.compareToIgnoreCase("BLACK")==0 )
        return true;
    else
        return false;
}

public void highlightCardAt(int x, int y){

    double modY = ((y-(buffer/2))%(cardDspSize));
     yStart = y-modY;

    double modX = ((x-(buffer/2))%(cellWidth));
    xStart = (x-modX)+2;
```

```
        highlight = true;

        repaint();
    }

    public void unHighlightCard(){
        highlight = false;
        repaint();
    }

}
```

8.2.6 Turn.java

```
Turn.java
import java.util.Vector;
/* LOG
 *================================
 * Last Modified: 5.6.03 by cheryl
 *
 *
 * -----
 * 5.6.03
 *
 * created the class
 *
 */

/* Turn Methods
 *
 *
 * public Turn(Vector r, RuleGrid tg, RuleGrid pg, ActionGrid ag)
 *    Create a new turn with a TakeGrid, PlaceGrid, and ActionGrid along with a
 *    vector of conditionals ending the turn.
 * public boolean checkIfTurnOver()
 *    Checks to see if the turn is over.
 *
 */

/*
The turn object represents a turn in a game.  A turn keeps going until the conditional is
true.
 In solitaire, the game is made of one turn.  The conditional for solitaire is when all 4
upper
 right hand spots contain kings, which is when the game ends.
*/


public class Turn
{
    //vector of conditionals (Vector of Rule objects that have Conditionals)
    //if all conditionals are true, the turn ends
    Vector conditionals;

    //is this turn over?
    //set to true when turn ends
    public boolean turnOver;

    //the rule grids and action grid
    public RuleGrid takeGrid;
    public RuleGrid placeGrid;
    public ActionGrid actionGrid;

    //constructor
    public Turn(Vector r, RuleGrid tg, RuleGrid pg, ActionGrid ag)
    {
        conditionals=r;
        turnOver=false;

        //set the rule grids and action grid
```

```
        takeGrid=tg;
        placeGrid=pg;
        actionGrid=ag;
    }

    //test if turn should loop again
    //returns true if all conditionals are true (turn over, exit turn loop)
    //returns false if conditional test failed and turn not over yet
    public boolean checkIfTurnOver()
    {
        //loop through vector, checking each conditional
        //return false if one fails
        for(int i=0;i<conditionals.size();i++)
            {
                //get Conditional
                Conditional c=((Rule)conditionals.elementAt(i)).conditional;

                //check it, if conditional failed, return false
                if(!c.check())
                    return false;
            }

        //all conditionals are satisfied, end turn, return true
        turnOver=true;
        return true;
    }

}
```

8.2.7 RuleGrid.java

RuleGrid.java

```
/* LOG
 *================================
 * Last Modified: 5.5.03 by cheryl
 *
 *
 * -----
 * 5.5.03
 *
 * created the class
 *
 */

/* RuleGrid Methods
 *
 *    public RuleGrid(int x, int y, Deck d)
 *
 *    public void setRule(Rule r, int x, int y)
 *
 *    public boolean isValid(Vector cards, int x, int y)
 */

/*
A RuleGrid object contains the rules for each grid box in the grid.  Each grid box can be
associated with two types of rules:  place, take.  A place rule allows the placement of a
card into that grid box if the rule is followed.  A take rule allows the player to take a
card from that grid box if the rule is followed.  A RuleGrid object is a 2 dimensional
array
representing the entire grid.  Each element of that 2 dimensional array represents a grid
box.
 Since each grid box has rules associated with it, each element of the array contains a
Vector
 of Rule objects.  To differentiate between the two types of rule, place and take, two
different
  RuleGrid objects will need to be made, one for place, and one for take.RuleGrid objects,
  however, are independent of place and take, meaning the objects themselves have no
concept
  of whether the RuleGrid represents a grid of place rules or take rules.
```

```
*/

import java.util.Vector;

public class RuleGrid
{
    //2 dimensional array representing the grid
    //each element represents a grid box
    //each element is a Vector of Rule objects
    public Vector[][] theRuleGrid;
    private int xLength, yLength;  //size of grid

    private Deck deck;

    //constructor
    // x is width of grid
    // y is height of grid
    //initialize to empty rulegrid
    public RuleGrid(int x, int y, Deck d)
    {
        deck=d;

        xLength=x;
        yLength=y;
        theRuleGrid=new Vector[x][y];

        //initialize each element to an empty Vector
        for(int i=0;i<x;i++)
            {
                for(int j=0;j<y;j++)
                    theRuleGrid[i][j]=new Vector();
            }
    }

    //add a Rule r to the rule grid, add to grid box (x, y)
    public void setRule(Rule r, int x, int y)
    {
        theRuleGrid[x][y].add(r);
    }

    //check to see if vector of cards follow a rule in grid box
    //place:  1st card in vector is card being moved to the grid box
    public boolean isValid(Vector cards, int x, int y)
    {
        //get the vector of rules for grid box (x, y)
        Vector rules=theRuleGrid[x][y];

        //if no rules exist, return true
        if(rules.size()==0)
            {
                System.out.println("valid move:  no rules exist!");
                return true;
            }

        //check each rule in the Vector for validity
        //if a single rule is found to be valid, return true
        //if no rule is valid, return false
        for(int i=0;i<rules.size();i++)
            {
                //get a rule
                Rule r=((Rule)rules.elementAt(i));

                //if vector doesnt have correct # cards for particular rule
                // the isValid ftn for Rule returns false

                System.out.print("Testing Rule "+ i+": ");
                for(int test=0;test<r.d;test++){
                    System.out.print("\""+r.dimArray[test].type+"
"+r.dimArray[test].value+"\" ");
                }
```

```
                //if rule is valid, return true
                if(r.isValid(cards, deck))
                    {
                        System.out.println("Rule result: true");
                        return true;
                    }
                else
                    System.out.println("Rule result: false");
            }

        //finished looping through vector of rules, no rule matched
        return false;
    }
}
```

8.2.8 ActionGrid.java

```
ActionGrid.java
/* LOG
 *================================
 * Last Modified: 5.6.03 by cheryl
 *
 *
 * -----
 * 5.6.03
 *
 * created the class
 *
 */

/* ActionGrid Methods
 *
 * public ActionGrid(int x, int y, Grid g)
 *
 * public void setActionConditional(ActionConditional ac, int x, int y)
 *
 * public void doValid(int x, int y)
 *
 */

/*
An ActionGrid is similar to a RuleGrid.  Instead, an ActionGrid is a 2-dimensional array
of ActionConditionals rather than Rules.  Also, the doValid() method performs the actions
if the
conditionals are satisfied.  Just like a RuleGrid, an ActionGrid represents the grid, and
the
Vector of ActionConditionals in each cell in the 2D array are the ActionConditionals
associated
with a particular grid box.  When the mouse is clicked in a particular grid box, the
action
conditionals are checked and executed as necessary.
*/

import java.util.Vector;

public class ActionGrid
{
    //2 dimensional array representing the grid
    //each element represents a grid box
    //each element is a Vector of ActionConditional objects
    public Vector[][] theActionGrid;
    private int xLength, yLength;  //size of grid
    private Grid theGrid;

    //constructor
    // x is width of grid
    // y is height of grid
    //initialize to empty action grid
    public ActionGrid(int x, int y, Grid g)
    {
```

```java
        xLength=x;
        yLength=y;
        theActionGrid=new Vector[x][y];

        //initialize each element to an empty Vector
        for(int i=0;i<x;i++)
            {
                for(int j=0;j<y;j++)
                    theActionGrid[i][j]=new Vector();
            }

        theGrid=g;
    }

    //add an ActionConditional ac to the action grid, add to grid box (x, y)
    public void setActionConditional(ActionConditional ac, int x, int y)
    {
        theActionGrid[x][y].add(ac);
    }

    //check ActionConditionals, do the actions for conditionals that are true
    public void doValid(int x, int y)
    {
        System.out.println("Performing actions for Grid["+x+","+y+"]");
        //get the vector of action conditionals for grid box (x, y)
        Vector actionConditionals=theActionGrid[x][y];

        //if no action conditionals exist, do nothing
        if(actionConditionals.size()==0)
            System.out.println("No action specified here.");

        //loop through Vector, check each ActionConditional
        for(int i=0;i<actionConditionals.size();i++)
            {
                //get an ActionConditional
                ActionConditional ac=((ActionConditional)actionConditionals.elementAt(i));

                ac.doValid(theGrid);
            }
    }
}
```

8.2.9 RuleType.java

RuleType.java

```java
/* LOG
 *================================
 * Last Modified: 4.25.03 by cheryl
 *
 * created the class
 *
 */

/* RuleType Methods
 *
 *    public RuleType()
 *
 *    public RuleType(String t, String val)
 *
 *
 */


/*what type of rule is it and what is its value?
- plus
- minus
- multipleMatch
- multipleMatchEq
- exactValue
- exactValueEq
```

```
- star
*/

public class RuleType
{
    public String type;  //type of rule (as listed above)
    public String value;    //value of rule

    //default constructor, for star rule type
    public RuleType()
    {
       type="star";
       value="";  //star doesnt need a value
    }

    //constructor
    public RuleType(String t, String val)
    {
       type=t;
       value=val;
    }
}
```

8.2.10 Rule.java

```
Rule.java
/* LOG
 *================================
 * Last Modified: 5.6.03 by cheryl
 *
 * added conditional rules functionality
 *
 * -----
 * 5.5.03
 *
 * conglomerated () cases and [] cases into one isValid ftn
 *
 * -----
 * 5.4.03
 *
 * fixed [] cases
 *   new method:
 *   public boolean isValid(Card c, Deck deck)
 *
 * fixed multipleMatch and multipleMatchEq
 *   changed params of isValid, Vector of cards instead of 2 Cards:
 *   public boolean isValid(Vector cards, Deck deck)
 *
 * -----
 * 4.25.03
 *
 * created the class
 *
 */

/* Rule Methods
 *
 *    public Rule(int dim)
 *
 *    public boolean isValid(Vector cards, Deck deck)
 *
 *    public void setPlus(int dim, int val)
 *
 *    public void setMinus(int dim, int val)
 *
 *    public void setMultipleMatch(int dim, int val)
 *
 *    public void setMultipleMatchEq(int dim, int val)
 *
 *    public void setExactValue(int dim, String val)
```

```
 *
 *    public void setExactValueEq(int dim, String val)
 *
 *
 */

/*A Rule object embodies a () or [] restriction specified in a place or take statement.
 * A single Rule object needs to be created for every restriction that is placed on the
grid box.
 * For example, a place rule can contain restrictions on what cards can be put in the
grid box.
 * A card is can be placed in a grid box if it follows that the top card on the stack and
the
 * card being moved to the grid box comply with the match rule.  The take rule contains
 * restrictions on whether cards can be taken or not from the grid box.  A card can be
taken
 * if the top two cards on the stack match the rule.  Each grid box has as many place and
take
 * rules as necessary to control card placement and withdrawal.

place[ (*, +1) (2, +2) [K, SPADES] ]

Rule1=(*, +1)
Rule2=(2, +2)
Rule3=[K, SPADES]

To associate this place rule, which is made of three Rule objects, with grid box (0, 0)
create
a RuleGrid object for place rules and add the Rules to the RuleGrid:
RuleGrid placeRules=new RuleGrid(7, 3, deck);   // 7x3 grid
placeRules.setRule(Rule1, 0, 0);
placeRules.setRule(Rule2, 0, 0);
placeRules.setRule(Rule3, 0, 0);

To associate a take rule with a grid box, create a RuleGrid object for take rules and
call
setRule() on that rule grid.

Conditional Rules
-----------------
Conditional rules are implemented by calling the Rule constructor that accepts a
Conditional object.
 The boolean called conditionalOn tells whether the particular Rule object is a
conditional rule or not.
   The isValid() ftn tests the conditional for conditional rules before testing the rule
itself.
   It returns false if the conditional is false.  It returns true iff the conditional and
the
   rule are both true.
*/

import java.util.Vector;

public class Rule
{
    //each dimension has a rule
    public RuleType[] dimArray;
    public int d;  //number of dimensions on a card

    //for conditional rules
    public boolean conditionalOn;
    public Conditional conditional;

    //constructor
    //int dim is total number of dimensions
    public Rule(int dim)
    {
        d=dim;

        //each dimension has an index
        dimArray=new RuleType[d];
```

```
    //set each dimension's rule to star as default
    for(int i=0;i<d;i++)
        {
            RuleType star=new RuleType();
            dimArray[i]=star;
        }

    conditionalOn=false;
}

//constructor for conditional rules
public Rule(int dim, Conditional c)
{
    //set conditionals on
    conditionalOn=true;
    conditional=c;

    d=dim;

    //each dimension has an index
    dimArray=new RuleType[d];

    //set each dimension's rule to star as default
    for(int i=0;i<d;i++)
        {
            RuleType star=new RuleType();
            dimArray[i]=star;
        }
}

//for () cases:  plus, minus, multipleMatch, multipleMatchEq, (star)
//for [] cases:  exactValue, exactValueEq, (star)
//test if cards from deck follow Rule
//cards are stored in a Vector
//first card in vector is card being moved to grid space
//plus, minus:  check the first two cards in the vector
//multipleMatch, multipleMatchEq:  check as many cards as needed
//exactValue, exactValueEq:  check first card in vector
//if vector doesnt have correct # cards for particular rule, return false
//rules are based on precedence order specified in ranksys
//(ranksys lives in rankedArray in the Deck class)
public boolean isValid(Vector cards, Deck deck)
{

    //check if conditionals are on
    if(conditionalOn)
        {
            boolean testing=conditional.check();
            System.out.println("conditional satisfied?"+testing);
            //if conditional isnt true, return false
            //otherwise, proceed through ftn checking rules
            if(!testing)
                return false;
        }
    else
        System.out.println("not bound by conditional.");

    //loop through dimensions checking each rule
    for(int dim=0;dim<d;dim++)
        {
            String type=dimArray[dim].type;  //get type of rule

            //each dimension can only have one type of rule

            //c1 is first card in vector
            //c2 is second card in vector

            //c2 has to have a value that is
            // n precedence levels above c1's value
            if(type.compareToIgnoreCase("plus")==0)
```

```
                    {
                        //check to see there are two cards in vector
                        if(cards.size()<2)
                            {
                                System.out.println("validation:  plus:  not enough cards in
vector");
                                return false;
                            }

                        //find values on the cards for a dimension
                        String val1=((Card)cards.elementAt(0)).rankLabels[dim];
                        String val2=((Card)cards.elementAt(1)).rankLabels[dim];

                        //find rank values for the cards for a dimension
                        int rank1=deck.rankof(val1, dim);
                        int rank2=deck.rankof(val2, dim);

                        //get specified offset
                        int offset=Integer.parseInt(dimArray[dim].value);

                        //if rank2 is not greater than rank1 by specified num
                        //no match, return false
                        if((rank1+offset) != rank2)
                            return false;
                    }

                //c2 has to have a value that is
                // n precedence levels below c1's value
                if(type.equals("minus"))
                    {
                        //check to see there are two cards in vector
                        if(cards.size()<2)
                            {
                                System.out.println("validation:  minus:  not enough cards
in vector");
                                return false;
                            }

                        //find values on the cards for a dimension
                        String val1=((Card)cards.elementAt(0)).rankLabels[dim];
                        String val2=((Card)cards.elementAt(1)).rankLabels[dim];

                        //find rank values for the cards for a dimension
                        int rank1=deck.rankof(val1, dim);
                        int rank2=deck.rankof(val2, dim);

                        //get specified offset
                        int offset=Integer.parseInt(dimArray[dim].value);

                        //if rank2 is not less than rank1 by specified num
                        //no match, return false
                        if(rank1 != (rank2+offset))
                            return false;
                    }

                //all cards in vector have to have the same exact value
                //this value can be any value from the dimension
                if(type.compareToIgnoreCase("multipleMatch")==0)
                    {
                        //get # cards to match
                        int numMatch=Integer.parseInt(dimArray[dim].value);

                        //check to see there are correct # cards in vector
                        if(cards.size()<numMatch)
                            {
                                System.out.println("validation:  multipleMatch:  not enough
cards in vector");
                                return false;
                            }

                        //find value of first card for a dimension
```

```
                    String val1=((Card)cards.elementAt(0)).rankLabels[dim];

                    //loop through Vector comparing value to first card
                    for(int i=1;i<numMatch;i++)
                        {
                            //find value on card for a dimension
                            String val=((Card)cards.elementAt(i)).rankLabels[dim];

                            //compare card's value to first card's value
                            //if doesnt match, return false
                            if(!val.equals(val1))
                                return false;
                        }
                }

            //all cards in vector have to have values
            // from the same equivalence class in the dimension
            if(type.compareToIgnoreCase("multipleMatchEq")==0)
                {
                    //get # cards to match
                    int numMatch=Integer.parseInt(dimArray[dim].value);

                    //check to see there are correct # cards in vector
                    if(cards.size()<numMatch)
                        {
                            System.out.println("validation:  multipleMatchEq:  not
enough cards in vector");
                            return false;
                        }

                    //find value of first card for a dimension
                    String val1=((Card)cards.elementAt(0)).rankLabels[dim];

                    //find rank of first card for a dimension
                    int rank1=deck.rankof(val1, dim);

                    //loop through Vector comparing rank to first card
                    for(int i=1;i<numMatch;i++)
                        {
                            //find value on card for a dimension
                            String val=((Card)cards.elementAt(i)).rankLabels[dim];

                            //find rank of card for a dimension
                            int rank=deck.rankof(val, dim);

                            //no match, return false
                            if(rank != rank1)
                                return false;
                        }
                }

            //c is first card in Vector
            //for [] cases, only need 1st card

            // [] case
            // c has to match the exact value
            // that is specified in RuleType
            if(type.compareToIgnoreCase("exactValue")==0)
                {

                    //check to see there is one card in vector
                    if(cards.size()<1)
                        {
                            System.out.println("validation:  exactValue:  not enough
cards in vector");
                            return false;
                        }

                    //find value on the card for a dimension
                    String cVal=((Card)cards.elementAt(0)).rankLabels[dim];
```

```
                    //find specific value to match
                    String val=dimArray[dim].value;

                    //if card doesnt match the specified value
                    //no match, return false
                    if(!cVal.equals(val))
                        return false;
                }

            // [] case
            //c has to have a value from the same
            // equivalence class as the value specified in RuleType
            if(type.compareToIgnoreCase("exactValueEq")==0)
                {
                    //check to see there is one card in vector
                    if(cards.size()<1)
                        {
                            System.out.println("validation:  exactValueEq:  not enough
cards in vector");
                            return false;
                        }

                    //find value on the card for a dimension
                    String cVal=((Card)cards.elementAt(0)).rankLabels[dim];

                    //find rank value for the card for a dimension
                    int cRank=deck.rankof(cVal, dim);

                    //find specific rank to match
                    String val=dimArray[dim].value;
                    int rank=deck.rankof(val, dim);

                    //no match, return false
                    if(cRank!=rank)
                        return false;
                }

            //if type is star,
            //everything in this dimension matches

        }//end for loop

        //each dimension's rule matches
        return true;
    }


    //for all the set methods:
    //int dim is the dimension number to associate this rule with
    //int val is the value of the rule, what the rule actually is
    //if you want to associate the rule with ranksys.1, dim=1
    //if you want to associate the rule with ranksys.i, dim=i
    //so, with (*, +3) the plus rule is associated with dim=2
    //and the value of this plus rule would be val=3

    //set plus constant rule (i.e. +1, +2, +int)
    public void setPlus(int dim, int val)
    {
        String sval=val+"";
        RuleType type=new RuleType("plus", sval);
        dimArray[dim]=type;
    }

    //set minus constant rule (i.e. -1, -2, -int)
    public void setMinus(int dim, int val)
    {
        String sval=val+"";
        RuleType type=new RuleType("minus", sval);
        dimArray[dim]=type;
    }
```

```java
    //set multiple match rule (i.e. 2 for pairs, 3 for triplets, etc)
    public void setMultipleMatch(int dim, int val)
    {
        String sval=val+"";
        RuleType type=new RuleType("multipleMatch", sval);
        dimArray[dim]=type;
    }

    //set multiple match rule with equivalence classes
    //anything in equiv class counts as pair/triplet/multiple
    public void setMultipleMatchEq(int dim, int val)
    {
        String sval=val+"";
        RuleType type=new RuleType("multipleMatchEq", sval);
        dimArray[dim]=type;
    }

    //set exact value match rule
    public void setExactValue(int dim, String val)
    {
        RuleType type=new RuleType("exactValue", val);
        dimArray[dim]=type;
    }

    //set exact value match with equivalence classes
    //anything in equiv class counts as exact match of value
    public void setExactValueEq(int dim, String val)
    {
        RuleType type=new RuleType("exactValueEq", val);
        dimArray[dim]=type;
    }

}
```

8.2.11 FlipRule.java

FlipRule.java

```java
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * -----
 * 5.9.03
 *
 * created the class
 *
 */

/* FlipRule Methods
 *
 * public static void flip(Vector rules)
 *
 */

/*
FlipRule contains one static method that takes in a Vector of Rule objects.
This method is called flip and can be called as such:
      FlipRule.flip(myVectorOfRules);
The Conditional part in every Rule in the Vector is flipped.  In other words,
the Conditional is reset to be a Conditional that is the opposite of the
original Conditional.

IsEmpty         <-->  NotIsEmpty
Contains        <-->  NotContains
AtDepth         <-->  NotAtDepth
UpToDepth       <-->  NotUpToDepth
IsTotallyEmpty  <-->  NotIsTotallyEmpty
 */
```

```
import java.util.Vector;

public class FlipRule
{
    //flip the Conditional part in every Rule in the Vector
    //reset the Conditional to be the opposite of the original Conditional
    public static Vector flip(Vector rules)
    {
        //iterate through Vector
        for(int i=0;i<rules.size();i++)
            {
                //get the Rule
                Rule r=((Rule)rules.elementAt(i));

                System.out.print("before: "+r.conditional);

                //if rule has a conditional, flip it
                if(r.conditionalOn)
                    {
                        Conditional c=r.conditional.flip();
                        r.conditional=c;
                    }

                System.out.println("\tafter: "+r.conditional);
            }
        return rules;
    }
}
```

8.2.12 Conditional.java

```
Conditional.java
/* LOG
 *================================
 * Last Modified: 5.10.03 by cheryl
 *
 * added toString() method
 *
 * -----
 * 5.9.03
 *
 * added flip()  //changes the type of Conditional to its opposite
 *
 * -----
 * 5.5.03
 *
 * created the interface
 *
 */

/* Conditional Methods
 *
 * public boolean check()
 *
 * public Conditional flip()
 *
 */

/*
 *  TEST CONDITIONALS
 *
 *    isEmpty - no visible cards on stack
 *    contains - specified card is in visible stack
 *    atDepth - specified card is in visible stack at specified depth
 *    upToDepth - specified card is in visible stack up to specified depth
 *    isTotallyEmpty - no cards on stack
 */

/*
Types of Conditionals
---------------------
```

```
public IsEmpty(Grid grid, int xCoord, int yCoord)
public Contains(Grid grid, int xCoord, int yCoord, Rule r, Deck d)
public AtDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
public UpToDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
public IsTotallyEmpty(Grid grid, int xCoord, int yCoord)

Not Conditionals
----------------
public NotIsEmpty(Grid grid, int xCoord, int yCoord)
public NotContains(Grid grid, int xCoord, int yCoord, Rule r, Deck d)
public NotAtDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
public NotUpToDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
public NotIsTotallyEmpty(Grid grid, int xCoord, int yCoord)
*/


public interface Conditional
{
    //possible types of conditionals
    // "isEmpty"
    // "contains"
    // "atDepth"
    // "upToDepth"
    // "isTotallyEmpty"

    //check to see if conditional is satisfied
    public boolean check();

    //flip the conditional to its opposite
    public Conditional flip();

    //to print the conditional
    public String toString();
}
```

8.2.13 AtDepth.java

AtDepth.java

```
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.6.03
 *
 * wrote check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* AtDepth Methods
 *
 *    public boolean check()
 *
 */

/*
AtDepth
-------
depth is measured from top of stack
TRUE - card at specified depth in stack of visible cards in grid box (x, y) follows rule
FALSE - card at specified depth in stack of visible cards does not follow rule
```

```
Rules in Conditionals can only be of the [] case because Contains, AtDepth, and UpToDepth
only test an exact match of one card.
 */

import java.util.Vector;

public class AtDepth implements Conditional
{
    Grid theBigGrid;
    CardStack[][] theGrid;  //the grid
    int x, y;  //coordinates of grid box
    Rule rule;  //rule to find a matching card by
    int depth;  //depth to find the matching card at
    Deck deck;

    public String id;

    //constructor
    public AtDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
    {
        theBigGrid=grid;
        this.theGrid=grid.theGrid;
        x=xCoord;
        y=yCoord;
        rule=r;
        depth=dp;
        deck=d;

        id="AtDepth";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        //find index of card at specified depth
        int i=theGrid[x][y].size() - depth;

        //valid depth, card exists at this depth
        if(i>=0)
            {
                //get card at specified depth
                Card c=theGrid[x][y].elementAt(i);

                //if card is visible, check it for rule match
                if(c.VISIBLE)
                    {
                        Vector v=new Vector();
                        v.add(c);

                        //if rule matches, return true
                        if(rule.isValid(v, deck))
                            return true;
                    }
            }

        //if card is not visible, return false
        //depth is out of range of stack, return false
        return false;
    }

    //flip conditional to be the opposite:  NotAtDepth
    public Conditional flip()
    {
        NotAtDepth c=new NotAtDepth(theBigGrid, x, y, rule, depth, deck);
        return c;
    }

    //for printing
    public String toString()
    {
        return id;
```

```
    }
}
```

8.2.14 NotAtDepth.java

```
NotAtDepth.java
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.7.03
 *
 * wrote check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* NotAtDepth Methods
 *
 *    public boolean check()
 *
 */

/*
NotAtDepth
----------
returns opposite of AtDepth

AtDepth
-------
depth is measured from top of stack
TRUE - card at specified depth in stack of visible cards in grid box (x, y) follows rule
FALSE - card at specified depth in stack of visible cards does not follow rule

Rules in Conditionals can only be of the [] case because Contains, AtDepth, and
UpToDepth only test an exact match of one card.
 */

import java.util.Vector;

public class NotAtDepth implements Conditional
{
    Grid theGrid;  //the grid
    int x, y;  //coordinates of grid box
    Rule rule;  //rule to find a matching card by
    int depth;  //depth to find the matching card at
    Deck deck;

    public String id;

    //constructor
    public NotAtDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
    {
       this.theGrid=grid;
       x=xCoord;
       y=yCoord;
       rule=r;
       depth=dp;
       deck=d;

       id="NotAtDepth";
    }
```

```
    //check to see if conditional is satisfied
    public boolean check()
    {
       AtDepth conditional=new AtDepth(theGrid, x, y, rule, depth, deck);
       return !conditional.check();
    }

    //flip conditional to be the opposite:  AtDepth
    public Conditional flip()
    {
       AtDepth c=new AtDepth(theGrid, x, y, rule, depth, deck);
       return c;
    }

    //for printing
    public String toString()
    {
       return id;
    }
}
```

8.2.15 Contains.java

```
Contains.java
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.6.03
 *
 * wrote the check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* Contains Methods
 *
 *    public boolean check()
 *
 */

/*
Contains
--------
TRUE - a card that follows the rule can be found in stack of visible cards in specified
grid box (x, y)
FALSE - no card in stack of visible cards follows the rule

Rules in Conditionals can only be of the [] case because Contains, AtDepth, and UpToDepth
only test an exact match of one card.
 */

import java.util.Vector;

public class Contains implements Conditional
{
    Grid theBigGrid;
    CardStack[][] theGrid;  //the grid
    int x, y;  //coordinates of grid box
    Rule rule;  //rule to find a matching card by
    Deck deck;
```

```
    public String id;

    //constructor
    public Contains(Grid grid, int xCoord, int yCoord, Rule r, Deck d)
    {
        theBigGrid=grid;
        this.theGrid=grid.theGrid;
        x=xCoord;
        y=yCoord;
        rule=r;
        deck=d;

        id="Contains";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        //iterate through stack of cards starting at top of stack
        //stop looking when you get to nonvisible cards
        for(int i=theGrid[x][y].size()-1;i>=0;i--)
            {
                Card c=theGrid[x][y].elementAt(i);

                //if card is visible, check it for rule match
                if(c.VISIBLE)
                    {
                        Vector v=new Vector();
                        v.add(c);
                        //if rule finds a match, return true
                        if(rule.isValid(v, deck))
                            return true;
                    }

                //if card is not visible, no more visible cards, return false
                else
                    return false;
            }

        //no rule matched
        return false;
    }

    //flip conditional to be the opposite:  NotContains
    public Conditional flip()
    {
        NotContains c=new NotContains(theBigGrid, x, y, rule, deck);
        return c;
    }

    //for printing
    public String toString()
    {
        return id;
    }

}
```

8.2.16 NotContains.java

NotContains.java

```
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.7.03
 *
```

```java
 * wrote the check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* NotContains Methods
 *
 *    public boolean check()
 *
 */

/*
NotContains
-----------
returns opposite of Contains

Contains
--------
TRUE - a card that follows the rule can be found in stack of visible cards in
specified grid box (x, y)
FALSE - no card in stack of visible cards follows the rule

Rules in Conditionals can only be of the [] case because Contains, AtDepth,
and UpToDepth only test an exact match of one card.
 */

import java.util.Vector;

public class NotContains implements Conditional
{
    Grid theGrid;  //the grid
    int x, y;  //coordinates of grid box
    Rule rule;  //rule to find a matching card by
    Deck deck;

    public String id;

    //constructor
    public NotContains(Grid grid, int xCoord, int yCoord, Rule r, Deck d)
    {
       this.theGrid=grid;
       x=xCoord;
       y=yCoord;
       rule=r;
       deck=d;

       id="NotContains";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
       Contains conditional=new Contains(theGrid, x, y, rule, deck);
       return !conditional.check();
    }

    //flip conditional to be the opposite:  Contains
    public Conditional flip()
    {
       Contains c=new Contains(theGrid, x, y, rule, deck);
       return c;
    }

    //for printing
    public String toString()
    {
       return id;
```

```
    }
}
```

8.2.17 IsEmpty.java

IsEmpty.java

```
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.6.03
 *
 * wrote the check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* IsEmpty Methods
 *
 *    public boolean check()
 *
 */

/*
IsEmpty
-------
TRUE - no visible cards on the stack in the specified grid box (x, y).
FALSE - there are visible cards on stack
 */

public class IsEmpty implements Conditional
{
    Grid theBigGrid;
    CardStack[][] theGrid;  //the grid
    int x, y;  //coordinates of grid box

    public String id;

    //constructor
    //test isEmpty for grid box (xCoord, yCoord)
    public IsEmpty(Grid grid, int xCoord, int yCoord)
    {
        theBigGrid=grid;
        this.theGrid=grid.theGrid;
        x=xCoord;
        y=yCoord;

        id="IsEmpty";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        //if top card is visible, visible card present, return false
        if(!theGrid[x][y].isEmpty() && theGrid[x][y].top().VISIBLE)
            return false;

        //if top card is not visible, no visible cards in stack, return true
        else
            return true;
    }
```

```
    //flip conditional to be the opposite:  NotIsEmpty
    public Conditional flip()
    {
       NotIsEmpty c=new NotIsEmpty(theBigGrid, x, y);
       return c;
    }

    //for printing
    public String toString()
    {
       return id;
    }
}
```

8.2.18 NotIsEmpty.java

NotIsEmpty.java

```
/* LOG
 *===============================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.7.03
 *
 * wrote the check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* NotIsEmpty Methods
 *
 *    public boolean check()
 *
 */

/*
NotIsEmpty
----------
returns opposite of IsEmpty

IsEmpty
-------
TRUE - no visible cards on the stack in the specified grid box (x, y).
FALSE - there are visible cards on stack
 */

public class NotIsEmpty implements Conditional
{
    Grid theGrid;  //the grid
    int x, y;  //coordinates of grid box

    public String id;

    //constructor
    //test !isEmpty for grid box (xCoord, yCoord)
    public NotIsEmpty(Grid grid, int xCoord, int yCoord)
    {
       this.theGrid=grid;
       x=xCoord;
       y=yCoord;

       id="NotIsEmpty";
    }
```

```
    //check to see if conditional is satisfied
    public boolean check()
    {
        IsEmpty conditional=new IsEmpty(theGrid, x, y);
        return !conditional.check();
    }

    //flip conditional to be the opposite:  IsEmpty
    public Conditional flip()
    {
        IsEmpty c=new IsEmpty(theGrid, x, y);
        return c;
    }

    //for printing
    public String toString()
    {
        return id;
    }
}
```

8.2.19 IsTotallyEmpty.java

```
IsTotallyEmpty.java
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.6.03
 *
 * wrote check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* IsTotallyEmpty Methods
 *
 *    public boolean check()
 *
 */

/*
IsTotallyEmpty
--------------
TRUE - no cards on stack (visible or nonvisible)
FALSE - stack is not empty
 */

public class IsTotallyEmpty implements Conditional
{
    Grid theBigGrid;
    CardStack[][] theGrid;  //the grid
    int x, y;  //coordinates of grid box

    public String id;

    //constructor
    public IsTotallyEmpty(Grid grid, int xCoord, int yCoord)
    {
        theBigGrid=grid;
        this.theGrid=grid.theGrid;
        x=xCoord;
```

```java
        y=yCoord;

        id="IsTotallyEmpty";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        return theGrid[x][y].isEmpty();
    }

    //flip conditional to be the opposite:  NotIsTotallyEmpty
    public Conditional flip()
    {
        NotIsTotallyEmpty c=new NotIsTotallyEmpty(theBigGrid, x, y);
        return c;
    }

    //for printing
    public String toString()
    {
        return id;
    }
}
```

8.2.20 NotIsTotallyEmpty.java

NotIsTotallyEmpty.java

```java
/* LOG
 *=================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.7.03
 *
 * wrote the check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* NotIsTotallyEmpty Methods
 *
 *    public boolean check()
 *
 */

/*
NotIsTotallyEmpty
----------
returns opposite of IsTotallyEmpty

IsTotallyEmpty
-------
TRUE - no visible cards on the stack in the specified grid box (x, y).
FALSE - there are visible cards on stack
 */

public class NotIsTotallyEmpty implements Conditional
{
    Grid theGrid;  //the grid
    int x, y;  //coordinates of grid box

    public String id;
```

```java
    //constructor
    //test !isTotallyEmpty for grid box (xCoord, yCoord)
    public NotIsTotallyEmpty(Grid grid, int xCoord, int yCoord)
    {
        this.theGrid=grid;
        x=xCoord;
        y=yCoord;

        id="NotIsTotallyEmpty";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        IsTotallyEmpty conditional=new IsTotallyEmpty(theGrid, x, y);
        return !conditional.check();
    }

    //flip conditional to be the opposite:  IsTotallyEmpty
    public Conditional flip()
    {
        IsTotallyEmpty c=new IsTotallyEmpty(theGrid, x, y);
        return c;
    }

    //for printing
    public String toString()
    {
        return id;
    }
}
```

8.2.21 UpToDepth.java

UpToDepth.java

```java
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.6.03
 *
 * wrote check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* UpToDepth Methods
 *
 *    public boolean check()
 *
 */

/*
UpToDepth
---------
depth is measured from top of stack
"up to depth" is inclusive
TRUE - card up to specified depth in stack of visible cards in grid box (x, y)
follows rule
FALSE - no card up to specified depth in stack of visible cards follows rule

Rules in Conditionals can only be of the [] case because Contains, AtDepth,
```

```
and UpToDepth only test an exact match of one card.
 */

import java.util.Vector;

public class UpToDepth implements Conditional
{
    Grid theBigGrid;
    CardStack[][] theGrid;  //the grid
    int x, y;  //coordinates of grid box
    Rule rule;  //rule to find a matching card by
    int depth;  //depth to find the matching card at
    Deck deck;

    public String id;

    //constructor
    public UpToDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
    {
        theBigGrid=grid;
        this.theGrid=grid.theGrid;
        x=xCoord;
        y=yCoord;
        rule=r;
        depth=dp;
        deck=d;

        id="UpToDepth";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        //find index of card at specified depth
        int s=theGrid[x][y].size();
        int index=s-depth;

        //if index is out of range, check entire stack
        if(index<0)
            index=0;

        //check cards up to the index (of specified depth)
        for(int i=s-1;i>=index;i--)
            {
                Card c=theGrid[x][y].elementAt(i);

                //if card is visible, check it for rule match
                if(c.VISIBLE)
                    {
                        Vector v=new Vector();
                        v.add(c);

                        //if rule finds a match, return true
                        if(rule.isValid(v, deck))
                            return true;
                    }

                //if card is not visible, no more visible cards, return false
                else
                    return false;
            }

        //no match found, return false
        return false;
    }

    //flip conditional to be the opposite:  NotUpToDepth
    public Conditional flip()
    {
        NotUpToDepth c=new NotUpToDepth(theBigGrid, x, y, rule, depth, deck);
        return c;
```

```
    }

    //for printing
    public String toString()
    {
        return id;
    }
}
```

8.2.22 NotUpToDepth.java

```
NotUpToDepth.java
/* LOG
 *================================
 * Last Modified: 5.9.03 by cheryl
 *
 * added flip() method
 *
 * -----
 * 5.7.03
 *
 * wrote check() method
 *
 * -----
 * 5.5.03
 *
 * created the class interface
 *
 */

/* NotUpToDepth Methods
 *
 *    public boolean check()
 *
 */

/*
NotUpToDepth
------------
returns opposite of UpToDepth

UpToDepth
---------
depth is measured from top of stack
"up to depth" is inclusive
TRUE - card up to specified depth in stack of visible cards in grid box (x, y) follows
rule
FALSE - no card up to specified depth in stack of visible cards follows rule

Rules in Conditionals can only be of the [] case because Contains, AtDepth, and UpToDepth
only test an exact match of one card.
 */

import java.util.Vector;

public class NotUpToDepth implements Conditional
{
    Grid theGrid;  //the grid
    int x, y;  //coordinates of grid box
    Rule rule;  //rule to find a matching card by
    int depth;  //depth to find the matching card at
    Deck deck;

    public String id;

    //constructor
    public NotUpToDepth(Grid grid, int xCoord, int yCoord, Rule r, int dp, Deck d)
    {
        this.theGrid=grid;
        x=xCoord;
        y=yCoord;
```

```
        rule=r;
        depth=dp;
        deck=d;

        id="NotUpToDepth";
    }

    //check to see if conditional is satisfied
    public boolean check()
    {
        UpToDepth conditional=new UpToDepth(theGrid, x, y, rule, depth, deck);
        return !conditional.check();
    }

    //flip conditional to be the opposite:  UpToDepth
    public Conditional flip()
    {
        UpToDepth c=new UpToDepth(theGrid, x, y, rule, depth, deck);
        return c;
    }

    //for printing
    public String toString()
    {
        return id;
    }
}
```

8.2.23 ActionConditional.java

ActionConditional.java

```
/* LOG
 *=================================
 * Last Modified: 5.6.03 by cheryl
 *
 *
 * -----
 * 5.6.03
 *
 * created the class
 *
 */

/* ActionConditional Methods
 *
 * public ActionConditional(Conditional c, Action a, Grid g)
 *
 * public void addAction(Action a)
 *
 * public void doValid()
 *
 */

/*
An ActionConditional is made of an Action and a Conditional.  If the conditional is
satisfied, do the associated action.
*/

import java.util.Vector;

public class ActionConditional
{
    public Conditional conditional;
    public Vector actionVector;

    private Grid theGrid;  //the grid, in order to do actions

    //constructor
    //one conditional, start with one action
    public ActionConditional(Conditional c, Action a)
    {
```

```
        conditional=c;
        actionVector=new Vector();
        actionVector.add(a);
    }

    //add an action to the action vector
    public void addAction(Action a)
    {
        actionVector.add(a);
    }

    //check Conditional, if true, do associated actions
    public void doValid(Grid g)
    {
        theGrid =g;
        //if conditional is satisfied, do actions
        if(conditional.check())
            {
                //loop through actionVector
                for(int i=0;i<actionVector.size();i++)
                    {
                        //get Action
                        Action a=((Action)actionVector.elementAt(i));
                        //do the action
                        a.doAction(theGrid);
                    }
            }

        //conditional not satisfied, do nothing
    }
}
```

8.2.24 Action.java

Action.java

```
/* LOG
 *================================
 * Last Modified: 5.6.03 by cheryl
 *
 *
 * -----
 * 5.6.03
 *
 * created the class
 *
 */

/* Action Methods
 *
 *
 * doAction(Grid theGrid)
 *
 *
 *
 *
 */

/*
Action
------
MOVE
MOVESTAR
SHUFFLE
REVERSE
SETDEPTH
*/

import java.util.Vector;

public class Action
```

```
{
    String description=null;
    int srcX=0;
    int srcY=0;
    int srcNum=0;
    boolean srcView=true;
    boolean sVIEW=true;
    boolean sNOVIEW=true;
    int destX=0;
    int destY=0;
    int destVIEW=0;
    int destNOVIEW=0;
    boolean dVIEW=true;
    boolean dNOVIEW=true;
    Grid thatGrid;

  /**
   * Constructor for a MOVE action
   */
    public Action(String d, int sX, int sY, int sN, boolean sV, int dX, int dY, int dV,
int dNV)
    {
        description = d;
        srcX= sX;
        srcY=sY;
        srcNum=sN;
        srcView=sV;
        destX=dX;
        destY=dY;
        destVIEW=dV;
        destNOVIEW=dNV;
    }

        /**
         * Constructor for a MOVESTAR action
         */
    public Action(String d, int sX, int sY, boolean sV, boolean sNV, int dX, int dY,
boolean dV, boolean dNV){
        description =d;
        srcX= sX;
        srcY=sY;
        sVIEW=sV;
        sNOVIEW=sNV;
        destX=dX;
        destY=dY;
        dVIEW=dV;
        dNOVIEW=dNV;
    }

        /**
         * Constructor for a REVERSE action
         */
    public Action(String d, int x, int y, int v, int nv)
    {
        description = d;
        destX=x;
        destY=y;
        destVIEW=v;
        destNOVIEW=nv;
    }

        /**
         * Constructor for a SHUFFLE action
         */
    public Action(String d, int x, int y)
    {
        description = d;
        destX=x;
        destY=y;
    }
```

```java
        /**
         * Constructor for a SETDEPTH action
         */
    public Action(String d, int x, int y, int depth)
    {
        description = d;
        srcX = x;
        srcY = y;
        srcNum = depth;
    }

        /**
         * Perform the Action object's action
         */
    public void doAction(Grid theGrid)
    {
        System.out.println("ACTION called: "+description);
        if( description.equals("MOVE") ){
            theGrid.move(srcX, srcY, srcNum, srcView, destX, destY, destVIEW, destNOVIEW);
        }
        else if( description.equals("MOVESTAR") ){
            theGrid.move(srcX, srcY, sVIEW, sNOVIEW, destX, destY, dVIEW, dNOVIEW);
        }
        else if( description.equals("SHUFFLE") ){
            theGrid.shuffle(destX, destY);
        }
        else if( description.equals("REVERSE") ){
            theGrid.reverse(destX, destY, destVIEW, destNOVIEW);
        }
        else if( description.equals("SETDEPTH") ){
            theGrid.setDepth(srcX, srcY, srcNum);
        }
    }
}
```

8.2.25 Crème.g

```
//-----------------------------------------------------------------------
// Creme.g
// contains: Lexer, Parser, TreeParser
// author: Josh Mackler      jrm267
//-----------------------------------------------------------------------


class P extends Parser;
options { buildAST = true;}


tokens{
START;
GLOC;         // grid location
VIS;          // grid visibility
BOOLEAN;      // if conditional
CONDITION;    // conditional statement
CARD;
DIMENSIONS;   // int containing dimension number
RULE;
}

run
: (startRule)+ EOF! {#run = #([START,"start"], run);}
;


startRule
: deckCall
```

```
  | turn
  | // initialize Grid
    "Grid"^ (objectCall| EQUAL! LFTSQ! DIGIT COMMA! DIGIT RTSQ! SEMI!)
;

deckCall
: "Deck"^ PERIOD! (rankSys|deal)
;

deal!
: d:"deal" TO! (g:"Grid" gs:grid v:vis) SEMI!
// special tree structure
{#deal = #(d, #(g, gs,v));}
;


objectCall
: (grid PERIOD! (move|depth));

depth
: "setDepth"^ LFTPAREN! DIGIT RTPAREN! SEMI!
;

move
: "move"^ vis TO! ("Grid" grid) vis SEMI!
;

turn
: "turn"^ PERIOD! turnCond
  LFTBR! (conditional)* RTBR!
;

turnCond // multiple Grid conditionals
:  DIGIT^ LFTPAREN! ("Grid" grid booleanCheck (COMMA!)?)* RTPAREN!
;

grid  // can specify multiple grid spots
: LFTSQ! DIGIT (DASH DIGIT)? COMMA! DIGIT
 (DASH DIGIT)? RTSQ!
{  #grid = #([GLOC,"location"],  grid);}
;

rules // can specify single or multiple rules
: ("place"^|"take"^)   ((card) => card
       | (LFTPAREN! cardRule  RTPAREN!)
       | LFTSQ!  ((LFTPAREN! cardRule  RTPAREN!) |c:card
        )+ RTSQ!)
       SEMI!
;

conditional
:! f:"if"! {#f.setType(CONDITION);} bc:booleanCheck t:"then" LFTBR!
 (c1:conditional|a1:action)
  (options {greedy = true;}:e:"else" LFTBR! (c2:conditional|a2:action) RTBR!)?
RTBR!
{#conditional = #(f, bc , #(t,c1, a1) ,#(e,c2, a2) );}
| "Grid"^ grid PERIOD! "rules" TO! conditional
| rules
;

booleanCheck
:  (EXCLAIM|PERIOD)?
   ("isEmpty"| "isTotallyEmpty" | "contains" card
```

```
 | ("atDepth"|"upToDepth") LFTPAREN! DIGIT
   RTPAREN! card) {#booleanCheck = #([BOOLEAN,"boolean"], booleanCheck);}
;

action
:"action"^ LFTPAREN! (theAction)* RTPAREN! SEMI!
;

theAction
:"Grid"^ grid PERIOD! (move|("shuffle"|"setDepth" DIGIT|"reverse"
  (DIGIT|STAR) COMMA! (DIGIT|STAR)) SEMI!)
;

cardRule
: ruleSpec (COMMA! ruleSpec)*
 {#cardRule = #([RULE,"rule"], cardRule);}
;

ruleSpec
:  ((DASH|PLUS)? DIGIT^ (EQUAL)?)|STAR
;

vis
: LFTPAREN! (DIGIT|STAR) COMMA! (DIGIT|STAR)  RTPAREN!
 {#vis = #([VIS,"vis"], vis);}
;

rankSys
:"Ranksys"^  // heterogeneous type DIMENSTIONS set
 (PERIOD! DIGIT EQUAL! inRank| EQUAL! d:DIGIT {#d.setType(DIMENSIONS);} SEMI!)
;

inRank // part of Ranksys
: LFTPAREN! defRank (options {greedy = true;}:( GTR^ | PERIOD^ )
 defRank)* RTPAREN! SEMI!
;

defRank // ranking definition made
: ID^ (CARROT DIGIT)? (LFTSQ! map (COMMA! map)* RTSQ!)?
;

map // map of ranking
:DIGIT^ (PERIOD|EXCLAIM) LFTPAREN! ID (COMMA! ID)* RTPAREN!
;


seq
: ("seq"^|"cseq"^) CARROT! DIGIT
;

card
:LFTSQ!(ID (EQUAL)? |STAR)(COMMA!(ID  (EQUAL)? |STAR))* RTSQ!
{#card = #([CARD,"card"],card);}
;

class L extends Lexer;

options{
k=1; // only 1 character lookahead
charVocabulary='\3'..'\377';
testLiterals=false; // set true for IDS
}
```

```
DIGIT
: ('0'..'9')+
;

PERIOD:     '.' ;
LFTPAREN:   '(' ;
RTPAREN:    ')' ;
CARROT:     '^' ;
GTR:        '>' ;
LESS:       '<' ;
EQUAL:      '=' ;
COMMA:      ',' ;
EXCLAIM:    '!' ;
LFTSQ:      '[' ;
RTSQ:       ']' ;
SEMI:      ';' ;
STAR:       '*' ;
TO:        '~' ;
DASH:       '-' ;
PLUS:       '+' ;
LFTBR:     '{' ;
RTBR:       '}' ;


WS // white space skipped
: (' '
| NEWLINE
| '\t'
)
{$setType(Token.SKIP);}
;

protected
NEWLINE
:('\r' '\n')=>'\r''\n' {newline();}
| '\n' {newline();}
| '\r' {newline();}
;


COMMENT // comments skipped
:'/' (('*') => '*' (options {greedy=false;}:
       ( NEWLINE
       | (~('\n'|'\r')))))* "*/"
       | '/' (~('\n'|'\r'))* NEWLINE
)
{$setType(Token.SKIP);}
;


ID options { testLiterals = true; }
: ( 'a'..'z'|'A'..'Z')  ( 'a'..'z'|'_'|'A'..'Z'|'0'..'9')*
;



{import java.util.*;
 import java.lang.*;


}
```

```
class CremeTreeWalker extends TreeParser;
// private declarations for Global scope
{private Vector[] ranksysArray;
 private int dimensions;
 private QuadraticProbingHashTable[] labels;
 private MyHashString label;
 private Grid grid;
 private Deck myDeck;
 private Vector mapping = new Vector();
 private int width, length;
 private RuleGrid myTakeGrid, myPlaceGrid;
 private ActionGrid myActionGrid;
 private Vector turnVector = new Vector();
 private Vector turnCondVect;
}

run
:  #(START (object)+)
{// done at very end, after whole tree is walked
 grid.setTurn(turnVector);}
;

object
: deck
| grid
| turn
;

grid{Action tempAct;}
: #("Grid" (d1:DIGIT d2:DIGIT
{ // initialize Grid
  width = Integer.parseInt(d1.getText());
  length = Integer.parseInt(d2.getText());
  grid = new Grid(width, length);
}
|#(GLOC f1:DIGIT (DASH f3:DIGIT)? f2:DIGIT (DASH f4:DIGIT)?)
{// only from1 and from2 used for moving, from3 and from4 ignored
 int from1,from2,from3,from4,dep;
 from1 = Integer.parseInt(f1.getText());
 if(f3!=null){ from3 = Integer.parseInt(f3.getText());}
 else{from3 = from1;}
 from2 = Integer.parseInt(f2.getText());
 if(f4!=null){ from4 = Integer.parseInt(f4.getText());}
 else{from4 = from2;}
}
 (tempAct=move[from1, from2, false] |
  #("setDepth" d3:DIGIT // depth of how many cards can be taken at once
 {     dep = Integer.parseInt(d3.getText());
       for (int i = from1; i <= from3; i++){
          for (int j = from2; j <= from4; j++){
              grid.setDepth(i,j,dep);
          }
       }
})
)))
;

move[int f1, int f2, boolean action] returns [Action act]
{ act = new Action("REVERSE", 0,0,0,0); // initialize action
}
: #("move" #(VIS (v1:DIGIT|s1:STAR)
 (v2:DIGIT|s2:STAR))
 "Grid" #(GLOC t1:DIGIT t2:DIGIT)
```

```
#(VIS (v3:DIGIT|s3:STAR) (v4:DIGIT|s4:STAR)))
{int vis1=0, vis2=0, vis3=0, vis4=0, from1, from2, to1, to2, srcNum=0;

 boolean srcVIEW=false;
 boolean star1=false, star2=false, star3=false, star4=false;
if(v1!=null)
 vis1 = Integer.parseInt(v1.getText());
if(v2!=null)
 vis2 = Integer.parseInt(v2.getText());
if(v3!=null)
 vis3 = Integer.parseInt(v3.getText());
if(v4!=null)
 vis4 = Integer.parseInt(v4.getText());
 from1 = f1;
 from2 = f2;
 to1 = Integer.parseInt(t1.getText());
 to2 = Integer.parseInt(t2.getText());

// star move
if(s1!=null || s2!=null || s3!=null || s4!=null){
  if(s1!=null)star1=true;
  if(s2!=null)star2=true;
  if(s3!=null)star3=true;
  if(s4!=null)star4=true;
  if((star1==false && vis1!=0) || (star2==false && vis2!=0)
      || (star3==false && vis3!=0) || (star4==false && vis4!=0))
     { System.err.println("A star move must have zeros where there are not
stars");
       System.exit(0);
     }
  else{ if(action==true){// returned for action conditional
                        act = new Action("MOVESTAR",from1, from2,star1,star2,
                            to1,to2,star3,star4);}
                    else{// non-action
                    grid.move(from1,from2,star1,star2,to1,to2,star3,star4);}}
}

// non-star move
else{
 if (vis2==0){srcNum=vis1; srcVIEW=true;}
 else {if (vis1==0){srcNum=vis2;}
 else{ System.err.println("In move call, one of the visibility parameters "
                    + "in the Grid the move is from must be set to zero");

       System.exit(0);}}

    //check to make sure within grid parameters
       grid.gridCheck(0,to1);
       grid.gridCheck(0,from1);
       grid.gridCheck(1,to2);
       grid.gridCheck(1,from2);
if(action==true){
 // returned for action conditional
 act = new Action("MOVE",from1, from2, srcNum,srcVIEW,to1, to2, vis3, vis4);}
else{
 // non-action
 grid.move(from1, from2, srcNum, srcVIEW, to1, to2, vis3, vis4);}
}
 }

;

deck
```

```
: #("Deck" (ranksys|deal))
;


deal
{
    // Deck is created with first deal call.
    // Ranksys is checked to make sure it has been declared.
       if(myDeck == null){
       for (int i = 0; i< dimensions; i++){
       if(ranksysArray[i].isEmpty())
       {
              System.err.println("Ranksys " + (i+1) +
               " unspecified. Please initialize.");
              System.exit(0);
       }}
       myDeck = new Deck(ranksysArray);
Map mapObj;

for (int i = 0; i < mapping.size() ; i++){
       mapObj = (Map) mapping.get(i);
       int[] intMap = mapObj.getDim();
       String[] stringMap = mapObj.getLabels();
       int mapDim = mapObj.getMapDim();


       for (int j = 0; j < intMap.length; j++){
              if(intMap[j] < 0 || intMap[j] == mapDim ||
               (intMap[j]+1) > dimensions){
                     System.err.println("Rank mapping number "
                      + (intMap[j] + 1) + " used for mapping on " +
                      " is incorrect. Please try a different number" );
                     System.exit(0);}
              if(labels[intMap[j]].find(new MyHashString(stringMap[j]))
               == null)
              {
             // Check symbol table for mapping label
                     System.err.println("Name " + stringMap[j]
                      + " can not be mapped since it does not exist in Ranksys "
                      + (intMap[j]+1));
                     System.exit(0);
              }

       }
       mapObj.doMap(myDeck);

}

// RuleGrids declared here
myDeck.createCardStack();
myTakeGrid = new RuleGrid(width, length, myDeck);
myPlaceGrid = new RuleGrid(width, length, myDeck);
myActionGrid = new ActionGrid(width, length, grid);
turnCondVect = new Vector();
}

}
:#("deal" #(gr:"Grid" #(GLOC g1:DIGIT (DASH g3:DIGIT)? g2:DIGIT
        (DASH g4:DIGIT)?) #(VIS (s1:STAR|v1:DIGIT) (s2:STAR|v2:DIGIT))
{ int vis1, vis2, y1, y2, x1, x2;
  boolean visible=false;

  // regular deal
```

```
  if (s1 == null && s2 == null)
       {
       vis1 = Integer.parseInt(v1.getText());
       vis2 = Integer.parseInt(v2.getText());
       x1 = Integer.parseInt(g1.getText());
       y1 = Integer.parseInt(g2.getText());
       if(g3 != null)
              x2 = Integer.parseInt(g3.getText());
       else{ x2 =x1;}
       if(g4 != null)
              y2 = Integer.parseInt(g4.getText());
       else{ y2 = y1;}

    // verifies grid parameters
       grid.gridCheck(0,x1);
       grid.gridCheck(0,x2);
       grid.gridCheck(1,y1);
       grid.gridCheck(1,y2);

       for(int i = x1; i <= x2; i++){
              for (int j = y1; j <= y2; j++){
                     for(int k = 0; k < vis1; k++){
                            grid.put(i,j, myDeck.deal(), true);
                     }
                     for(int k = 0; k < vis2; k++){
                            grid.put(i,j, myDeck.deal(), false);
                     }
              }
       }
       }
    // deal full deck to a location
    else{
              x1 = Integer.parseInt(g1.getText());
           y1 = Integer.parseInt(g2.getText());
        if(s1!=null){
          if(s2!=null){
               System.err.println("Only one visiblity spot may have a star " +
                              "for moving a whole deck to Grid at " + x1 + " "
                              + y1);
               System.exit(0);
          }
          else{visible = true;}}
        grid.gridCheck(0,x1);
           grid.gridCheck(0,y1);
        grid.put(x1,y1,myDeck,visible);
}}
))
;

ranksys
{int n;}

: #("Ranksys"  (dim:DIMENSIONS // initialize Ranksys
{      dimensions = Integer.parseInt(dim.getText());
       System.out.println ("Dimensions number: " + dimensions);
       ranksysArray = new Vector[dimensions];
       labels = new QuadraticProbingHashTable[dimensions];
       for (int i=0; i < dimensions; i++){
              ranksysArray[i] = new Vector();
              labels[i] = new QuadraticProbingHashTable();
       }
}
```

```
|d1:DIGIT
 {      n = Integer.parseInt(d1.getText());
        n--; //to agree with array
        System.out.println ("Ranksys number: " + n);}
    rank[ranksysArray, n, '>']
{ labels[n].printTable();}
))
;

rank[Vector[] ranksysArray, int n, char c] // inside Ranksys, pass > or .
{
        String name = "";
        int multi;

}
: #(GTR rank[ranksysArray, n, '>'] #(card:ID (CARROT mult:DIGIT)?
(map[card.getText(),n])*))

        {
        name = card.getText() + c;
    // label added to HashTable
        label = new MyHashString(card.getText());
        labels[n].insert(label);

    System.out.println(name);

        if (mult != null)
        { multi = Integer.parseInt(mult.getText());
        for(int i = 0; i<multi; i++){
               ranksysArray[n].addElement(name);
        }}

        else{
        ranksysArray[n].addElement(name);
        }}

| #(PERIOD rank[ranksysArray,n,'.']  #(card2:ID (CARROT mult2:DIGIT)?
(map[card2.getText(),n])*))

        { name = card2.getText() + c;
        label = new MyHashString(card2.getText());
        // label added to HashTable
        labels[n].insert(label);
    System.out.println(name);
        if (mult2 != null)
        { multi = Integer.parseInt(mult2.getText());
        for(int i = 0; i<multi; i++){
               ranksysArray[n].addElement(name);
        }}

        else{
        ranksysArray[n].addElement(name);
        }

        }

|  #(card3:ID (CARROT mult3:DIGIT)? (map[card3.getText(),n])*)

        {name = card3.getText() + c;
        label = new MyHashString(card3.getText());
    // label added to HashTable
        labels[n].insert(label);
    System.out.println(name);
```

```
        if (mult3 != null)
        { multi = Integer.parseInt(mult3.getText());
        for(int i = 0; i<multi; i++){
               ranksysArray[n].addElement(name);
        }}

        else{

        ranksysArray[n].addElement(name);
        }}
;

map[String card, int n]
{String name="";
int rankMap;
Vector theInts = new Vector();
Vector theLabels = new Vector();
boolean save = true;
}
: #(d:DIGIT (exc:EXCLAIM|per:PERIOD) (cardN:ID
{
        name = cardN.getText();
        rankMap = Integer.parseInt(d.getText());
        theInts.addElement(new Integer(rankMap));
        theLabels.addElement(name);


} )*)
{
    Object[] intObj = (theInts.toArray());
    Object[] stringObj = (theLabels.toArray());
    int [] intArray = new int[intObj.length];
    String[] stringArray = new String[stringObj.length];
    for (int i = 0; i < intObj.length; i++){
         intArray[i] = ((Integer)intObj[i]).intValue()-1;
         stringArray[i] = (String) stringObj[i];
    }
    if(exc != null)
         save=false;

    mapping.addElement(new Map(intArray, stringArray, n, card, save));
    }
;

turn
{ int x1, x2, y1, y2;
}
: #("turn" turnCond
  (#(gr:"Grid" #(GLOC g1:DIGIT (DASH g3:DIGIT)? g2:DIGIT
         (DASH g4:DIGIT)?)

{
        x1 = Integer.parseInt(g1.getText());
        y1 = Integer.parseInt(g2.getText());
        if(g3 != null)
               x2 = Integer.parseInt(g3.getText());
        else{ x2 =x1;}
        if(g4 != null)
               y2 = Integer.parseInt(g4.getText());
        else{ y2 = y1;}

        grid.gridCheck(0,x1);
```

```
        grid.gridCheck(0,x2);
        grid.gridCheck(1,y1);
        grid.gridCheck(1,y2);

        g1=null; g2=null; g3 =null; g4=null;
}

 "rules" (#("place"(rule[x1,x2,y1,y2,true])*)
 | #("take" (rule[x1,x2,y1,y2,false])*)
 | (cond[x1,x2,y1,y2])*
 )))*)

{Turn turn = new Turn(turnCondVect,myTakeGrid,myPlaceGrid,myActionGrid);
turnVector.add(turn);
myTakeGrid = new RuleGrid(width, length, myDeck);
myPlaceGrid = new RuleGrid(width, length, myDeck);
myActionGrid = new ActionGrid(width, length, grid);
turnCondVect = new Vector();
}
;

turnCond:
{ int x1, x2, y1, y2, turnNum; Vector vcond;}
#(tN:DIGIT (gr:"Grid" #(GLOC g1:DIGIT (DASH g3:DIGIT)? g2:DIGIT
        (DASH g4:DIGIT)?)

{       vcond = new Vector();
        turnNum = Integer.parseInt(tN.getText());
        x1 = Integer.parseInt(g1.getText());
        y1 = Integer.parseInt(g2.getText());
        if(g3 != null)
               x2 = Integer.parseInt(g3.getText());
        else{ x2 =x1;}
        if(g4 != null)
               y2 = Integer.parseInt(g4.getText());
        else{ y2 = y1;}

        grid.gridCheck(0,x1);
        grid.gridCheck(0,x2);
        grid.gridCheck(1,y1);
        grid.gridCheck(1,y2);

        g1=null; g2=null; g3 =null; g4=null;
} vcond=bool[x1,x2,y1,y2]

{for(int x = 0; x < vcond.size(); x++){
turnCondVect.add(vcond.get(x));
}}

 )*)
;

rule[int i, int j, int k, int l, boolean place]
{int ruleDigit, dimNum = 0;
Rule rule = new Rule(dimensions);}
:
(#(CARD ((id:ID (eq2:EQUAL)?|st:STAR)
{
        System.out.println("["+i+"-"+j+"]"+"["+k+"-"+l+"]");
        if(id!=null){
                  if(labels[dimNum].find(new MyHashString(id.getText()))
                      == null)
                  {
```

```
                    System.err.println("Name " + id
                    + " can not be mapped since it does not exist in Ranksys "
                        + (dimNum+1));
                    System.exit(0);
                }


            if(eq2 !=null){
                    rule.setExactValueEq(dimNum, id.getText());
            }
            else{rule.setExactValue(dimNum, id.getText());}

        }
        dimNum++;
        id=null;
}
  )*)

| #(RULE ((star:STAR|#(d1:DIGIT (dash:DASH| plus:PLUS| eq:EQUAL)?))
{       if(d1 != null){
                 ruleDigit = Integer.parseInt(d1.getText());

            if(dash != null){
                     rule.setMinus(dimNum,ruleDigit);
            }
            else if(plus != null){
                     rule.setPlus(dimNum,ruleDigit);
            }
            else if(eq != null){
                     rule.setMultipleMatchEq(dimNum,ruleDigit);
            }
            else{ rule.setMultipleMatch(dimNum,ruleDigit);}
         }
        dimNum++;
        dash=null;
        plus=null;
        eq=null;
        d1=null;
}
)*))


{ if(dimNum!=dimensions)
              {
                  System.err.println("Incorrect number of dimension parameters "
                      + "for Rule declaration. " + dimNum+ " used, " +
                  dimensions + " needed.");
                  System.exit(0);
              }
        for(int wid = i; wid <= j; wid++){
            for (int len = k; len <= l; len++){
             if(place==true){
                        myPlaceGrid.setRule(rule,wid,len);
                    }
             else{myTakeGrid.setRule(rule,wid,len);}
            }
        }
}
;


bool[int i, int j, int k, int l] returns [Vector cRules]
{Conditional cond1;
```

```
 Rule rule = new Rule(dimensions);
 Rule crule;
 int dimNum = 0, cdepth=0;
 cRules = new Vector();}
:#(BOOLEAN  (exc:EXCLAIM|PERIOD)? (iE:"isEmpty"| iT:"isTotallyEmpty" |(
 (c:"contains" | ((a:"atDepth"|u:"upToDepth") dig:DIGIT
 {cdepth = Integer.parseInt(dig.getText());}))
 (#(CARD ((id:ID (eq2:EQUAL)?|st:STAR)*
 {
    //create card rule here
       if(id!=null){
                if(labels[dimNum].find(new MyHashString(id.getText()))
                    == null)
                {   System.err.println("Name " + id
                    + " can not be mapped since it does not exist in Ranksys "
                        + (dimNum+1));
                    System.exit(0);                 }
             if(eq2 !=null){
                    rule.setExactValueEq(dimNum, id.getText());           }
             else{rule.setExactValue(dimNum, id.getText());}
       }
       dimNum++;
       id=null;
}
))))))

// create a Vector of rules with conditions bound to them according to
appropriate
// child of the Boolean tree. Multiple grid locations can be done at once.
{      for(int wid = i; wid <= j; wid++){
             for (int len = k; len <= l; len++){
            if(iE!=null){
                        if(exc!=null){
                            cond1=new NotIsEmpty(grid,wid,len);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("NotIsEmpty");
                            cRules.addElement(crule);
                        }
                        else{
                            cond1=new IsEmpty(grid,wid,len);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("IsEmpty");
                            cRules.addElement(crule);
                    }}
            else if(iT!=null){
                        if(exc!=null){
                            cond1 = new NotIsTotallyEmpty(grid,wid,len);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("NotIsTotallyEmpty");
                            cRules.addElement(crule);
                        }
                        else{
                            cond1 = new IsTotallyEmpty(grid,wid,len);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("IsTotallyEmpty");
                            cRules.addElement(crule);
                    }}
            else if (c!=null){
                        if(exc!=null){
                            cond1 = new NotContains(grid,wid,len,rule,myDeck);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("NotContains");
                            cRules.addElement(crule);
```

```
                        }
                        else{
                            cond1 = new Contains(grid,wid,len,rule,myDeck);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("Contains");
                            cRules.addElement(crule);
                    }}
            else if (a!=null){
                        if(exc!=null){
                            cond1 = new NotAtDepth(grid,wid,len,rule,
                                cdepth, myDeck);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("NotAtDepth");
                            cRules.addElement(crule);
                        }
                        else{
                            cond1 = new AtDepth(grid,wid,len,rule, cdepth,
myDeck);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("AtDepth");
                            cRules.addElement(crule);
                    }}
            else if (u!=null){
                        if(exc!=null){
                            cond1 = new NotUpToDepth(grid,wid,len,rule,
                                cdepth, myDeck);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("NotUpToDepth");
                            cRules.addElement(crule);
                        }
                        else{
                            cond1 = new UpToDepth(grid,wid,len,rule,
                                cdepth, myDeck);
                            crule = new Rule(dimensions, cond1);
                            System.out.println("UpToDepth");
                            cRules.addElement(crule);
                        }}
                }
}}
;

cond[int i, int j, int k, int l]
{ Vector cRules;
 int dimNum = 0, cdepth=0;
}
: #(CONDITION  cRules=bool[i,j,k,l]

            (( #("then" (#("place"
            // redeclare the vector of rules with a new rules
            {Vector pass1 = new Vector();
              for (int x = 0; x < cRules.size(); x++){
                    Rule tempR1 = (Rule) cRules.get(x);
                    Rule newRule1 = new Rule(dimensions, tempR1.conditional);
                    pass1.add(newRule1);}
 }
            (crule[i,j,k,l,pass1,true])*)
            | #("take"
             // redeclare the vector of rules with a new rules
            {Vector pass2 = new Vector();
              for (int x = 0; x < cRules.size(); x++){
                    Rule tempR2 = (Rule) cRules.get(x);
                    Rule newRule2 = new Rule(dimensions, tempR2.conditional);
                    pass2.add(newRule2);}
```

```
                  }

            (crule[i,j,k,l,pass2,false] )*)
            |#("action"
             // redeclare the vector of rules with a new rules
              {Vector pass3 = new Vector();
              for (int x = 0; x < cRules.size(); x++){
                    Rule tempR3 = (Rule) cRules.get(x);
                    Rule newRule3 = new Rule(dimensions, tempR3.conditional);
                    pass3.add(newRule3);}
                  }



            action[i,j,k,l,pass3]))*)))
      (#("else" {cRules=FlipRule.flip(cRules);}

            (#("place"
            // redeclare the vector of rules with a new rules
            {Vector pass4 = new Vector();
                for (int x = 0; x < cRules.size(); x++){
                    Rule tempR4 = (Rule) cRules.get(x);
                    Rule newRule4 = new Rule(dimensions, tempR4.conditional);
                    pass4.add(newRule4);}
                  }


            (crule[i,j,k,l,pass4,true])*)
            | #("take"

           {Vector pass5 = new Vector();
                    for (int x = 0; x < cRules.size(); x++){
                    Rule tempR5 = (Rule) cRules.get(x);
                    Rule newRule5 = new Rule(dimensions, tempR5.conditional);
                    pass5.add(newRule5);
                  }
            }

            (crule[i,j,k,l,pass5,false])*)
             | #("action"
              // redeclare the vector of rules with a new rules
              {Vector pass6 = new Vector();
                    for (int x = 0; x < cRules.size(); x++){
                    Rule tempR6 = (Rule) cRules.get(x);
                    Rule newRule6 = new Rule(dimensions, tempR6.conditional);
                    pass6.add(newRule6);
                  }
            }


             action[i,j,k,l,pass6]))*))?
    )
;

action[int i, int j, int k, int l, Vector rules]
{
Action act= new Action("REVERSE",0,0,0,0); int depth, rev1, rev2, from1, from2;

if (i!=j || k!=l){
        System.err.println("Actions can only be specified to one grid spot "
            + "at a time");
        System.exit(0);
}}
```

```
: (#("Grid" #(GLOC f1:DIGIT f2:DIGIT)
                {from1 = Integer.parseInt(f1.getText());
                    from2 = Integer.parseInt(f2.getText());
                }
  (act=move[from1,from2,true]
  | ("shuffle"
        {act = new Action("SHUFFLE",from1,from2);})
  |( "setDepth" d:DIGIT
        {   depth = Integer.parseInt(d.getText());
            act = new Action("SETDEPTH",from1,from2,depth);})
  |("reverse"
        (d1:DIGIT|s1:STAR) (d2:DIGIT|s2:STAR)
        {
            if(s1!=null){
                rev1=-1;}
            else{rev1=Integer.parseInt(s1.getText());
                if(rev1!=0){System.err.println("To reverse, the" +
                        "parameters must be a star or a zero");}
            }
            if(s2!=null){
                rev2=-1;}
            else{rev2=Integer.parseInt(s1.getText());
                if(rev2!=0){System.err.println("To reverse, the" +
                        "parameters must be a star or a zero");}
            }
            act = new Action("REVERSE",from1,from2,rev1,rev2);}
    ))))
{for (int x =0; x <rules.size(); x++){
                Rule rule = (Rule) rules.get(x);
                ActionConditional acc = new ActionConditional(
                    rule.conditional, act);
                myActionGrid.setActionConditional(acc, i, k);
            }}
;

crule[int i, int j, int k, int l, Vector rules, boolean place]
{int ruleDigit, dimNum = 0;
 for (int x = 0; x < rules.size(); x++){
      Rule r = (Rule)(rules.get(x));
      Rule rule = new Rule(dimensions, r.conditional);
      rules.set(x,rule);
 }
}
:
(#(CARD ((id:ID (eq2:EQUAL)?|st:STAR)
{
    System.out.println("At Card: DimNum " + dimNum + " and rules size " +
rules.size());

       System.out.println("Cond Rule card: ["+i+"-"+j+"]"+"["+k+"-"+l+"]");
       if(id!=null){

                if(labels[dimNum].find(new MyHashString(id.getText()))
                    == null)
                {
                    System.err.println("Name " + id
                    + " can not be mapped since it does not exist in Ranksys "
                        + (dimNum+1));
                    System.exit(0);
                }


if(eq2 !=null){
```

```
    for (int x = 0; x < rules.size(); x++){
          Rule rule = (Rule)(rules.get(x));

          System.out.println("setExactEq");
          rule.setExactValueEq(dimNum, id.getText());
          rules.set(x,rule);
    }
}
else{
   for (int x = 0; x < rules.size(); x++){
          Rule rule = (Rule)(rules.get(x));
          System.out.println("setExact");
          rule.setExactValue(dimNum, id.getText());
          rules.set(x,rule);
}
}


       }

       dimNum++;
    eq2=null;
       id=null;
}
  )*)

| #(RULE ((star:STAR|#(d1:DIGIT (dash:DASH| plus:PLUS| eq:EQUAL)?))
{   System.out.println("Cond Rule rule: ["+i+"-"+j+"]"+"["+k+"-"+l+"]");
 System.out.println("At Rule: DimNum " + dimNum + " and rules size " +
rules.size());
       if(d1 != null){
              ruleDigit = Integer.parseInt(d1.getText());

       if(dash != null){
        for (int x = 0; x < rules.size(); x++){

          Rule rule = (Rule)(rules.get(x));
          rule.setMinus(dimNum,ruleDigit);
          rules.set(x,rule);
          System.out.println("setMinus");
       }
       }
       else if(plus != null){
        for (int x = 0; x < rules.size(); x++){

         Rule rule = (Rule)(rules.get(x));
          rule.setPlus(dimNum,ruleDigit);
          rules.set(x,rule);
          //tempV.addElement(rule);
          System.out.println("setPlus");
          //rules.set(x, ((Rule)(rules.get(x)).setPlus(dimNum,ruleDigit)));
       }
       }
       else if(eq != null){
       for (int x = 0; x < rules.size(); x++){
          Rule rule = (Rule)(rules.get(x));
          rule.setMultipleMatchEq(dimNum,ruleDigit);
          rules.set(x,rule);
          System.out.println("setMultEq");

       }
       }
```

```
        else{
        for (int x = 0; x < rules.size(); x++){

           Rule rule = (Rule)(rules.get(x));
           rule.setMultipleMatch(dimNum,ruleDigit);
           rules.set(x,rule);
             System.out.println("setMulti");

        }
    }
}
        dimNum++;
        dash=null;
        plus=null;
        eq=null;
        d1=null;
}
)*))


{ if(dimNum!=dimensions)
            {
                System.err.println("Incorrect number of dimension parameters "
                    + "for Rule declaration. " + dimNum+ " used, " +
                dimensions + " needed.");
                System.exit(0);
            }
        int count = 0;
    for(int wid = i; wid <= j; wid++){
            for (int len = k; len <= l; len++){
                    System.out.println("Rules size " + rules.size() + " count
is " +
                            count);
                    if(place==true){
                        Rule tempR = (Rule) (rules.get(count));
                        System.out.println(tempR + "\n");
                        myPlaceGrid.setRule(tempR,wid,len);
                    }

                    else{   Rule tempR = (Rule) (rules.get(count));
                             System.out.println(tempR + "\n");
                            myTakeGrid.setRule(tempR,wid,len);}


            count++;
             }
        }
}
;
```

8.2.26 Map.Java

```
Joshua Mackler
public class Map{
    private int [] mapDim;
    private String[] mapName;
    private String name;
    private int dim;
    private boolean keep;
    public Map(int[] mD, String[] mN, int di, String nam, boolean save){
       mapDim = mD;
       mapName = mN;
```

```
      name = nam;
      dim = di;
      keep = save;
   }
   public void doMap(Deck myDeck){
      myDeck.darwinize(mapDim, mapName, dim, name, keep);
   }
   public int[] getDim (){
      return mapDim;
   }

   public String[] getLabels(){
      return mapName;
   }

   public int getMapDim(){
      return dim;
   }

   public boolean getKeep(){
      return keep;
   }
}
```