Joker

a Card Game Programming Language
Final Report

Team Leader

JGE15@columbia.edu Jeffrey Eng

Team Members

HHC42@columbia.edu Howard Chu
TKS21@columbia.edu Timothy SooHoo
JLT93@columbia.edu Jonathan Tse

Document History

2003 December 14 – Document Created

Contents

1 Introduction

1.1 Background

Joker is a simple, portable language for systematically specifying multiplayer turn-based card games using a programmer-specified card deck.

The intention of our language is to allow programmers to succinctly describe the rules of a card game and to create a runtime card game engine. The structure and the rule-driven nature of card games create a ripe domain to construct a language for. Blackjack and War are examples of turn-based games in this domain. This language will allow for the prototyping of card games. Our language creates a framework for programming any turn-based card game.

1.2 Language Characteristics

1.2.1 Simple and Intuitive

Our language is simple and intuitive. The language provides a platform for programmers to easily define and describe the structure, rules, and game flow of a card game. Our language abstracts all turn-based card game concepts and simplifies the programming mechanics involved in coding a card game. For example, the concept of “cards” and “decks” is embedded into the language as fundamental data structures. This allows programmers to focus on specifying the rules and the algorithmic components of the game flow. The syntax is designed to be programmer-friendly and intuitive.

1.2.2 Portable

Our language can run on many different platforms. Our language is compiled into Java™ bytecode which can be run by the Java Virtual Machine, available for virtually any machine.

1.2.3 Memory Managed

Programmers do not need to worry about the complexity involved with memory management. Using Java as our underlying foundation, memory allocation and garbage collection is handled by the system automatically.

1.3 Dynamically Enabled Abstract Language Runtime

The output of our compiler will be a game executable by the Java Virtual Machine. The core of the executable is a rules engine, determining what actions are permissible for what player at what time. The user interface for each player will allow players to perform actions, such as drawing a card from the deck, placing down cards, taking cards from other players, and so forth. Currently, this interface is a simple text-based interface and is scalable to allow expansion to other interface models.

1.4 Language Features

1.4.1 Custom loops

Inherent in the concept of turn-based card games, is, well, the turn. A “turn” or a “round” is typically implemented in other generic languages using a while or for loop. Our language understands the idea of a turn and that it will stop when a winning condition (in addition to other conditions) is met.

1.4.2 Custom data types

Our language has several specialized (and fundamental) data types unique to the domain of card games. For example: the “card” and the “deck”. Cards have values and can be compared to determine dominance. Cards can be grouped, stacked, given, taken, pushed, and popped. Decks are finite card repositories. Decks can be shuffled, have cards added to, have cards removed from.

2 Language Tutorial

2.1 Fire it up

The first thing to know is how to run a Joker program. To run a program, type:

```
$ java JokerMain [joker file name]]
```

This runs your program and begins your game. Let's try the classic hello world program:

```
[ helloworld.jkr ]

game OurFirstGame {
   init  { }

   main {
      print "Hello world!";
   }
}
```

```
$ java JokerMain helloworld.jkr
Hello world!
$
```

Note the required program structure. A Joker program consists of a `game` definition. A `game` definition consists of an `init` block and a `main` block, where statements are placed. These blocks help in structuring your program into declaration and initialization of main variables (`init`) and the main logic execution (`main`).

2.2 The Basics

Joker supports many of the common operators and data types found in other imperative programming languages.

Joker has the standard **data types** such as integers, strings, and Booleans. Joker also has more advanced data types such as cards and packs. **Variables** can be declared, initialized, and later assigned in the following way:

```
int x = 3;
boolean y = true;
string z = "I am a pretty girl."
x = 10;
```

Joker has a `print` statement that supports all the data types:

```
int x = 5;
print x;
```

Joker also supports automatic conversion to strings for the print statement, as well as concatenation of strings. This makes for concise statements such as:

```
string x = "The meaning of life is ";
int y = 42;
print  x + y;
```

When placed in a complete program and run, this outputs:

```
$ java JokerMain sample.jkr
The meaning of life is 42
$
```

Joker supports the traditional conditional and relational operators, as well as many of the regular control flow statements:

```
for (int i = 0; i  < 5 ; i ++ )
       if ( (x == y) && (y > z ) )
              isDone = true;
```

Joker has special data types and control flow statements built for card game logic. The best way to present this more advanced functionality would be to write a small demonstrative game: a form of a silly game called High Card.

2.3 A Case Study: High Card

Joker has a data-type for cards and packs (or "decks", for you Yanks). In order to create cards, you must first define how these cards relate to one another, or what their **hierarchy** is. This is done in a hierarchy declaration.
In a hierarchy declaration, you define a set of rank rules and a set of suit rules. Examples of card ranks are: ace, queen, ten, two. Examples of card suits are: spades and diamonds. Each rule has the name of the rank or suit followed by the value. This value determines if a card "beats", or is worth more than, another card. Joker then creates a set of cards as a cross product of the rank and suit rules. An example:

```
pack theDeck;
hierarchy : { A(14), K(13), Q(12), J(11) }
       by { spades (2) , hearts (1) } into theDeck;
```

Here, we made a set of aces, kings, queens, and jacks, of the two suits spades and hearts. The code segment "`into theDeck`" stores the set of cards within the declared `pack theDeck`.

So now we have our deck of cards. We most likely would like to begin manipulate cards from this deck. The pack type supports several list- and stack-style operations, such as pushing, popping, enqueueing, and a lil' something we like to call, back-popping.

Removing a card from the top of the deck (pop) is simple:

```
pack myHand = (theDeck >> 1);
```

This removes the top card from the deck and returns it as a pack. All the pack operations return references to packs. See the reference manual for additional information on the other operators.

Because we are playing high card, we probably just want the card. Cards can be accessed individually using the normal bracket-index syntax:

```
card myCard = myHand[0];
```

If we were code-scrooges, we would probably want to condense this to:

```
card myCard = (theDeck >> 1 )[0];
```

To finish off this simple one round version of High Card, let's compare two cards and print a winner.

Comparison of cards is programmed exactly like integers. Say we already have two cards, `card1` and `card2`:

```
if ( card1.value >  card2.value )
    print "Player 1 is the winner";

else if ( card1.value <  card2.value )
    print "Player 2 is the winner";

else
    print "Fooey, a tie.";
```

A little explanation is needed. Certain data types have properties known as attributes that can be accessed with the syntax "`variable.attributeName`". Cards have an attribute known as value, which you can use to get the "more dominant" card as based on the hierarchy declaration. (See the reference manual for additional attributes.)
So once we have figured out who has the "bigger" card, we declare the winner with a joyous print statement. Hurrah.

Here is the complete code sample:

[HighCard.jkr]

```
game HighCard {

    init {
       pack theDeck;
       // this is a shortened deck, for clarity's sake.
       hierarchy : { A(14), K(13), Q(12), J(11) }
              by { spades (2) , hearts (1) } into theDeck;
    }

    game {
       card card1 = (theDeck >> 1 )[0];
       card card2 = (theDeck >> 1 )[0];

       if ( card1.value >  card2.value )
           print "Player 1 is the winner";

       else if ( card1.value <  card2.value )
           print "Player 2 is the winner";

       else
           print "Fooey, a tie.";

    }
}
```

3 Language Reference

3.1 Lexical Conventions

3.1.1 Tokens

There are four classes of tokens: identifiers, keywords, operators and other separators. As in the other free-form languages, blanks, horizontal and vertical tabs, newlines (and "form feeds"), and comments as described below (collectively as "white space") are ignored except to note they separate tokens. Whitespace is required to separate otherwise adjacent identifiers, and keywords.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

3.1.2 Comments

The two types of comments: the start-stop comment and the end-of-line comment. A start-stop comment begins with a /* and includes all text until a closing */. The end-of-line comment starts with a // and continues until the end of the line. Comments do not nest and do not incur within string or character literals.

3.1.3 Identifiers

An identifier is a sequence of letters and digits. The first character of an identifier must be a letter; an underscore counts as a letter. Identifiers are case sensitive. Identifiers may have any length.

3.1.4 Keywords

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

int	***pack***	***else***
boolean	***string***	***for***
card	***if***	***foreach***
while	***hierarchy***	***value***
break	***by***	***size***
continue	***into***	***lt***
print	***default***	***gt***
init	***true***	***le***
main	***false***	***ge***
game	***rank***	
option	***suit***	

3.2 Meaning of Identifiers

Identifiers refer to actions and variables and are described by their *type*.

3.2.1 Basic Types

Types in Joker can be divided into two categories: *value* types and *reference* types. Value types consist of *string, boolean* and *integer* types. References types consist of *card* or *pack* types. Value types are passed by value. And reference types are, unsurprisingly, passed by reference.

Variables of the *integer* type are unsigned integers.

Variables of the *boolean* type can have the value of *true* or *false*.

Variables of the *string* type may be assigned a string constant. However, the string is immutable once assigned. Strings can be concatenated.

Variables of the *card* type represent playing cards.

Variables of the *pack* type represent groups of playing cards.

3.2.1.1 Card type

The *card* type is an object that represents a playing card. These objects have attributes, which consist of: a **rank**, a **suit**, and a **value**.

The **rank** is the rank value of a card. On a playing card, this is the corner numeral or letter. This attribute is a string-type and is access-only.

```
foo.rank == "A"
```

The **suit** is the suit of a card. This attribute is a string-type and is access-only.

```
foo.suit == "spades";
```

The **value** is the product of a card's rank-value and suit-value. The attribute is an integer-type and is access-only.

```
int tmp = foo.value;
```

3.2.1.2 Pack type

The *pack* type is an object that represents a group of playing cards. A number of special operators can be performed on pack. Like the *card* type, *pack* objects have attributes, which currently consists of: **size**.

The **size** of a pack is the number of elements in the pack. This attribute is access-only. The "bottom" or last element of the pack can be found at **size**-1.

3.2.2 Arrays

array-declaration
: *type-specifier* **[** *additive-expression* **]** *identifier*
| *type-specifier* **[ ]** *identifier init-array-identifier*

init-array-identifier
: **=** **{** *conditional-expression (, conditional-expression)** **}**

Users have the ability to make one-dimensional arrays of *int* and *boolean* values. For example, declaring an array of five integers is:

int[5] foo;

Arrays' indexes are zero-based and individual values can be accessed through the bracket notation.

3.2.3 Literals

Joker has three types of literals: integer, boolean and string.

An integer literal is a series of numerals, 0 to 9.

A boolean literal is the keyword **`true`** or the keyword **`false`**.

A string literal is a series of characters enclosed within double quotes. Escaped double quotes are supported.

3.3 Expressions

The precedence of expressions operators is the same as the order of the major subsections of this section, highest precedence first. Within each subsection, the operators have the same precedence. Left- or right- associativity is specified in each subsection for the operators discussed therein.

Like in Java™, the order of evaluation of expressions is left-to-right.

3.3.1 Primary Expressions

Primary expressions are identifiers, string, or expressions in parentheses.

primary-expression
: *identifier*
| *constant*
| `(` *expression* `)`

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. Examples of an identifier include: variable names.

A parenthesized expression is a primary expression, whose type and value are identical to those of the unadorned expression.

3.3.2 Postfix expressions

postfix-expression
: *primary-expression*
| *postfix-expression* `++`
| *postfix-expression* `--`
| *postfix-expression* `@`
| *postfix-expression* `[` *additive-expression* `]`

The operand is incremented (++) or decremented (--) and the new value stored within the operand. The value of the expression is the new value of the operand.

The shuffle operator is a unary operator that randomly reorders the elements of a pack. This operator can only be applied to a pack-type. The value of this expression is a reference to the reordered pack.

For example, the expression

```
deck @
```

reorders deck randomly and returns that deck itself.

The access operators (`[ ]`) can be used on an array or a pack-type. When used on an array, these operators return the individual value found that the index. When used on an array, these operators return a reference of type *card* to the card found at this index.

3.3.3 Arithmetic Expressions

3.3.3.1 Multiplicative Operators

multiplicative-expression
: attribute-expression
| *multiplicative-expression* ***** *postfix-expression*
| *multiplicative-expression* **%** *postfix-expression*

The binary operators "*" and "%" indicate multiplication and modulus, respectively. They are grouped left-to-right. They are only applicable to *int*.

3.3.3.2 Additive Operators

additive-expression: multiplicative-expression

| *additive-expression* + *multiplicative-expression*

| *additive-expression* **-** *multiplicative-expression*

When applied to integers, the additive operators "+", "-" indicate addition and subtraction, respectively and are grouped from left to right.

When applied to strings, the + operator performs concatenation and returns a reference to a new string object whose value is the second string concatenated onto the first string.

3.3.4 Pack Operators

Pack expressions are grouped from right-to-left. The pack type supports a number of stack operations, including pop, push, enqueue, backpop (which is popping from the rear).

pack-expression : pack-expression **>>** *additive-expression*
| *pack-expression* **<<** *pack-expression*
| *pack-expression* **+=** *pack-expression*
| *pack-expression* **-=** *additive-expression*

3.3.4.1 Pop operator (or "deal")

pack-expression >> integer-expression

The pop operator removes a variable number of elements from the "top" of a pack. The expression returns a pack containing the elements removed from the pack. These elements are in the same order as they were in the pack. That is, the top element on the source pack is the top element on the returned pack. The pack operand no longer contains these elements.

For example,

```
sourcePack >> 1
```

returns a pack containing a card from the top of sourcePack.

3.3.4.2 Push operator

pack-expression << *pack-expression*

The push operator inserts elements at the “top” of a pack. The elements are removed from “source” pack, maintaining order, moving as a whole, and place on top of the “destination” pack. The “source” pack is now emptied of elements.

For example, the expression

```
destinationPack <<  sourcePack
```

has a value of the reference to the newly modified destinationPack.

3.3.4.3 Enqueue operator

pack-expression += *pack-expression*

The enqueue operator inserts elements from the right operand onto the “bottom” or “back” of the left pack operand. The right pack operand is emptied of elements; all elements are transferred to the left pack operand.

For example, the expression

```
hand += (deck >> 1);
```

enqueues 1 element from the deck onto the bottom of hand.

The expression has the value of a reference to the left pack operand. Note: this operator is not to be confused with the integer assignment operator (+=).

3.3.4.4 Back-pop operator

pack-expression -= *integer-expression*

The back-pop operator removes a variable number (determined by *integer-expression)* elements from the “bottom” of the pack operand. The value of this expression is a reference to a new pack containing the removed elements.

For example, the expression

```
hand -= 2;
```

removes and returns 2 elements from the bottom of hand. Note: this operator is not to be confused with the integer assignment operator (+=).

3.3.5 Relational Expressions

relational-expression
: *pack-expression*
| *relational-expression* **<** *pack-expression*
| *relational-expression* **>** *pack-expression*
| *relational-expression* **<=** *pack-expression*
| *relational-expression* **>=** *pack-expression*
| *relational-expression* **lt** *pack-expression*
| *relational-expression* **gt** *pack-expression*
| *relational-expression* **le** *pack-expression*
| *relational-expression* **ge** *pack-expression*
| **true**
| **false**

The relational operators are binary which require two operands. They include >=, <=, > and < which corresponds to greater than or equal to, less than or equal to, greater than and less than, respectively.

The operands must both be of the integer type or both have the *card* type.

In the case of the integer comparison, the expression returns a boolean true if the specified relation is evaluated to true. Otherwise, they return a boolean false. For example,

```
int foo = 5;
if (foo > 3)
```

3.3.6 Equality Expressions

These equality expressions are binary and require two operands. They include equal to ("==") and not equal to ("!=").

equality-expression: relational-expression
| *equality-expression* == *relational-expression*
| *equality-expression* != *relational-expression*

Equality expressions return true if the specified equality is true. Otherwise, they return false.

3.3.7 Logical Expressions

These logical expressions are binary and require two operands. They include "&&" and "||" which correspond to "and" and "or" respectively. For example,

conditional-expression: logical-OR-expression

logical-OR-expression: logical-AND-expression
| *logical-OR-expression* || *logical-AND-expression*

logical-AND-expression: equality-expression
| *logical-AND-expression* **&&** *equality-expression*

equality-expression: relational-expression
| *equality-expression* == *relational-expression*
| *equality-expression* **!=** *relational-expression*

Logical expressions return true if the logical statement is true. Otherwise, they return false.

3.3.8 Assignment Operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value returned is the value stored in the left operand after the assignment has taken place.

assignment-expression: conditional-expression
| *unary-expression* **=** *assignment-expression*

An assignment is in the form:

lvalue = expression;

in which a semi-colon indicates the termination of this assignment operation. The value of lvalue will be replaced by that of the expression on the right provided that both the lvaue and the expression are of the same type.

3.4 Statements

3.4.1 Statements surrounded by “{“ and “}”

A group of zero or more statements can be surrounded by “{“ and “}”, in which case, all the statements are treated as a single statement.

compound-statement:
{ *(statement-list) ?* **}**

statement-list: *statement*
| *statement-list statement*

statement: *declaration-statement*
| *expression-statement*
| *iteration-statement*
| *selection-statement*
| *compound-statement*
| *jump-statement*
| *print-statement*

3.4.2 Conditional Statements

Begins with the `if` keyword and followed by an optional `else`. The `if` is followed by an *conditional-expression* contained within parenthesis. This expression is evaluated returning a boolean value `true` or `false`. If the expression returns true, then the statement following the `if` is executed. Otherwise, the statement following the `if` is skipped. If an optional `else` follows the `if`, then the statement following the `else` is executed.

selection-statement: **if (** *conditional-expression* **)** *statement*
| **if (** *conditional-expression* **)** *statement* **else** *statement*

3.4.3 Iterative Statements

iterative-statement
: **`while (`** *conditional-expression* **`)`** *statement*
| **`for (`** *(declaration | expression-list)? ;*
(conditional-expression)? ;
(expression-list)? **`)`** *statement*
| **`foreach`** *identifier* **`as`** *identifier statement*

Iterative statements are loops. There are two kinds of loops in Joker: **for** (and its cousin **foreach**) and **while**.

3.4.3.1 for statement

This statement has the form:

`for (` *(declaration | expression-list)? ;*
(conditional-expression)? ;
(expression-list)? **`)`** *statement*

The first *expression* specifies variable initialization for the loop. The second *expression* specifies the testing condition, which is made before each iteration such that the loop is terminated when *expression* evaluates to false. The third *expression* specifies the increment performed after each iteration.

3.4.3.2 foreach statement

foreach *identifier* ***as*** *identifier statement*

The *foreach-statement* loops over the array given by *array-expression* or *pack-expression*. On each loop, a reference of the current element is assigned to *identifier*. That is worth re-iterating: a modification to the value contained in *identifier* will modify the values in *array-expression* or *pack-expression*.

3.4.3.3 while statement

while **(** *conditional-expression* **)** *statement*

The statement is executed repeatedly as long as expression is evaluated as true. The test takes place before each execution of the statement.

3.4.4 Break statement

This statement will terminate the execution of the inner-most iterative statement; it consists of keyword **break**, followed by a ";". It has the form:

```
break;
```

3.4.5 Continue statement

This statement will end the current iteration of the inner-most iterative statement and proceed to its next iteration, if any. It consists of keyword **continue**, followed by a ";". It has the form:

```
continue;
```

3.4.6 Option statement

option-statement
: **option** **(** *pack-expression* **)** **{** *option-list* **}**

option-list
: *option-list -item (option-list -item)**

option-list-item
: *identifier* **(** *conditional-expression* **)** *statement*

| **default** *statement*

This statement specifies the actions simultaneously available to users. It consists of a list of conditional expressions and their appropriate actions. The *option* statement causes control to be transferred to one of the several actions, depending on the choice made from input.

The *identifier* is the name of action. The value of the *conditional-expression* determines if this action is available (true) or unavailable (false). The *statements* are executed if the corresponding action had been chosen.

If a default case is specified, this is executed when none of the actions are available; that is, their conditional expressions evaluated to false. If no default case is specified and no actions are available, execution control will exit the option statement harmlessly.

3.4.7 Print statement

A built in print function is available to print a message to standard output.

print-statement: **print** *expression*

3.4.8 Game statement

A **Joker** program is in the following form:

game *identifier* **{**
init *compound-statement*
main *compound-statement*
}

identifier is the name of the game.

State variables declared and within the init block exist for the entire lifespan of the game, and have a global scope, accessible everywhere. Hierarchy of cards should be defined within the init block.

The main section is where programmers specify the logic and rules of their game. In other words, this is where the modification of the declared state variables occurs. When all instructions in the main section are executed and when the end of the main section is reached, the game is over.

3.5 Declarations

A declaration consists of a type specifier, followed by an identifier (or a list of identifiers), each may or may not be followed by an assignment to an initial expression value.

declaration
: *type-specifier identifier-list*

type-specifier
: **`int boolean pack card`**

identifier-list
: *identifier-init*
| *identifier-list* **`,`** *identifier-init*

identifier-init
: *identifier*
| *identifier* (**`=`** *conditional-expression*)?

3.5.1 Array Declarations for *int* and *boolean*

array-declaration
: *type-specifier* **`[`** *additive-expression* **`]`** *identifier*
| *type-specifier* **`[ ]`** *identifier init-array-identifier*

init-array-identifier
: **`=`** **`{`** *conditional-expression* (**`,`** *conditional-expression*)* **`}`**

3.5.2 Hierarchy Declaration

Hierarchy of the domain of cards is defined by in the init block of every program. The compiler will automatically generate a pack of cards with named, valued cards. These cards are then stored in a specified pack variable.

A card will be created by taking an element-rule from rank and suit. Thus, the pack will be a Cartesian product of all ranks and suits. The hierarchy of the cards is defined by the *value* in the *element-rule*. Element-rules with an equivalent value are appended to the back of a name(value) with a period. Cards with the same rank and suit are equal.

For example,

```
pack myPack;

hierarchy: {A(12), K(11).Q.J, 10(10)} by
{spades(4), hearts(3), clubs(2), diamonds(1)} into myPack;
```

A domain of 25 cards covering all combinations of element-rules will be generated of this hierarchy definition. For comparison between two individual cards, see Section

3.4.2.

hierarchy-declaration
: **hierarchy** : *row* **by** *row* **into** *identifier*

row
: **{** *element-rule* (**,** *element-rule)** **}**

element-rule
: *identifier* **(** *integer-constant* **)** (**.** *identifier)**

3.6 Grammar

game
: **game** *identifier* **{**
init *compound-statement*
main *compound-statement*
}

compound-statement
: **{** *(statement)** **}**

statement
: *declaration-statement*
| *expression-statement*
| *iteration-statement*
| *selection-statement*
| *compound-statement*
| *jump-statement*
| *print-statement*
| *option-statement*

declaration-statement
: *declaration ;*
| *array-declaration ;*
| *hierarchy-declaration ;*

declaration
: *type-specifier identifier-list*

type-specifier
: **int boolean pack card**

identifier-list
: *identifier-init*
| *identifier-list* **,** *identifier-init*

identifier-init
: *identifier*
| *identifier* (**=** *conditional-expression)?*

array-declaration

: type-specifier **[** *additive-expression* **]** *identifier*
| *type-specifier* **[ ]** *identifier init-array-identifier*

init-array-identifier
: **=** **{** *conditional-expression (, conditional-expression)** **}**

hierarchy-declaration
: **hierarchy** **:** *row* **by** *row* **into** *identifier*

row
: **{** *element-rule (, element-rule)** **}**

element-rule
: identifier **(** *integer-constant* **)** *(. identifier)**

expression-statement
: (expression) ? **;**

selection-statement
: **if** **(** *conditional-expression* **)** *statement*
| **if** **(** *conditional-expression* **)** *statement* **else** *statement*

iterative-statement
: **while** **(** *conditional-expression* **)** *statement*
| **for** **(** *(declaration | expression-list)? ;*
(conditional-expression)? ;
(expression-list)? **)** *statement*
| **foreach** *identifier* **as** *identifier statement*

jump-statement
: **break** **;**
| **continue** **;**

option-statement
: **option** **(** *pack-expression* **)** **{** *option-list* **}**

option-list
*: option-list -item (option-list -item)**

option-list-item
: identifier **(** **c***onditional-expression* **)** *statement*
| **default** *statement*

print-statement
: **print** *expression* **;**

expression
: assignment-expression

expression-list
*: expression (, expression) **

assignment-expression
: conditional-expression
| *conditional-expression* **=** *assignment-expression*

conditional-expression
: *logical-OR-expression*

logical-OR-expression
: *logical-AND-expression*
| *logical-OR-expression* **||** *logical-AND-expression*

logical-AND-expression
: *equality-expression*
| *logical-AND-expression* **&&** *equality-expression*

equality-expression
: *relational-expression*
| *equality-expression* **==** *relational-expression*
| *equality-expression* **!=** *relational-expression*

relational-expression
: *pack-expression*
| *relational-expression* **<** *pack-expression*
| *relational-expression* **>** *pack-expression*
| *relational-expression* **<=** *pack-expression*
| *relational-expression* **>=** *pack-expression*
| *relational-expression* **lt** *pack-expression*
| *relational-expression* **gt** *pack-expression*
| *relational-expression* **le** *pack-expression*
| *relational-expression* **ge** *pack-expression*
| **true**
| **false**

pack_expression
: *additive_expression*
| *pack-expression* **>>** *additive_expression*
| *pack-expression* **<<** *additive_expression*
| *pack-expression* **-=** *additive_expression*
| *pack-expression* **+=** *additive_expression*

additive-expression
: *multiplicative-expression*
| *additive-expression* **+** *multiplicative-expression*
| *additive-expression* **-** *multiplicative-expression*

multiplicative-expression
: *attribute-expression*
| *multiplicative-expression* ***** *postfix-expression*
| *multiplicative-expression* **%** *postfix-expression*

attribute-expression
: *postfix-expression*
| *attribute-expression* **.rank**
| *attribute-expression* **.value**
| *attribute-expression* **.suit**
| *attribute-expression* **.size**

postfix-expression
: *primary-expression*

| *postfix-expression* **++**
| *postfix-expression* **--**
| *postfix-expression* **@**
| *postfix-expression* **[** *additive-expression* **]**

primary-expression
: *identifier*
| *constant*
| **(** *expression* **)**

constant
: *integer-constant*
| *string-constant*

4 Project Plan

4.1 Overview

We have successfully divided the project into different modules and every member on the team was able to involve in both the programming and documentation part of the project. The project took about a month to finish, and it's divided into four phases as specified below. Overall, we were able to implement all the features which we specified in our LRM. As a matter of fact, we were able to implement a couple of new nifty features in our language which we did not think of before. Therefore, we considered we exceeded our original goal.

4.2 Team Members Responsibilities

Howard Chu	Primary- Tree Walker, Secondary- Backend Classes, Documentation
Timothy SooHoo	Primary- Backend Classes, Secondary- Tree Walker, Documentation
Jonathan Tse	Lexer/Parser, Testing, Documentation
Jeffrey Eng	Lexer/Parser, Testing, Documentation

4.3 Planning Phase

At the planning phase, we met twice a week to first figure out what domain of card games will our language support. Then, instead of diving into creating the syntax, we try to code BlackJack with the syntax and operations that we think will be useful and efficient for

computer card games developers. After specifying the domain, we set our goal to be able to at least code a game of black jack and war. We also set the runtime system of our language to be a modest console output because we want to focus our energy on developing the actual card game languages.

4.4 Specification Phase

Given the fact that we have our goal, we tried to come up with a list of useful operations and built in types that will be helpful for card game developers. To do that, we researched on the common rules of turn based card games, which we created various operations in order for developers to implement those common rules such as packs of card, easy retrival of the rank and value of a card, easy access of a pack and setting the hierarchy of a pack. All these were included in our LRM, and the LRM served as our major source of the specification throughout the course of the development of the project.

4.5 Development Phase

We have adopted the extreme programming module in which all four of us were actively participated in the development process. Two of us were working on the lexer/parser and designing of test plan while the other two of us were working on tree walker/backend together. Even though we splitted our team into two groups, we communicated very frequently as tree walker were highly dependent on the parser and we developed these two modules parallely.

4.6 Testing Phase

We have developed an automatic test suite which allows us to do regression testing. This test suite takes in set of dummy programs (which just test a feature of Joker) and generate the Joker source files and then run these programs through the Joker interpreter to make sure the outputs of these programs are what they are supposed to be. Details to be described in the test plan section below.

4.7 Programming-style Guide

4.7.1 Antlr Code Style

If an Antlr rule is short and contains only one choice without any action, then it will be written in one line. The colon ":" always starts at the ninth column unless the string in front of ":" is too long. Both the ";" and "|" is placed on the next line regardless of how long the rule is.

Short actions are in the same line of its rule and at the right half of the screen. Long action lines start at the thirteenth

column of the next line. Code in actions follows the Java coding style.

Lexem (token) names are in upper cases and syntactic items (non-terminals) are in lower cases, which may contain the underscore "."

4.7.2 Java Code Style

As the programmers using multiple programming languages, we use a coding style that is a little different to the standard Java coding style.

4.7.2.1 Indentation and spacing

- Indentation of each level is 4 spaces.
- The left brace "{" occupies a full line and is at the same column as the first character of the previous text.
- The right brace "}" occupies a full line and at the same column of its corresponding "{".
- One space between arguments and their parentheses "(" ")", but no space between function name and "(".
- Add spaces between operators and operands for outer expressions.
- Use spaces, do not use tabs.
- The then-part and else-part of an \if" statement must not be at the same line of "if" or "else". Similar for "for" statement.

4.7.2.2 Naming

- Class names start with “Jkr” followed by English words whose first character is capital.
- Variables are in lower cases, and words are separated by underscore "_".
- Do not use long names for local variables. More concise, the better.
- Method names are English words whose first characters are capital except for the first word.

Follow Java javadoc standard for comments.

4.8 Project Timeline

11/16-11/22	Lexer/Parser is in the process of development. CVS is setup
11/23-11/29	Finish the development of Lexer/Parser and formulate a preliminary set of test cases. Start the development of backend classes and tree walker.
11/30-12/6	Update the LRM and finalize the features of the language and make sure the Lexer/Parser code is stable after first set of testing.
12/7-12/13	Tree walking and backend should be completed.
12/14-12/18	Wrap up the project

4.9 Software Project Environment

The backend classes and majority of the tree walker was written in Java. Lexer, Parser and part of the tree walker was written in ANTLR. The documents are created with Microsoft Word, Microsoft Visio and Adobe Acrobat.

4.9.1 Operating Systems

Since this project is developed in pure Java, it can be used anywhere that a Java VM is running as long as ANTLR is also installed. We have developed our project with Linux and Windows. Because we use CVS, and since the development is not OS dependent, we switch from different OS at our convenience without any hassle.

4.9.2 Java 1.4

Java is a "simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language". Its object-oriented nature helped us to develop our backend classes in a systematic and organized fashion.

4.9.3 ANTLR

The language parser is written in Antlr, "a language tool that

provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions."

4.9.4 CVS

CVS is a well-known concurrent version system for developers. We kept our CVS repository on a Linux machine, and use SSH to check in/out our programs with secured network connections.

4.9.5 Make/Ant

Make is a well-known program which takes in a makefile to build a project automatically. Ant is the next generation language for building makefile types of program to build a project and it is XML based. We have used both of these languages to build our project in an efficient manner.

4.10 Project Log

Here are the dates of significant project milestones. Most of them come from the CVS logs.

Nov 1	Lexer/Parser started
Nov 19	CVS repository initialized, Lexer/Parser in good shape
Nov 23	Lexer/Parser done
Nov 23	Backend/Treewalker started
Dec 2	Major changes in Backend/Treewalker
Dec 7	Backend/Treewalker done
Dec 9	All the code are done
Dec 10	Start the testing suite
Dec 14	Stablized code is in repository after running through the test suite
Dec 18	Demo, all documents and final report due

5 Architectural Design

5.4 Overview

The Joker programming language utilizes an Antlr generated lexer and parser which passes an abstract syntax tree to the Antlr generated tree walker. With the tree walker, static semantic analysis is performed on the fly with the aid of backend Java classes. The tree walker also depends on the backend Java classes to interpret and execute the Joker program. As the tree walker interprets and executes the program, the user can view the results via the program output.

5.5 Joker Source File

A .jkr file must first be created following the syntax rules and conventions of the Joker programming language. This file is fed into 'JokerMain.Java' which will perform lexical analysis, parse the source file, and perform static semantic analysis while interpreting and executing the code within the tree walker

5.6 Lexer and Parser Design

The lexer and parser are generated by Antlr with syntax and semantic programming rules created within 'joker.g', the grammar file used by Antlr to generate the lexer and parser. The lexer performs lexical analysis on the Joker source file while the parser parses the source file into an abstract syntax tree to pass to the tree walker. We turned off Antlr's default error handler within the lexer and parser to enable out own exception handling. Exceptions are passed up through the program stack until it reaches the JokerMain in which all exceptions are caught and reported to the user. This prevents the program from executing if there are any lexical or parsing errors returned.

5.7 Tree Walker Design

The Joker tree walker is generated by Antlr from 'joker_walker.g'. The tree walker performs static semantic analysis as it walks the abstract syntax tree generated by the parser. At the same time, the tree walker interprets and executes the program code. The tree walker utilizes backend Java classes to provide wrappers for its data types and keep track of multiple symbol tables. The tree walker interfaces to the Java backend components through the JkrInterpreter class. This gives the walker access to its symbol tables and helper functions during execution.

5.8 Backend Java Class Design

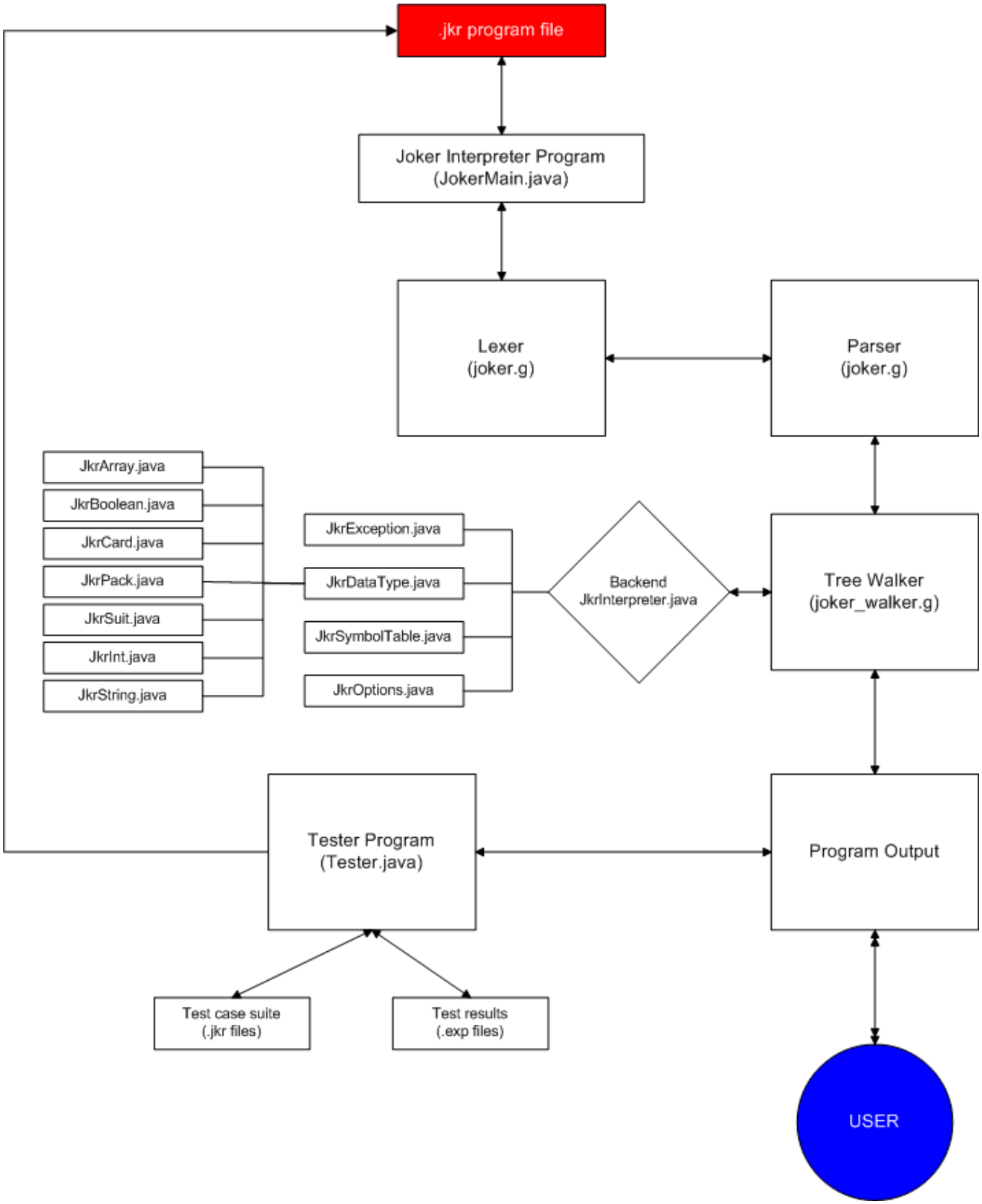
The Java backend classes wrap Joker data types as well as provide book keeping and helper functionality to the tree walker. Joker data type wrappers are all implemented from a parent class JkrDataType. This provides a unified interface for which all child data types can be derived. This helps in error handling as well as providing functions for child data types to both inherit or override if they choose. The backend classes are utilized by the Joker tree walker via the JkrInterpreter class which provides an interface to other backend classes. The JkrInterpreter is responsible for the symbol table book keeping as well as providing functions to help in loop execution and execution of the option block. This again provides a single interface to connect the tree walker to the backend classes.

5.9 Tester Program

This portion of the Joker language architecture was created to allow for execution of a test suite of sample Joker source code. These source files are executed in a batch processing style with program output being written to .exp files which are stored for immediate inspection or comparison to the correct program output. This provided a means for incremental testing as new features of the language were implemented.

5.10 Implementation Responsibility

All of us originally worked on language specifications and overall design of the language. This involved determining what features and implement the general architectural design decision. From there, Jeff and Jon primarily worked on implementing the lexer and parser forming the foundation of our language. With the tree well planned and carefully created, Tim and Howard implemented the tree walker. Tim also implemented the backend classes working closely with Howard to ensure that the tree walker and backend classes remained synchronized and streamlined. Jon also worked on the testing suite and the engine that would run these regression tests to ensure our implementations were following our specifications after performing incremental testing. [See Section 4.2]


.jkr program file
Joker Interpreter Program
(JokerMain.java)
Lexer
(joker.g)
Parser
(joker.g)
JkrArray.java
JkrBoolean.java
JkrCard.java
JkrPack.java
JkrSuit.java
JkrInt.java
JkrString.java
JkrException.java
JkrDataType.java
JkrSymbolTable.java
JkrOptions.java
Backend
JkrInterpreter.java
Tree Walker
(joker_walker.g)
Tester Program
(Tester.java)
Program Output
Test case suite
(.jkr files)
Test results
(.exp files)
USER

6 Test Plan

6.1 Overview

As a tool for the programmer, any incorrectness in the Joker interpreter is unacceptable. In order to achieve this absolute correctness, rigorous and methodical testing must be done. The Joker team has submitted the Joker interpreter to several rounds of testing as detailed below.

6.2 First Round: Unit Testing

The first round of testing began on the unit level. Team members responsible for each unit (e.g. parser, treewalker) incrementally developed their respective unit. Each feature was added, followed by a rigorous round testing of that feature. Each unit had a corresponding output from which to debug from. For example, the lexer and parser were able to print out a tree in Lisp notation. The backend was able to use the tried-and-true System.out.println( ) And so forth.

6.3 Second Round: Regression Testing

When we began to piece together the components of the Joker interpreter, the size of the application gave us pause. To test this complex program now required a more advanced solution. We decided to build an automated test suite.

Test cases were manually generated to test each piece of functionality in the language. Each test case was designed to be the smallest program required to test the intended functionality. In the end, we were able to create a pool of over 100 test cases with which to test. The tested functionality varied from declarations to pack operations. Team members Jeffrey and Jonathan manually generated these test cases.

Now comes the automation. A Java program was written to loop over the set of test cases and run the interpreter on each program. Depending on this program's usage, it would either write the output to a store of expected outputs or compare the output with the store of expected outputs. The format of each test case mandated that the first line of the file would contain a short description of the functionality tested. This allowed us to quickly isolate problems and understand bugs when the automated test suite abort prematurely. This automated testing program was developed by team member Jonathan Tse.

Once this automated test suite was complete, we now had a measure of interpreter behavior. We could continue to add new feature or modify the interpreter, and running the test suite would let us know if we broke something.

6.4 Third Round: Advanced Code Samples

At the end of our implementation, we began to develop sample Joker programs for demonstrating the capabilities of the Joker language. These programs include Black Jack and War (both samples can be found in the Appendix.) These programs allowed us to test our functionality working in a complex and nontrivial example. These programs also enabled us to gain insight into our language design and to see several benefits and shortcomings to the decisions we made.

7 Lessons Learned

7.1 Howard Chu

This project wasn't that enjoyable until at the ending phase where we can actually view the output of the language that we created. It's definitely the coolest project I have done in Columbia. The main lesson I learned from this project was first of all the importance of picking a good team. I am managed to pair up with very reliable teammates who looked over each other shoulders and filled each others hole. The completion of this project is definitely a team effort. Secondly, CVS is definitely a key. Besides team file access, the ability to be able to rollback to previous versions definitely save us a lot of times. Overall, I really enjoy this project.

7.2 Timothy SooHoo

Creating the Joker programming language with my fellow group members was truly a good learning experience individually and as a group. I was reminded of the importance of communication as the binding thread between all the group members and its ability to make a project go well or poorly. Our group had great communication whether over email, phone, or simply meeting in person. Weekly meetings were very good in keeping the group organized and on task. It also forced each member to report the progress that he had made in the previous week which helped to keep each group member focused on the tasks ahead. We also set incremental milestones and laid out a calendar of due dates for different parts of the project. Although a project schedule is hard to stick to, it again helps to organize the group so that nobody is caught off guard when we hit the final crunch at the end. I was definitely glad that I had learned to use team development tools such as Current Versioning System (CVS) as well as personal tools such as IntelliJ Idea, a java integrated development environment, as well as Apache's Ant build scripts which helped me to increase productivity while staying organized. Besides the individual work each group member did, we also split up the project tasks using a "divide and conquer" approach. Two people worked on the lexer and parser while the two others worked on the tree walker. This allowed each pair to try "Extreme Programming" (XP) which basically involves programming in front of the same terminal together. This definitely proved very productive and kept each group member aware of the development progress. We also divided up the final report parts so that it could be created quicker. Overall, working with my project team was a positive experience and I can only hope to transfer these lessons to future projects whether they be academic or work related.

7.3 Jonathan Tse

When I first heard about the project, I thought it was going to be a complex abstract mess and I had no idea where to start. It was difficult to visual what the next few months were going to be like. I learned that there's no other way to start other than to jump into the deep end. After the fact, it turned out that doing the ANTLR part was a lot simpler than it seemed.

From the implementation side of the project, a huge lesson was in a style of programming known as pair programming. I saw more pros than cons in this programming paradigm, and it worked well for us. I recommend it to future groups.

In all, a programming language is a lot more complicated than it seemed from the surface. It was interesting as I was taking Japanese at the same time and I saw similarities between how humans do our own version of lexical scanning.

7.4 Jeffrey Eng

I was team leader of the scrappy band of programmers known as the Joker team. It was a great challenge to design one's own programming language. I have some quick thoughts on the subject:

CVS is a clunky piece of versioning software. But it's free and it works. Use it. The night before the project demo, our team began cleaning up the code and removing Java debug System.out.println statements from everywhere. When we tried to compile again, a horrible, horrible untraceable bug came up that we could not track down after an hour (at 5am). We then decided to just roll back our code to the version we knew worked.

Jump into ANTLR early. The better you know how ANTLR works, the better you can design your whole project and the better you know what's actually hard to implement and what's only hard when you didn't know ANTLR.

Set up the tools for everyone. Within our team, we had varying levels of familiarity with the tools we used (CVS, Makefiles, ANTLR, etc.). Make sure everyone knows how to use your tools and make sure they have it installed (install it for them, if need be.)

Make a calendar and keep staring at it. At one point we made a calendar with all our planned milestones ("parser on this day, tree walker due on this day, etc."). Of course, we didn't make a single deadline, but every meeting we had, we brought out the calendar and stared at it and understood how much time was left.

Pair programming worked for us. At least one person was always hyped up enough to code and then one could tag out and be replaced. And you do not lose 4 hours of project time hunting down a vagrant semicolon. And most importantly, having two people meet to work always ensures that something gets worked on, at least a little bit. Also, Mario Kart Double Dash plays better with teams of two.

8 Appendix

8.1 Lexer and Parser – Joker.g

```
/**
 * Name:  Jeffrey Eng and Jonathan Tse
 * Date:  November 15, 2003
 *
 * File:  Joker.g
 * Note:  Lexer and Parser
 */

class JokerLexer extends Lexer;

options {
    defaultErrorHandler = false;
        k = 2;
        charVocabulary = '\3'..'\377';
        testLiterals = false;
}

PLUS    : '+';
MINUS   : '-';
TIMES   : '*';
MOD             : '%';

ASSIGN  : '=';
PLUSEQ  : '+' '=';
MINUSEQ : '-' '=';

DOT     : '.';
COMMA   : ',';

PLUSPLUS   : '+' '+';
MINUSMINUS : '-' '-';

LPAREN  : '(';
RPAREN  : ')';
LBRACE  : '{';
RBRACE  : '}';
LBRAKT  : '[';
RBRAKT  : ']';

SHIFTL  : '<' '<';
SHIFTR  : '>' '>';

ATSIGN  : '@';

LANGLE  : '<';
RANGLE  : '>';
EQUALS  : '=' '=';
NOTEQ   : '!' '=';
LESSEQ  : '<' '=';
GRTREQ  : '>' '=';

AND     : '&' '&';
OR      : '|' '|';

COLON   : ':';
SEMI    : ';' ;
```

```
protected LETTER : ('a'..'z' | 'A'..'Z') ;
protected DIGIT : '0'..'9' ;

ID options { testLiterals = true; }
   : LETTER (LETTER | DIGIT | '_')* ;

NUMBER : (DIGIT)+;

// double quotes are escaped with a \"
STRING : '"'! ( ('\\' '"')  => '\\'! '"'
                | ~('"'))* '"'! ;

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

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

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


class JokerParser extends Parser;

options {
    defaultErrorHandler = false;
    buildAST = true;
    k = 2;
}

tokens {
    STATEMENTS;
    DECL;
    DECL_HIER;
    DECL_ARR;
    DECL_EMPTY_ARR;
    VALUES;
    RANK_ROW;
    SUIT_ROW;
    ELEMENT_RULE;
    EXPR;
    EXPR_STMT;
    ACCESS;

    FOR_PARAMS;
    FOR_INIT;
    FOR_COND;
    FOR_ITER;

    ATTRIB;
}

game
    :  "game"^ ID LBRACE!
            "init"! compound_statement
            "main"! compound_statement
        RBRACE! EOF!;

compound_statement
    : LBRACE! (statement)* RBRACE!
        { #compound_statement = #([STATEMENTS, "stmts"], compound_statement); }
    ;
```

```
statement
        : declaration_statement
        | expression_statement
        | iteration_statement
        | selection_statement
        | compound_statement
        | jump_statement
        | print_statement
        | option_statement
        ;

declaration_statement
        : declaration SEMI!
        | array_declaration SEMI!
        | hierarchy_declaration SEMI!
        ;

declaration
        : type_specifier identifier_list
                { #declaration = #([DECL, "decl"], declaration); }
        ;

type_specifier
        : "int" | "boolean" | "pack" | "card" | "string" ;

identifier_list
        : identifier_init (COMMA! identifier_init)*
        ;

identifier_init
        : ID^ (ASSIGN! conditional_expression )?
        ;

array_declaration
        : type_specifier LBRAKT!
         ( ( additive_expression RBRAKT! ID
             { #array_declaration = #([DECL_ARR, "decl_arr"], array_declaration); } )
        |
         ( RBRAKT! ID init_array_identifier
              { #array_declaration = #([DECL_EMPTY_ARR, "decl_empty_arr"], array_declaration); } )
         )
        ;

init_array_identifier
        : ASSIGN! LBRACE! conditional_expression
          (COMMA! conditional_expression)*   RBRACE!
          { #init_array_identifier = #([VALUES, "values"], init_array_identifier); }
        ;

hierarchy_declaration
        : "hierarchy"! COLON! rank_row "by"! suit_row "into"! ID
          { #hierarchy_declaration =
             #([DECL_HIER, "decl_hier"], hierarchy_declaration); }
        ;


rank_row
        : LBRACE! element_rule  (COMMA! element_rule)*  RBRACE!
        { #rank_row = #([RANK_ROW, "rank_row"], rank_row); }
        ;

suit_row
        : LBRACE! element_rule  (COMMA! element_rule)*  RBRACE!
        { #suit_row = #([SUIT_ROW, "suit_row"], suit_row); }
        ;

element_rule
        :  element_rule_head (DOT! ID )*
        { #element_rule = #([ELEMENT_RULE, "element_rule"], element_rule); }
        ;

element_rule_head
```

```
        :  ID^ LPAREN! NUMBER RPAREN! ;

expression_statement
        : ( expression )? SEMI!
        { #expression_statement
            = #([EXPR_STMT, "expr_stmt"], expression_statement); }
        ;

selection_statement
        : "if"^ LPAREN! conditional_expression RPAREN! statement
        (options {greedy=true;} : "else"! statement)?
        ;

iteration_statement
        : "while"^ LPAREN!  conditional_expression RPAREN! statement
        | "for"^ for_params statement
        | "foreach"^ ID "as"! ID statement
        ;

for_params
        : LPAREN! for_init SEMI! for_cond SEMI! for_iter RPAREN!
         { #for_params = #([FOR_PARAMS, "for_params"], for_params); }
        ;

for_init
        : (declaration | expression_list )?
        { #for_init = #([FOR_INIT, "for_init"], for_init); }
        ;

for_cond
        : (conditional_expression)?
        { #for_cond = #([FOR_COND, "for_cond"], for_cond); }
        ;

for_iter
        : (expression_list)?
        { #for_iter = #([FOR_ITER, "for_iter"], for_iter); }
        ;

jump_statement
        : "break"^ SEMI!
        | "continue"^ SEMI!
        ;

option_statement
        : "option"^ LPAREN! pack_expression RPAREN!
        LBRACE! option_list RBRACE!
        ;

option_list
        : option_list_item (option_list_item)*;

option_list_item
        : ID^  LPAREN! conditional_expression RPAREN! statement
        | "default"^  statement
        ;

print_statement
        : "print"^ expression SEMI!;

expression
        : assignment_expression
        { #expression = #([EXPR, "expr"], expression); }
        ;

expression_list
        : expression (COMMA! expression) *
        ;

assignment_expression
        : conditional_expression (ASSIGN^  conditional_expression )?
        ;
```

```
conditional_expression
        : logical_OR_expression
        ;

logical_OR_expression
        : logical_AND_expression  (OR^ logical_AND_expression)*
        ;

logical_AND_expression
        : equality_expression (AND^ equality_expression)*
        ;

equality_expression
        : relational_expression ( (EQUALS^ | NOTEQ^) relational_expression ) *     ;

relational_expression
        : pack_expression
         (( LANGLE^ | RANGLE^ | LESSEQ^ | GRTREQ^
            | "lt"^ |  "gt"^ | "le"^ | "ge"^  ) pack_expression  ) *
        | "true"
        | "false"
        ;

pack_expression
        : additive_expression ((SHIFTR^ | MINUSEQ^ | SHIFTL^ | PLUSEQ^)
         additive_expression)*
        ;

additive_expression
        : multiplicative_expression ((PLUS^ | MINUS^) multiplicative_expression)*
        ;

multiplicative_expression
        : attrib_expression ((TIMES^ | MOD^) attrib_expression)*
        ;

attrib_expression
        : postfix_expression
         (DOT! ("rank" | "suit" | "value" | "private" | "size" )
             { #attrib_expression = #([ATTRIB, "attrib"], attrib_expression); }
         )?
            ;

postfix_expression
        : primary_expression
         (PLUSPLUS^ | MINUSMINUS^ | ATSIGN^ | (LBRAKT! additive_expression RBRAKT!
          { #postfix_expression = #([ACCESS, "access"], postfix_expression); }
           )
         )?
        ;

primary_expression
        : ID
        | constant
        | LPAREN! expression RPAREN!
        ;

constant
        : NUMBER
        | STRING
        ;
```

8.2 Tree Walker – joker_walker.g

```
/**
 * Tree walker for the Joker Programming Language
 *
 * @author Howard Chu, Timothy SooHoo
 */
class JokerWalker extends TreeParser;
options
{
    importVocab = JokerLexer;
}

{
    static JkrDataType null_data = new JkrDataType("<null>");
    JkrInterpreter interpreter = new JkrInterpreter(this);
    JkrPack pack = new JkrPack();
    JkrOptions option = new JkrOptions(this);
    java.util.Vector cards = new java.util.Vector();
    java.util.Vector suits = new java.util.Vector();
}

expr returns [ JkrDataType r ]
    {
        if(interpreter.cont || interpreter.brk)
            return null;

        JkrDataType a = null;
        JkrDataType b = null;
        JkrDataType c = null;
        r = null_data;
    }
    : #("print" a=expr  { System.out.println(a); } )
    | #(ATTRIB
            attrib_tree:.
            {
                //System.out.println("in attrib");
                // primary expression of attribute tree, the type
                a = expr(#attrib_tree);

                AST actual_attrib = attrib_tree.getNextSibling();
                if(actual_attrib == null) {
                    System.err.println("Invalid attribute statement");
                    System.exit(-1);
                }
                if(!actual_attrib.getText().equals("rank") &&
                   !actual_attrib.getText().equals("suit") &&
                   !actual_attrib.getText().equals("private") &&
                   !actual_attrib.getText().equals("value") &&
                   !actual_attrib.getText().equals("size") ) {
                    System.err.println("Invalid attribute statement. Wrong attribute type");
                    System.exit(-1);
                }

                if(a instanceof JkrArray) {
                    if(!actual_attrib.getText().equals("size")) {
                        System.err.println("Invalid attribute statement: can only use size attribute
                        on array");
                        System.exit(-1);
                    }
                    r = new JkrInt(((JkrArray)a).length());
                } else if(a instanceof JkrPack) {
                    if(!actual_attrib.getText().equals("size")) {
                        System.err.println("Invalid attribute statement: can only use size attribute
                        on pack");
                        System.exit(-1);
                    }
                    r = new JkrInt(((JkrPack)a).getSize());
```

```
            } else if(a instanceof JkrCard) {
                if(!actual_attrib.getText().equals("rank") &&
                   !actual_attrib.getText().equals("suit") &&
                   !actual_attrib.getText().equals("value") &&
                   !actual_attrib.getText().equals("private")) {
                        System.err.println("Invalid attribute statement: only use rank, suit,
                         private on card");
                    System.exit(-1);
                }
                if(actual_attrib.getText().equals("rank")) {
                    r = ((JkrCard)a).getRank();
                } else if(actual_attrib.getText().equals("value")) {
                    r = new JkrInt(((JkrCard)a).getValue());
                }
                else if(actual_attrib.getText().equals("suit")) {
                    JkrSuit tmp_suit = ((JkrCard)a).getSuit();
                    r = tmp_suit;
                } else if(actual_attrib.getText().equals("private")) {
                    r = new JkrBoolean(((JkrCard)a).getViewability());
                }
            } else {
                System.err.println("Invalid attribute statement");
                System.exit(-1);
            }
        }
   )
| #("option"
        option_tree:.
        {
            a = expr(#option_tree);
            if(!(a instanceof JkrPack)) {
                System.err.println("Invalid option statemement");
                System.exit(-1);
            }

            boolean hasDefaultCase = false;
            option.init();
            AST option_case = option_tree.getNextSibling();
            while(option_case != null) {
                String op_case_id = #option_case.getText();

                if(op_case_id.equals("default")) {
                    AST op_default_stmt = option_case.getFirstChild();
                    option.addOptionStatement(JkrOptions.DEFAULT, op_default_stmt);
                    hasDefaultCase = true;
                } else {
                    AST op_case_cond = option_case.getFirstChild();
                    AST op_case_stmt = op_case_cond.getNextSibling();

                    option.addOptionCondition(op_case_id, op_case_cond);
                    option.addOptionStatement(op_case_id, op_case_stmt);
                }

                option_case = option_case.getNextSibling();
            }

            if(!hasDefaultCase)
            {
                System.err.println("Error:  'option' block requires default case");
                System.exit(-1);
            }
            else{
                option.execOptions();
                option.end();
            }
        }
   )
| #("foreach"
        foreach_tree:.
        {
            //System.out.println("Begin foreach tree");
```

```
            AST foreach_expr = #foreach_tree;
            AST foreach_id = foreach_expr.getNextSibling();
            AST foreach_stmt = foreach_id.getNextSibling();

            JkrDataType container = expr(foreach_expr);

            if(container instanceof JkrArray) {
                //System.out.println("Within container instanceof JkrArray");
                JkrArray tmp = (JkrArray)container;

                if(tmp == null)
                    System.out.println("ARRAY IS NULL");

                interpreter.newScope();
                JkrDataType itr_id = expr(foreach_id);
                interpreter.execForeach(itr_id, tmp, foreach_stmt);
                interpreter.closeScope();
            }
            else if(container instanceof JkrPack) {
                JkrArray tmp = new JkrArray((JkrDataType[])((JkrPack)container).getArray());
                tmp.name = container.name;

                interpreter.newScope();
                JkrDataType itr_id = expr(foreach_id);
                interpreter.execForeach(itr_id, tmp, foreach_stmt);
                interpreter.closeScope();
            }
            else {
                System.err.println("invalid foreach statement");
                System.exit(-1);
            }

            //System.out.println("End foreach tree");
        }
    )
| #(FOR_PARAMS for_params_tree:.
        {
            //System.out.println("Begin of for params statement");

            AST for_init = #for_params_tree;
            //System.out.println("for_init: " + for_init.toStringList());
            AST for_cond_expr = for_init.getNextSibling();
            //System.out.println("for_cond: " + for_cond_expr.toStringList());

            AST for_itr = null;
            for_itr = for_cond_expr.getNextSibling();

            if(for_cond_expr.getFirstChild() != null) {
                interpreter.setForCond(for_cond_expr);
            }
            else
            {
                interpreter.setForCond(null);
            }

            //start a new scope here
            interpreter.newScope();

            if(#for_init.getFirstChild() != null) {
                expr(#for_init);
            }

            if(for_itr.getFirstChild() != null) {
                interpreter.setForItr(for_itr);
            }
            else {
                interpreter.setForItr(null);
            }

            //System.out.println("End of for params statement");
        }
    )
```

```
| #(FOR_INIT r=expr)
| #(FOR_COND r=expr)
| #(FOR_ITER r=expr)
| #("for" for_tree:.
    {
        //System.out.println("in for");

        AST for_params = #for_tree;
        AST for_stmts = for_params.getNextSibling();

        //System.out.println("for_params: " + for_params.toStringList());
        //System.out.println("for_stmts: " + for_stmts.toStringList());

        expr(for_params);

        while(!interpreter.brk && interpreter.forCondTRUE())
        {
            interpreter.newScope();
            expr(for_stmts);
            interpreter.execForItr();
            interpreter.closeScope();
        }

        //close for loop scope
        interpreter.closeScope();

        //System.out.println("end for stmt");
    }
  )
| #("while"
    while_tree:.
    {
        //System.out.println("in while");
        AST while_cond_expr = #while_tree;
        AST while_stmts = while_cond_expr.getNextSibling();

        a = expr(while_cond_expr);


        if(a instanceof JkrBoolean)
        {
            while(!interpreter.brk && ((JkrBoolean)a).bool)
            {
                //start new scope
                interpreter.newScope();

                r = expr(while_stmts);
                a = expr(while_cond_expr);

                //up one scope
                interpreter.closeScope();
            }
            interpreter.setContinue(false);
            interpreter.setBreak(false);

        } else {
            System.err.println("invalid while condition");
            System.exit(-1);
        }

        //System.out.println("end while stmt");
    }
   )
| #("if"
    if_tree:.
    {
        //System.out.println("in if statement");
        // IN OUR LANG, IF REQUIRES ST
        AST if_cond_expr = #if_tree;
        AST if_then_stmt = if_tree.getNextSibling();
        AST if_else_stmt = if_then_stmt.getNextSibling();
```

```
        a = expr(if_cond_expr);
        //System.out.println("if condition: " + if_tree.toStringList());

       if(a instanceof JkrBoolean)
        {
            if(((JkrBoolean)a).bool == true)
            {
                //start new scope
                interpreter.newScope();

                r = expr(if_then_stmt);

                //up one scope
                interpreter.closeScope();
            }
            else if(if_else_stmt != null)
            {
                //start new scope
                interpreter.newScope();

                r = expr(if_else_stmt);

                //up one scope
                interpreter.closeScope();
            }
            else
            {
            }

        } else {
            System.err.println("invalid if condition");
            System.exit(-1);
        }
      }
   )
| #(PLUSPLUS a=expr
        {
            if(a instanceof JkrInt) {
               b  = new JkrInt(JkrInt.intValue(a) + 1);
               interpreter.assign(a, b);
            }
            else {
                System.out.println("Invalid postfix operation");
                System.exit(-1);
            }
        }
    )
| #(MINUSMINUS a=expr
        {
            if(a instanceof JkrInt) {
                b = new JkrInt(JkrInt.intValue(a) - 1);
                interpreter.assign(a, b);
            }
            else {
                System.out.println("Invalid postfix operation");
                System.exit(-1);
            }
        }
    )
| #(SHIFTR a=expr
        shiftr_add_exp:.
        {
            b = expr(shiftr_add_exp);
            if(a instanceof JkrPack && b instanceof JkrInt)
            {
                r = ((JkrPack)a).pop(JkrInt.intValue(b));
            } else {
                System.err.println("illegal pack operation");
                System.exit(-1);
            }
        }
   )
```

```
| #(SHIFTL a=expr
        shiftl_add_exp:.
        {
            b = expr(shiftl_add_exp);
            if(a instanceof JkrPack && b instanceof JkrPack)
            {
                ((JkrPack)a).push((JkrPack)b);
            }
            else if(a instanceof JkrPack && b instanceof JkrCard)
            {
                JkrPack tmp = new JkrPack(((JkrCard)b));
                ((JkrPack)a).push(tmp);
            }
            else {
                System.err.println("illegal pack operation");
                System.exit(-1);
            }
        }
   )
| #(PLUSEQ a=expr
        pluseq_add_exp:.
        {
            b = expr(pluseq_add_exp);
            if(a instanceof JkrPack && b instanceof JkrPack)
            {
                r = ((JkrPack)a).enqueue((JkrPack)b);
            }
            else if(a instanceof JkrPack && b instanceof JkrCard)
            {
                JkrPack tmp = new JkrPack(((JkrCard)b));
                ((JkrPack)a).enqueue(tmp);
            }
            else {
                System.err.println("illegal pack operation");
                System.exit(-1);
            }
        }
   )
| #(MINUSEQ a=expr
        minuseq_add_exp:.
        {
            b = expr(minuseq_add_exp);
            if(a instanceof JkrPack && b instanceof JkrInt)
            {
                r = ((JkrPack)a).backPop(JkrInt.intValue(b));
            } else {
                System.err.println("illegal pack operation");
                System.exit(-1);
            }
        }
   )
| #(ATSIGN a=expr
        {
            if(a instanceof JkrPack)
                ((JkrPack)a).shuffle();
            else {
                System.err.println("illegal pack shuffle operation");
                System.exit(-1);
            }
        }
   )
| #(GE a=expr
        ge_card_exp:.
        {
            b=expr(#ge_card_exp);
            if(a instanceof JkrCard && b instanceof JkrCard)
            {
                if(a.compareTo(b) >= 0)
                    r = new JkrBoolean(true);
                else
                    r = new JkrBoolean(false);
            } else {
```

```
                System.err.println("Invalid card comparison.");
                System.exit(-1);
            }
        }
   )
| #(LE a=expr
        le_card_exp:.
        {
            b=expr(#le_card_exp);
            if(a instanceof JkrCard && b instanceof JkrCard)
            {
                if(a.compareTo(b) <= 0)
                    r = new JkrBoolean(true);
                else
                    r = new JkrBoolean(false);
            } else {
                System.err.println("Invalid card comparison.");
                System.exit(-1);
            }
        }
   )
| #(GT a=expr
        gt_card_exp:.
        {
            b=expr(#gt_card_exp);
            if(a instanceof JkrCard && b instanceof JkrCard)
            {
                if(a.compareTo(b) > 0)
                    r = new JkrBoolean(true);
                else
                    r = new JkrBoolean(false);
            } else {
                System.err.println("Invalid card comparison.");
                System.exit(-1);
            }
        }
   )
| #(LT a=expr
        lt_card_exp:.
        {
            b=expr(#lt_card_exp);
            if(a instanceof JkrCard && b instanceof JkrCard)
            {
                if(a.compareTo(b) < 0)
                    r = new JkrBoolean(true);
                else
                    r = new JkrBoolean(false);
            } else {
                System.err.println("Invalid card comparison.");
                System.exit(-1);
            }
        }
   )
| #(LANGLE a=expr
        langle_additive_expression:.
        {
            if( a instanceof JkrInt )
            {
                b=expr(#langle_additive_expression);
                if(JkrInt.intValue(a) < JkrInt.intValue(b)) {
                    r = new JkrBoolean(true);
                }
                else { r = new JkrBoolean(false); };
            }
        }
    )
| #(RANGLE a=expr
        rangle_additive_expression:.
        {
            if( a instanceof JkrInt )
            {
                b=expr(#rangle_additive_expression);
```

```
                if(JkrInt.intValue(a) > JkrInt.intValue(b)) { r = new JkrBoolean(true); }
                else { r = new JkrBoolean(false); }
            }
        }
    )
| #(LESSEQ a=expr
        lesseq_additive_expression:.
        {
            if( a instanceof JkrInt )
            {
                b=expr(#lesseq_additive_expression);
                if(JkrInt.intValue(a) <= JkrInt.intValue(b))
                {
                    r = new JkrBoolean(true);
                } else { r = new JkrBoolean(false); }
            }
        }
    )
| #(GRTREQ a=expr
        grtreq_additive_expression:.
        {
            if( a instanceof JkrInt )
            {
                b=expr(#grtreq_additive_expression);
                if(JkrInt.intValue(a) >= JkrInt.intValue(b))
                {
                    r = new JkrBoolean(true);
                } else { r = new JkrBoolean(false); }
            }
        }
    )
| #(EQUALS a=expr
        eq_relational_expression:.
        {
            //System.out.println("In EQUALS");

            b=expr(#eq_relational_expression);
            //System.out.println("== A type " + a + "\t" + a.typename());
            //System.out.println("== B type " + b + "\t" + b.typename());
            if( a instanceof JkrSuit && b instanceof JkrSuit )
            {

                if( ((JkrSuit)a).equals( (JkrSuit)b ) )
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrCard && b instanceof JkrCard )
            {
                char suit = ((JkrSuit)a).getSuit();
                char bsuit = JkrSuit.getSuitValue(((JkrString)b).str);
                if( ((JkrCard)a).equals((JkrCard)b) )
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrSuit && b instanceof JkrString )
            {
                char suit = ((JkrSuit)a).getSuit();
                char bsuit = JkrSuit.getSuitValue(((JkrString)b).str);

                //System.out.println("compare suit & string: " + suit + " == " + bsuit);
```

```
                if( suit == bsuit )
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrString && b instanceof JkrSuit )
            {
                char bsuit = ((JkrSuit)a).getSuit();
                char suit = JkrSuit.getSuitValue(((JkrString)b).str);
                if( suit == bsuit )
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrString )
            {
                if( ((JkrString)a).equals( (JkrString)b ) )
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrInt )
            {
                if(JkrInt.intValue(a) == JkrInt.intValue(b))
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrBoolean )
            {
                //System.out.println("in equals, a is boolean");
                if(((JkrBoolean)a).bool == ((JkrBoolean)b).bool)
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            } else {
                System.err.println("Invalid '==' operation");
                System.exit(-1);
            }
        }
    )
| #(NOTEQ a=expr
        noteq_relational_expression:.
        {
            //System.out.println("In NOTEQ");
            b=expr(#noteq_relational_expression);
```

```
if( a instanceof JkrSuit && b instanceof JkrSuit )
{

    if( !((JkrSuit)a).equals( (JkrSuit)b ) )
    {
        //System.out.println("evaluate to true");
        r = new JkrBoolean(true);
    } else
    {
        //System.out.println("evaluate to false");
        r = new JkrBoolean(false);
    }
}
else if( a instanceof JkrCard && b instanceof JkrCard )
{
    char suit = ((JkrSuit)a).getSuit();
    char bsuit = JkrSuit.getSuitValue(((JkrString)b).str);
    if( !((JkrCard)a).equals((JkrCard)b) )
    {
        //System.out.println("evaluate to true");
        r = new JkrBoolean(true);
    } else
    {
        //System.out.println("evaluate to false");
        r = new JkrBoolean(false);
    }
}
else if( a instanceof JkrSuit && b instanceof JkrString )
{
    char suit = ((JkrSuit)a).getSuit();
    char bsuit = JkrSuit.getSuitValue(((JkrString)b).str);
    if( suit != bsuit )
    {
        //System.out.println("evaluate to true");
        r = new JkrBoolean(true);
    } else
    {
        //System.out.println("evaluate to false");
        r = new JkrBoolean(false);
    }
}
else if( a instanceof JkrString && b instanceof JkrSuit )
{
    char bsuit = ((JkrSuit)a).getSuit();
    char suit = JkrSuit.getSuitValue(((JkrString)b).str);
    if( suit != bsuit )
    {
        //System.out.println("evaluate to true");
        r = new JkrBoolean(true);
    } else
    {
        //System.out.println("evaluate to false");
        r = new JkrBoolean(false);
    }
}
else if( a instanceof JkrString )
{
    if( !((JkrString)a).equals( (JkrString)b ) )
    {
        //System.out.println("evaluate to true");
        r = new JkrBoolean(true);
    } else
    {
        //System.out.println("evaluate to false");
        r = new JkrBoolean(false);
    }
}
else if( a instanceof JkrInt )
{
    if(JkrInt.intValue(a) != JkrInt.intValue(b))
    {
        //System.out.println("evaluate to true");
```

```
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            }
            else if( a instanceof JkrBoolean )
            {
                if(((JkrBoolean)a).bool != ((JkrBoolean)b).bool)
                {
                    //System.out.println("evaluate to true");
                    r = new JkrBoolean(true);
                } else
                {
                    //System.out.println("evaluate to false");
                    r = new JkrBoolean(false);
                }
            } else {
                System.err.println("Invalid '!=' operation");
                System.exit(-1);
            }
        }
    )
| #(OR a=expr
        or_and_expression:.
        {
            //System.out.println("in OR");
            // get the sibling
            if ( a instanceof JkrBoolean )
                r = ( ((JkrBoolean)a).bool ? a : expr(#or_and_expression) );
            else
                r = a.or( expr(#or_and_expression) );
        }
   )
| #(AND a=expr
        and_equality_expression:.
        {
            if(a instanceof JkrBoolean)
                r = ( ((JkrBoolean)a).bool ? expr(#and_equality_expression) : a );
            else
                r = a.and( expr(#and_equality_expression) );
        }
    )
| #(DECL a=expr // a is type specifier
        identifier:. // identifier
        {
            if(a instanceof JkrCard) {
                //System.out.println("declare a new JkrCard");
                while(identifier != null)
                {
                    AST init_value = identifier.getFirstChild();
                    if(init_value == null) {
                        //System.out.println("no init value");
                        r = interpreter.assign(new JkrCard(), identifier.getText(), null);
                    }
                    else {
                        //System.out.println("Child: " + init_value.toStringList());
                        r = interpreter.assign(new JkrCard(), identifier.getText(),
                          expr(init_value));
                    }
                    identifier = identifier.getNextSibling();
                    //interpreter.printSymbols();
                }
            }
            else if(a instanceof JkrPack) {
                //System.out.println("declare a new JkrPack");
                while(identifier != null)
                {
                    AST init_value = identifier.getFirstChild();
                    if(init_value == null) {
                        //System.out.println("no init value");
```

```
            r = interpreter.assign(new JkrPack(), identifier.getText(), null);
        }
        else {
            //System.out.println("Child: " + init_value.toStringList());
            r = interpreter.assign(new JkrPack(), identifier.getText(),
                  expr(init_value));
        }
        identifier = identifier.getNextSibling();
        //interpreter.printSymbols();
    }
}
else if(a instanceof JkrString) {
    //System.out.println("declare a new JkrString");
    while(identifier != null)
    {
        AST init_value = identifier.getFirstChild();
        if(init_value == null) {
            //System.out.println("no init value");
            r = interpreter.assign(new JkrString(), identifier.getText(), null);
        }
        else {
            //System.out.println("Child: " + init_value.toStringList());
            if(expr(init_value) instanceof JkrString)
            r = interpreter.assign(new JkrString(), identifier.getText(),
                  expr(init_value));
        }
        identifier = identifier.getNextSibling();
        //interpreter.printSymbols();
    }
}
else if(a.typename().equals("int"))
{
    while(identifier != null)
    {
        AST init_value = identifier.getFirstChild();
        if(init_value == null) {
            //System.out.println("no init value");
            r = interpreter.assign(new JkrInt(), identifier.getText(), null);
        }
        else {
            //System.out.println("Child: " + init_value.toStringList());
            r = interpreter.assign(new JkrInt(), identifier.getText(),
                  expr(init_value));
        }
        identifier = identifier.getNextSibling();
        //interpreter.printSymbols();
    }
}
else if(a.typename().equals("boolean"))
{
    //System.out.println("boolean is the type specifier");
    while(identifier != null)
    {
        AST init_value = identifier.getFirstChild();
        if(init_value != null)
        {
            JkrBoolean i_value = (JkrBoolean)expr(init_value);
            //System.out.println("id: " + identifier.getText() + ", ival: " +
              i_value);
            r = interpreter.assign(new JkrBoolean(), identifier.getText(), i_value);
            //interpreter.printSymbols();
        }
        else
        {
            r = interpreter.assign(new JkrBoolean(), identifier.getText(), null);
            //interpreter.printSymbols();
        }
        identifier = identifier.getNextSibling();
    }
}
else {
    System.err.println("Wrong declaraction");
```

```
                System.exit(-1);
            }
        }
    )
| #(DECL_EMPTY_ARR
        a=expr // a is the type specifier
        id_tree:.
        {
            AST empty_arr_id = #id_tree;
            if(empty_arr_id == null) {
                System.err.println("Invalid array declaration. Requires identifier");
                System.exit(-1);
            }
            boolean hasValues = false;
            AST values = id_tree.getNextSibling();

            if(values == null)
                hasValues = false;
            else
                hasValues = true;

            int values_count = 0;
            AST values_list = null;
            if(hasValues) {
                values_list = values.getFirstChild();
                //System.out.println("values_list: " + values_list.toStringList());

                while(values_list != null) {
                    values_count++;
                    values_list = values_list.getNextSibling();
                }
            } else {
                System.err.println("Invalid array declaration: Need assignment");
                System.exit(-1);
            }

            //reset values list
            values_list = values.getFirstChild();
            //System.out.println("Num of values: " + values_count);
            if(a instanceof JkrInt)
            {
                interpreter.assignArray(new JkrInt(), empty_arr_id.getText(),
                                                        new JkrInt[values_count]);
                JkrArray newArray = (JkrArray)interpreter.getVariable(empty_arr_id.getText());

                for(int i = 0; i < values_count; i++) {
                    newArray.setElementAt(i, expr(#values_list));
                    values_list = values_list.getNextSibling();
                }
                interpreter.setValue(empty_arr_id.getText(), newArray);
                newArray.printElements();
            }
            else if(a instanceof JkrBoolean)
            {
                interpreter.assignArray(new JkrBoolean(), empty_arr_id.getText(),
                                                        new JkrBoolean[values_count]);
                JkrArray newArray = (JkrArray)interpreter.getVariable(empty_arr_id.getText());

                for(int i = 0; i < values_count; i++) {
                    newArray.setElementAt(i, expr(#values_list));
                    values_list = values_list.getNextSibling();
                }
                interpreter.setValue(empty_arr_id.getText(), newArray);
                newArray.printElements();
            }
            else if(a instanceof JkrCard)
            {
                interpreter.assignArray(new JkrCard(), empty_arr_id.getText(),
                                                        new JkrCard[values_count]);
                JkrArray newArray = (JkrArray)interpreter.getVariable(empty_arr_id.getText());

                for(int i = 0; i < values_count; i++) {
```

```
                    newArray.setElementAt(i, expr(#values_list));
                    values_list = values_list.getNextSibling();
                }
                interpreter.setValue(empty_arr_id.getText(), newArray);
            }
            else if(a instanceof JkrPack)
            {
                interpreter.assignArray(new JkrPack(), empty_arr_id.getText(),
                                                                    new JkrPack[values_count]);
                JkrArray newArray = (JkrArray)interpreter.getVariable(empty_arr_id.getText());

                for(int i = 0; i < values_count; i++) {
                    newArray.setElementAt(i, expr(#values_list));
                    values_list = values_list.getNextSibling();
                }
                interpreter.setValue(empty_arr_id.getText(), newArray);
            }
            else
            {
                System.err.println("Invalid array declaration. Wrong identifier");
                System.exit(-1);
            }
        }
   )
| #(ACCESS
        access_tree:.
        {
            //System.out.println("Begin of access");
            //System.out.println(access_tree.toStringList());

            AST access_tree_pexpr = #access_tree;
            AST access_tree_aexpr = access_tree_pexpr.getNextSibling();

            if(access_tree_pexpr==null || access_tree_aexpr==null) {
                System.err.println("Invalid array/pack access");
                System.exit(-1);
            }

            a = expr(access_tree_pexpr);
            if(a instanceof JkrPack) {
                //System.out.println("----in JkrPack");
                b = expr(access_tree_aexpr);
                if(!(b instanceof JkrInt)) {
                    System.err.println("Invalid pack access:"
                                    + #access_tree_pexpr.getText()
                                    + " is not an integer.");
                    System.exit(-1);
                }
                r = ((JkrPack)a).getNewCard(JkrInt.intValue(b));

                if(r == null) {
                    System.err.println("Invalid pack access: invalid pack index");
                    System.exit(-1);
                }
            } else {

            if(!(a instanceof JkrArray)) {
                System.err.println("Invalid array access: "
                                    + #access_tree_pexpr.getText()
                                    + " is not an array.");
                System.exit(-1);
            }
            b = expr(access_tree_aexpr);
            if(!(b instanceof JkrInt)) {
                System.err.println("Invalid array access: "
                                    + #access_tree_pexpr.getText()
                                    + " is not an integer.");
                System.exit(-1);
            }

            r = ((JkrArray)a).getElementAt(JkrInt.intValue(b));
```

```
            }
        }
   )
| #(DECL_ARR a=expr
        bracket_list:.
        {
            AST arr_identifier = bracket_list.getNextSibling();
            if(arr_identifier == null) {
                System.err.println("Invalid array declaration. Requires identifier");
                System.exit(-1);
            }

            r = expr(bracket_list);

            if(!(r instanceof JkrInt)) {
                System.err.println("Invalid array declaration. Bracket list requires integer");
                System.exit(-1);
            }

            if(a instanceof JkrInt)
            {
                interpreter.assignArray(new JkrInt(), arr_identifier.getText(),
                                                                    new JkrInt[JkrInt.intValue(r)]);
            }
            else if(a instanceof JkrBoolean)
            {
                interpreter.assignArray(new JkrBoolean(), arr_identifier.getText(),
                  new JkrBoolean[JkrInt.intValue(r)]);
            }
            else if(a instanceof JkrCard)
            {
                interpreter.assignArray(new JkrCard(), arr_identifier.getText(),
                                                                    new JkrCard[JkrInt.intValue(r)]);
            }
            else if(a instanceof JkrPack)
            {
                interpreter.assignArray(new JkrPack(), arr_identifier.getText(),
                                                                    new JkrPack[JkrInt.intValue(r)]);
            }
            else
            {
                System.err.println("Invalid array declaration. Wrong identifier");
                System.exit(-1);
            }

           //interpreter.printSymbols();
        }
   )
| #(RANK_ROW r_element_rules:. // the element rules
        {
            while(r_element_rules != null)
            {
                //System.out.println("In r_element_rules: " + r_element_rules.toStringList());
                AST er_identifier = r_element_rules.getFirstChild();
                AST value = er_identifier.getFirstChild();

                //System.out.println("`````id: " + er_identifier.getText());

                pack.addHierarchyCardRule(er_identifier.getText(), value.getText());
                cards.add(new Integer(JkrCard.getCardNumber(er_identifier.getText())));
                //System.out.println("Added: Card- " + er_identifier.getText() + " Value- " +
                  value.getText());

                AST eq_identifier = er_identifier.getNextSibling();
                while(eq_identifier != null)
                {
                    //System.out.println("`````id: " + eq_identifier.getText());
                    pack.addHierarchyCardRule(eq_identifier.getText(), value.getText());
                    cards.add(new Integer(JkrCard.getCardNumber(eq_identifier.getText())));
                    eq_identifier = eq_identifier.getNextSibling();
                }
```

```
                r_element_rules = r_element_rules.getNextSibling();
            }
        }
    )
| #(SUIT_ROW s_element_rules:. // the element rules
        {
            while(s_element_rules != null)
            {
                //System.out.println("In s_element_rules: " + s_element_rules.toStringList());
                AST er_identifier = s_element_rules.getFirstChild();
                AST value = er_identifier.getFirstChild();

                String suit_string = er_identifier.getText();
                //System.out.println("suit_string: " + suit_string);
                char suit_char = JkrSuit.getSuitValue(suit_string);
                suit_string = "" + suit_char;
                //System.out.println("suit_char: " + suit_string);

                pack.addHierarchySuitRule(suit_string, value.getText());
                suits.add(new Character(JkrSuit.getSuitValue(er_identifier.getText())));

                AST eq_identifier = er_identifier.getNextSibling();
                while(eq_identifier != null)
                {
                    String a_suit_string = eq_identifier.getText();
                    char a_suit_char = JkrSuit.getSuitValue(a_suit_string);
                    a_suit_string = "" + a_suit_char;

                    pack.addHierarchySuitRule(a_suit_string, value.getText());
                    suits.add(new Character(JkrSuit.getSuitValue(eq_identifier.getText())));
                    eq_identifier = eq_identifier.getNextSibling();
                }

                s_element_rules = s_element_rules.getNextSibling();
            }
        }
    )
| #(DECL_HIER a=expr b=expr
        decl_hier_id:.
        {
            if(decl_hier_id != null) {
                //System.out.println("DECL_HIER");
                //System.out.println("pack id: " + decl_hier_id.getText());
                pack.initPack(cards, suits);

                JkrDataType tmp = interpreter.getVariable(#decl_hier_id.getText());
                interpreter.assign(tmp, pack);

                //pack.printHierarchy();
                //pack.printPack();

                //add pack to the symbol table with getText() as the name
            } else {
                System.err.println("Invalid hierarchy declaration: Need to assign to a new
                 pack");
                System.exit(-1);
            }
        }
    )
| #(ASSIGN
        assign_tree:.
        {
            //System.out.println("assign");
            //System.out.println(assign_tree.toStringList());

            if(#assign_tree.getText().equals("attrib")) {
                AST atree = assign_tree.getFirstChild();
                AST aval = assign_tree.getNextSibling();

                //System.out.println("attr: " + atree.toStringList());
                //System.out.println(atree.getNextSibling().getText());
                //System.out.println("op: " + aval.toStringList());
```

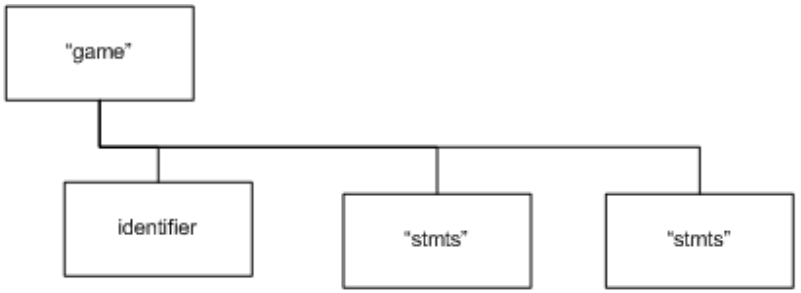
```
                a = expr(#atree);
                String ops = atree.getNextSibling().getText();
                c = expr(aval);

                if(a instanceof JkrCard) {
                    if(ops != null && ops.equals("private")){
                        if(c instanceof JkrBoolean) {
                            ((JkrCard)a).setViewability( ((JkrBoolean)c).bool );
                            interpreter.setValue(#atree.getText(), ((JkrCard)a));
                        } else {
                            System.err.println("invalid attribute assignment");
                            System.exit(-1);
                        }
                    }
                }
            }
            else if(#assign_tree.getText().equals("access"))
            {
                AST atree = assign_tree.getFirstChild();
                AST aval = assign_tree.getNextSibling();

                a = expr(#atree); // array name
                b = expr(#atree.getNextSibling()); // array index
                c = expr(aval);

                if(!(a instanceof JkrArray) || !(b instanceof JkrInt))
                {
                    System.err.println("Invalid array access");
                    System.exit(-1);
                }

                boolean validArrayAssign = false;

                if(((JkrArray)a).array instanceof JkrInt[] && c instanceof JkrInt)
                    validArrayAssign = true;
                else if(((JkrArray)a).array instanceof JkrBoolean[] && c instanceof JkrBoolean)
                    validArrayAssign = true;
                else if(((JkrArray)a).array instanceof JkrCard[] && c instanceof JkrCard)
                    validArrayAssign = true;
                else  if(((JkrArray)a).array instanceof JkrPack[] && c instanceof JkrPack)
                    validArrayAssign = true;
                else
                    validArrayAssign = false;

                if(validArrayAssign)
                {
                    JkrArray newArray = (JkrArray)a;
                    newArray.setElementAt(JkrInt.intValue(b), c);
                    interpreter.setValue(#atree.getText(), newArray);
                }
                else
                {
                    System.err.println("Invalid array assignment, datatype mismatch");
                    System.exit(-1);
                }
            }
            else {
                a = interpreter.getVariable(#assign_tree.getText());

                if(assign_tree.getNextSibling() != null) {
                    b = expr(assign_tree.getNextSibling());
                    r = interpreter.assign(a, b);
                } else {
                    System.err.println("illegal assignment operation");
                    System.exit(-1);
                }

            }
        }
    )
| #(PLUS a=expr
```

```
        b0:.
        {
            //System.out.println("plus operator");
            JkrDataType adder = expr(b0);

            if(a instanceof JkrInt && adder instanceof JkrInt)
            {
                //System.out.println("adding ints");
                r = a.plus(adder);
            }
            else if(a instanceof JkrString || adder instanceof JkrString)
            {
                //System.out.println("concat strings");
                r = new JkrString(a.toString() + adder.toString());
            }
            else
            {
                System.err.println("Illegal '+' operation");
                System.exit(-1);
            }
        }
   )
| #(MINUS a=expr
        b1:.
        {
            if(a instanceof JkrInt)
            {
                JkrInt adder = (JkrInt)expr(b1);
                r = a.minus(adder);
            }
        }
   )
| #(TIMES a=expr
        b2:.
        {
            if(a instanceof JkrInt)
            {
                JkrInt adder = (JkrInt)expr(b2);
                r = a.mult(adder);
            }
        }
   )
| #(MOD a=expr
        b3:.
        {
            if(a instanceof JkrInt)
            {
                JkrInt adder = (JkrInt)expr(b3);
                r = a.mod(adder);
            }
        }
   )
| num:NUMBER
    {
        //System.out.println("num");
        r = interpreter.getInteger(num.getText());
    }
| str:STRING
    {
        //System.out.println("STRING: " + new JkrString(str.getText()));
        r = new JkrString(str.getText());
    }
| "string"
    {
        r = new JkrString();
    }
| "true"
    {
        //System.out.println("true");
        r = new JkrBoolean(true);
    }
| "false"
```

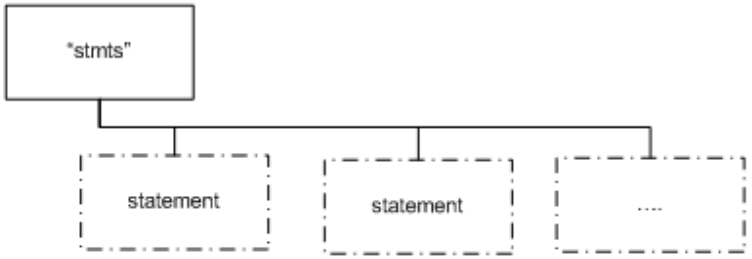
```
    {
        //System.out.println("false");
        r = new JkrBoolean(false);
    }
| "int"
    {
        //System.out.println("int");
        r = new JkrInt();
    }
| "boolean"
    {
        //System.out.println("boolean");
        r = new JkrBoolean();
    }
| "card"
    {
        //System.out.println("card");
        r = new JkrCard();
    }
| "pack"
    {
        //System.out.println("pack");
        r = new JkrPack();
    }
| #(EXPR expr_assignment_expr:.
        {
            while(expr_assignment_expr != null)
            {
                //System.out.println("in EXPR");
                //System.out.println(expr_assignment_expr.toStringList());
                AST assgn_expr = expr_assignment_expr;
                //System.out.println(assgn_expr.toStringList());

                if(assgn_expr != null) {
                    r = expr(assgn_expr);
                }
                else {
                    System.err.println("illegal expression");
                    System.exit(-1);
                }
                expr_assignment_expr = assgn_expr.getNextSibling();
            }
        }
    )
| #(EXPR_STMT r=expr)

| "break"
    {
        //System.out.println("break");
        interpreter.setBreak(true);
    }
| "continue"
    {
        //System.out.println("continue");
        interpreter.setContinue(true);
    }
| #(id:ID
        {
            //System.out.println("ID: " + id.getText());
            r = interpreter.getVariable( id.getText());
            //System.out.println("after after variable in ID");
        }
    )
| #("game"
        // a is first children of game. st1 is the subtree with the 2nd sibling as a root.
        a=expr st1:.
        {
            //System.out.println("st1 " + st1.toStringList());
            b=expr(#st1);
            //System.out.println("st1 " + st1.getNextSibling().toStringList());
            c=expr(#st1.getNextSibling());
        }
```

```
    )
| #(STATEMENTS
        //st2:.
        {
            if(_t != null)
            {
                AST st2 = _t;
                while(st2 != null)
                {
                    r=expr(#st2);
                    st2 = st2.getNextSibling();
                }
            }
        }
    )
;
```

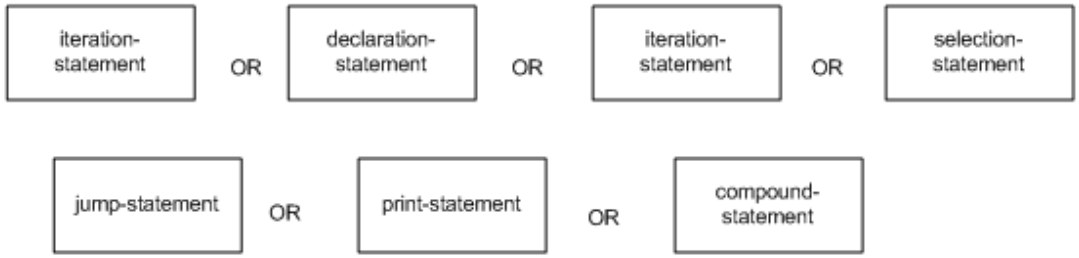
Joker : an AST diagram
last modified 2003 dec 13



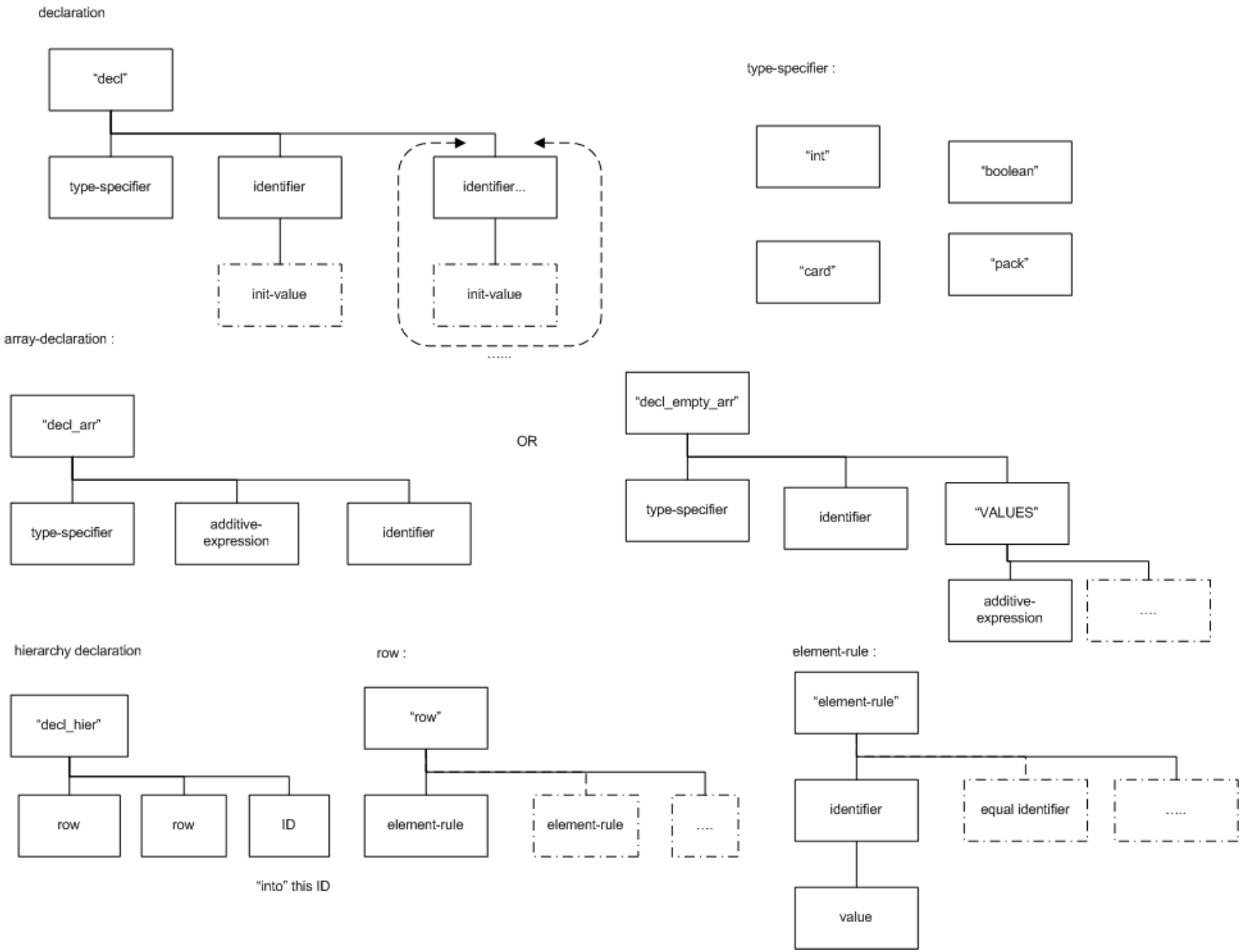
compound-statement



statement :

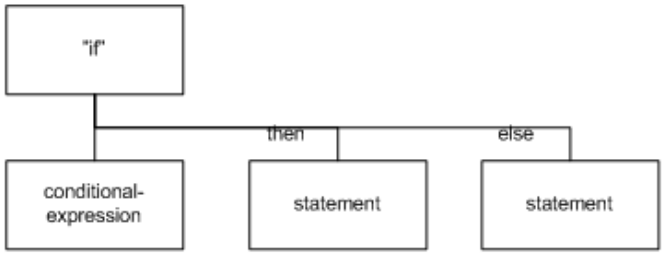


declaration - statement :


declaration
"decl"
type-specifier
identifier
identifier...
init-value
init-value
.....
type-specifier :
"int"
"boolean"
"card"
"pack"
array-declaration :
"decl_arr"
type-specifier
additive-expression
identifier
OR
"decl_empty_arr"
type-specifier
identifier
"VALUES"
additive-expression
....
hierarchy declaration
"decl_hier"
row
row
ID
"into" this ID
row :
"row"
element-rule
element-rule
....
element-rule :
"element-rule"
identifier
equal identifier
.....
value

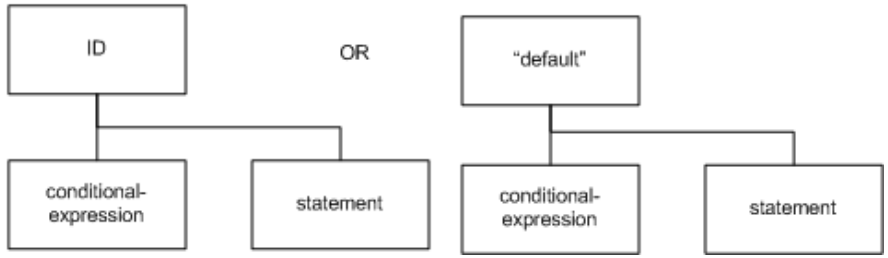
expression - statement :

if statement

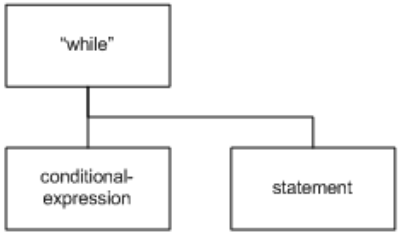


option statement :

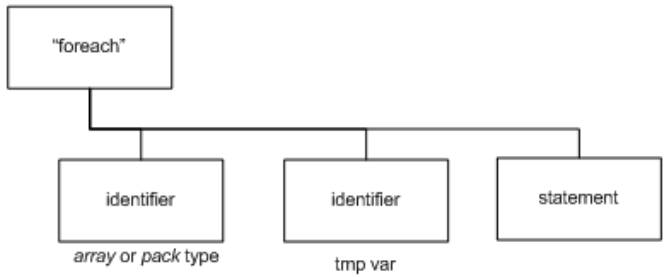
option case :



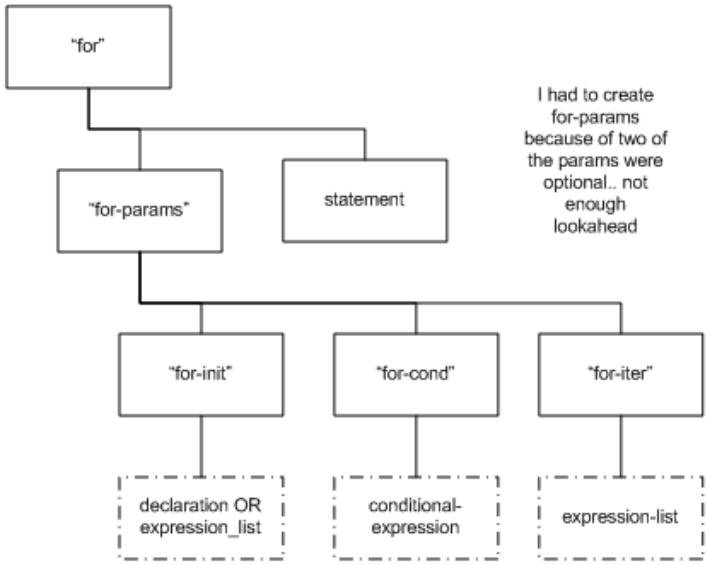
while statement



for statement



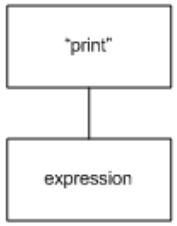
for statement :



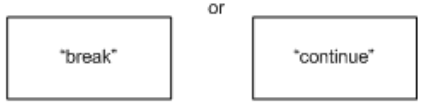
I had to create for-params because of two of the params were optional.. not enough lookahead

if there's no conditional expression.. then you should use the value of TRUE, right?

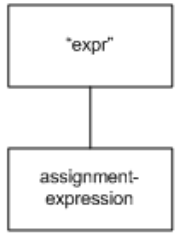
print-statement



jump statement



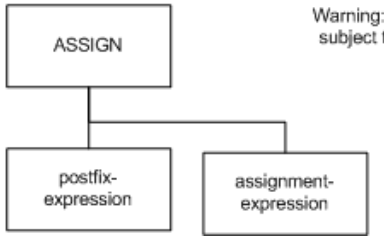
expression



expression-list

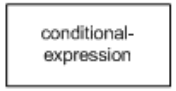
.....and more

assignment-expression

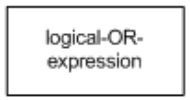


Warning: this part is subject to change.

OR

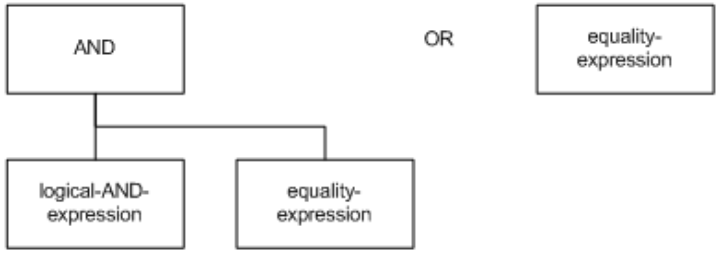


conditional-expression

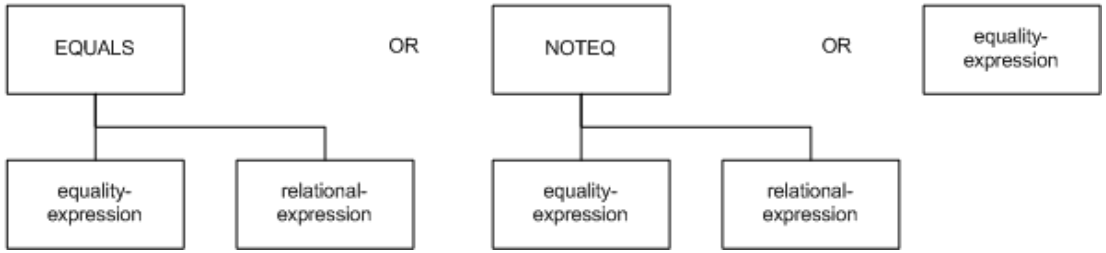


logical-OR-expression

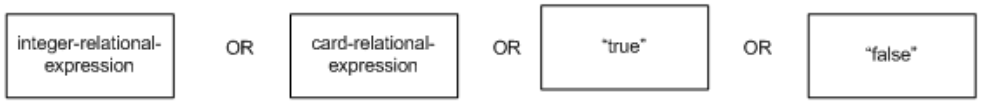
logical-AND-expression



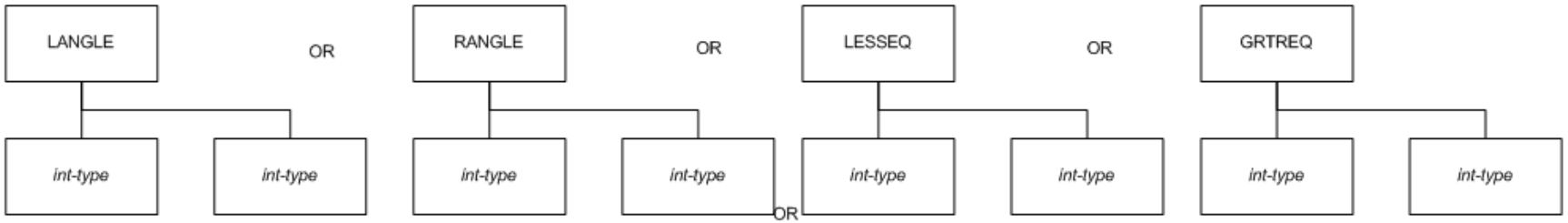
equality-expression



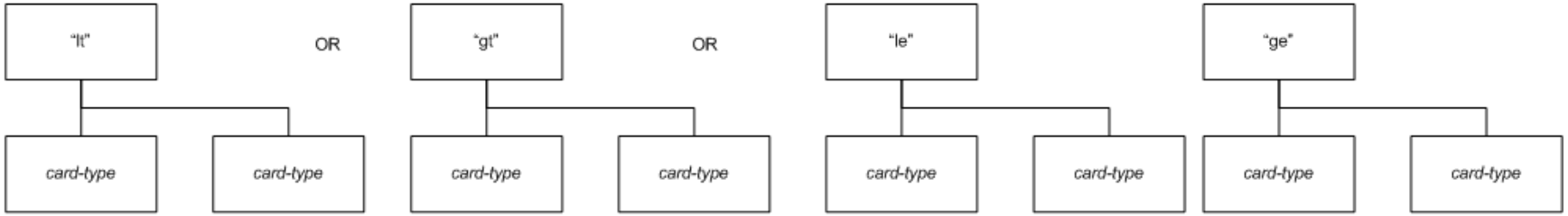
relational-expression :



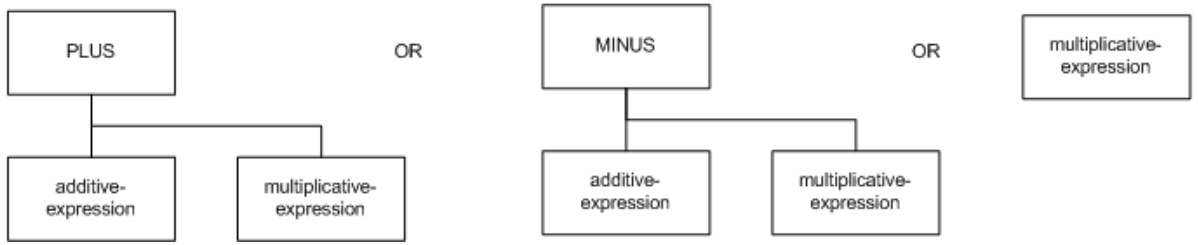
integer-relational-expression :



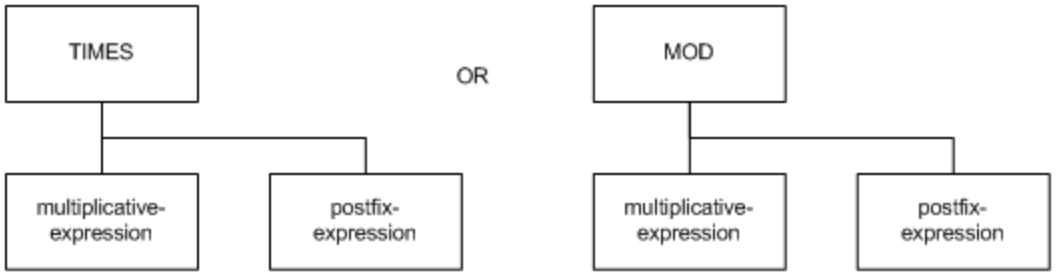
card-relational-expression :



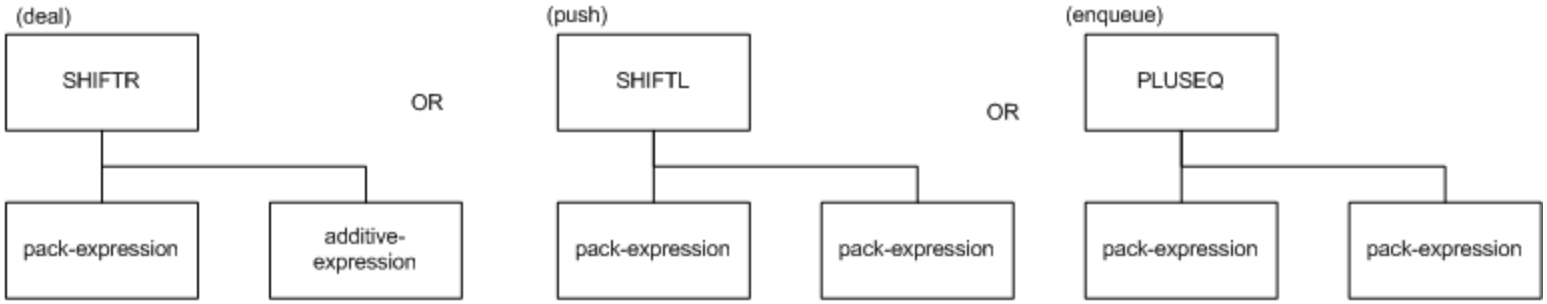
additive-expression :

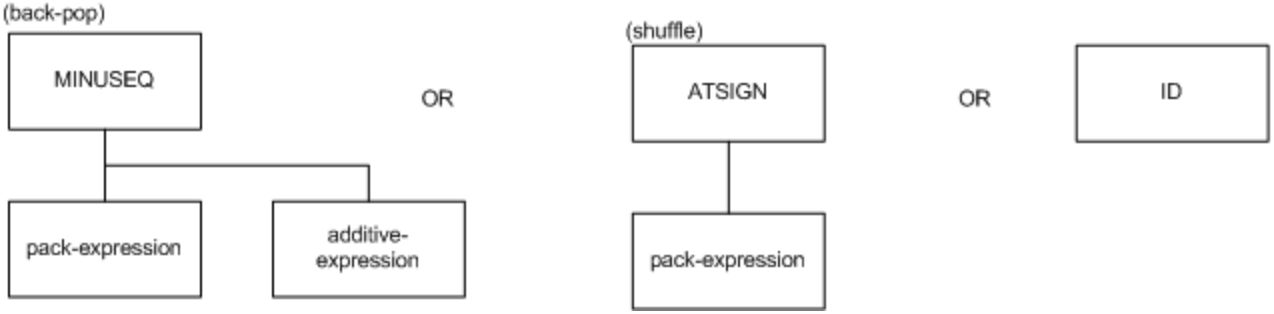


multiplicative-expression :

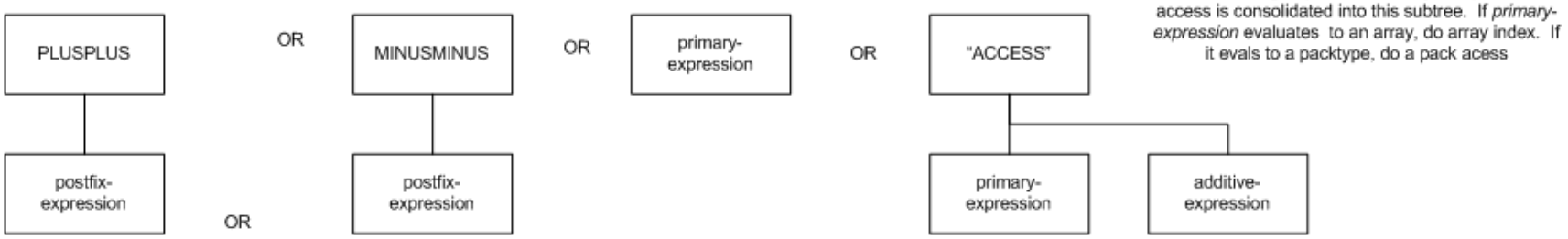


pack-expression :



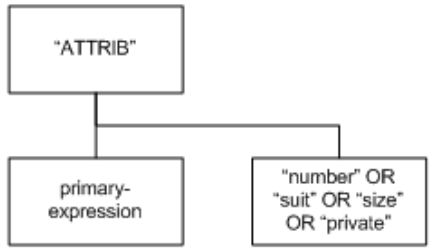


postfix-expression



Comment: forget "packess". Array and pack index access is consolidated into this subtree. If *primary-expression* evaluates to an array, do array index. If it evals to a packtype, do a pack acess

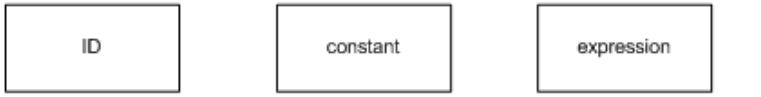
OR



Note: number, suit, and private only can apply to cards.

Size applies to packs and arrays.

primary-expression :



(with discarded parens)

constant :

(integer-constant)

(string-constant) ?

8.4 Java BackEnd

8.3.1 JkrArray.java

```
/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 11, 2003<br>
 * Time: 10:45:52 PM<br>
 * <br>
 *
 * Represents an 'array' type in the Joker programming language containing JkrDataType objects
 */
public class JkrArray extends JkrDataType
{
    JkrDataType[] array;

    /**
     * Construstor
     *
     * @param a array of objects stored in this instance of JkrArray
     */
    public JkrArray(JkrDataType[] a)
    {
        array = a;
    }

    /**
     * Construstor calling parent 'JkrDataType' constructor
     */
    public JkrArray()
    {
        super();
    }

    /**
     * Gets the type name of this object
     *
     * @return object type
     */
    public String typename()
    {
        return "array";
    }

    /**
     * Creates a new copy of this object copying each element from this object's array
     *
     * @return clone of this object instance
     */
    public JkrDataType copy()
    {
        JkrDataType[] temp = new JkrDataType[array.length];

        for(int i = 0; i < temp.length; i++) //copy each element
        {
            temp[i] = array[i];
        }

        return new JkrArray(temp); //return a clone
    }

    /**
     * Accessor method to get an element at a specified index.  Error if index is out of
     * bounds.
     *
     * @param i index into the array of the object to be retrieved
     * @return the object at the ith index of the array
```

```
     */
    public JkrDataType getElementAt(int i)
    {
        if(i < array.length) //validate index range
            return array[i];
        else //exit if index out of bounds
        {
            System.err.println("Error:  Array index out of bounds @ '" + i + "'");
            System.exit(-1);
            return array[0];
        }
    }

    /**
     * Method to set an element at the specified index in the array
     *
     * @param i index into the array to be set to the object j
     * @param j set the specified object in the array at index i to object j
     */
    public void setElementAt(int i, JkrDataType j)
    {
        array[i] = j;
    }

    /**
     * Get the array size
     *
     * @return array size
     */
    public int length()
    {
        return array.length;
    }

    /**
     * Prints comma delimited list of array elements
     */
    public void printElements()
    {
        String str = "";

        str += array[0].toString();

        for(int i = 1; i < array.length; i++)
        {
            str += ", " + array[i].toString();
        }

        System.out.println("Array Elements: " + str);
    }

    /**
     * String representation of an array reveals type of objects in array and its size
     *
     * @return string representation of array
     */
    public String toString()
    {
        if(array instanceof JkrInt[])
            return "int array [" + array.length + "]";
        else if(array instanceof JkrBoolean[])
            return "boolean array [" + array.length + "]";
        else if(array instanceof JkrCard[])
            return "card array [" + array.length + "]";
        else if (array instanceof JkrPack[])
            return "pack array [" + array.length + "]";
        else
            return "null";
    }
}
```

8.3.2 JkrBoolean.java

```
/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 1, 2003<br>
 * Time: 1:17:50 AM<br>
 * <br>
 *
 * Represents a 'boolean' type in the Joker programming language
 */
public class JkrBoolean extends JkrDataType
{
    public static final String TRUE = "true";
    public static final String FALSE = "false";

    boolean bool;

    /**
     * Construstor
     *
     * @param b boolean value
     */
    public JkrBoolean(boolean b)
    {
        bool = b;
    }

    /**
     * Construstor calling parent 'JkrDataType' constructor and initializing boolean value to false
     */
    public JkrBoolean()
    {
        super();
        bool = false;
    }

    /**
     * Creates a new copy of this object
     *
     * @return clone of this object instance
     */
    public JkrDataType copy()
    {
        return new JkrBoolean(bool);
    }

    /**
     * Gets the type name of this object
     *
     * @return object type
     */
   public String typename()
    {
        return "boolean";
    }

    /**
     * Implements 'and' (&&) logical operation between two JkrBoolean objects
     *
     * @param j another JkrBoolean object
     * @return true if both JkrBoolean objects are true, otherwise false
     */
    public JkrDataType and(JkrDataType j)
    {
        if(j instanceof JkrBoolean)
            return new JkrBoolean(bool && ((JkrBoolean) j).bool);

        return error(j, "and");
    }

    /**
```

```
     * Implements 'or' (||) logical operation between two JkrBoolean objects
     *
     * @param j another JkrBoolean object
     * @return true if either JkrBoolean object is true, otherwise false
     */
    public JkrDataType or(JkrDataType j)
    {
        if(j instanceof JkrBoolean)
            return new JkrBoolean(bool || ((JkrBoolean) j).bool);

        return error(j, "or");
    }

    /**
     * Implements 'not' (!) logical operation between two JkrBoolean objects
     *
     * @return opposite value of JkrBoolean value
     */
    public JkrDataType not()
    {
        return new JkrBoolean(!bool);
    }

    /**
     * Compares two JkrBoolean objects
     *
     * @param j another JkrBoolean object
     * @return true if both objects are equal, false otherwise
     */
    public JkrDataType eq(JkrDataType j)
    {
        if(j instanceof JkrBoolean)
            return new JkrBoolean((bool && ((JkrBoolean) j).bool) || (!bool && !((JkrBoolean)
j).bool));
        else
            return error(j, "==");
    }

    /**
     * Implements 'not equal' (!=) logical comparison between two JkrBoolean objects
     *
     * @param j another JkrBoolean object
     * @return true if objects not equal, false otherwise
     */
    public JkrDataType notEquals(JkrDataType j)
    {
        if(j instanceof JkrBoolean)
            return new JkrBoolean((bool && !((JkrBoolean) j).bool) || (!bool && ((JkrBoolean)
j).bool));
        else
            return error(j, "!=");
    }

    /**
     * String representation of this JkrBoolean object
     *
     * @return "true" if true, "false" otherwise
     */
    public String toString()
    {
        if(bool)
            return JkrBoolean.TRUE;
        else
            return JkrBoolean.FALSE;
    }
}
```

8.3.3 JkrCard.java

```
/**
 * Name:  Timothy SooHoo<br>
 * Date: Nov 23, 2003<br>
 * Time: 5:47:25 PM<br>
 * <br>
 *
 * Represents a 'card' type in the Joker Programming Language.Cards include a number and a suit.
 * Numbers can range from 2-14 in which 11, 12, 13, 14 are JACK, QUEEN, KING, ACE respectively.
 */
public class JkrCard extends JkrDataType
{
    public static final String JACK = "JACK";
    public static final String QUEEN = "QUEEN";
    public static final String KING = "KING";
    public static final String ACE = "ACE";

    private int number;
    private JkrSuit suit;

    private int value;
    private boolean viewability; // the viewability of the card, default it's private

    /**
     * Constructor for JkrCard Class
     *
     * @param number range 2-14 in which 11, 12, 13, 14 are JACK, QUEEN, KING, ACE
     * respectively
     * @param suit suit for this card
     */
    public JkrCard(int number, JkrSuit suit)
    {
        this.number = number;
        this.suit = suit;
        this.viewability = false;
    }

    /**
     * Constructor for JkrCard Class
     *
     * @param number range 2-14 in which 11, 12, 13, 14 are JACK, QUEEN, KING, ACE
     * respectively
     * @param suit suit for this card
     * @param value hierarchical value of this card
     */
    public JkrCard(int number, JkrSuit suit, int value)
    {
        this.number = number;
        this.suit = suit;
        this.value = value;
        this.viewability = false;
    }

    /**
     * Constructor for JkrCard Class
     *
     * @param number range 2-14 in which 11, 12, 13, 14 are JACK, QUEEN, KING, ACE
     * respectively
     * @param suit char representing this suit using static constants in JkrSuit
     * class HEART, DIAMOND, SPADE, CLUB
     */
    public JkrCard(int number, char suit)
    {
        this.number = number;
        this.suit = new JkrSuit(suit);
        this.viewability = false;
    }

    /**
     * Constructor for JkrCard Class
     *
     * @param number range 2-14 in which 11, 12, 13, 14 are JACK, QUEEN, KING, ACE
     * respectively
```

```
 * @param suit char representing this suit using static constants in JkrSuit
 * class HEART, DIAMOND, SPADE, CLUB
 * @param value hierarchical value of this card
 */
public JkrCard(int number, char suit, int value)
{
    this.number = number;
    this.suit = new JkrSuit(suit);
    this.value = value;
    this.viewability = false;
}

/**
 * Construstor calling parent 'JkrDataType' constructor
 */
public JkrCard()
{
    super();
}

/**
 * Creates a new copy of this object
 *
 * @return clone of this object instance
 */
public JkrDataType copy()
{
    return new JkrCard(number, suit, value);
}

/**
 * Gets the type name of this object
 *
 * @return object type
 */
public String typename()
{
    return "card";
}

/**
 * Gets the number from a given string.  The string could be 2-10 or 'J', 'Q', 'K', 'A'
 *
 * @param s string representation of a card number
 * @return the integer value of this card number 2-14
 */
public static int getCardNumber(String s)
{
    try
    {
        return Integer.parseInt(s);
    }
    catch(NumberFormatException e)
    {
        if(s.equals("J"))
            return 11;
        else if(s.equals("Q"))
            return 12;
        else if(s.equals("K"))
            return 13;
        else if(s.equals("A"))
            return 14;
        else if(s.equals("ten"))
          return 10;
        else if(s.equals("nine"))
          return 9;
        else if(s.equals("eight"))
          return 8;
        else if(s.equals("seven"))
          return 7;
        else if(s.equals("six"))
          return 6;
```

```
        else if(s.equals("five"))
          return 5;
        else if(s.equals("four"))
          return 4;
        else if(s.equals("three"))
          return 3;
        else if(s.equals("two"))
          return 2;
        else
        {
            System.err.println("Error:  Invalid card number '" + s + "'");
            System.exit(-1);
            return -1;
        }
    }
}

/**
 * Accessor method to get number of this card
 *
 * @return card number 2-14 in which 11, 12, 13, 14 are JACK, QUEEN, KING, ACE
 * respectively
 */
public int getNumber()
{
    return number;
}

/**
 * Gets the rank (aka number) of this card
 *
 * @return value 2-14
 */
public JkrString getRank()
{
  if(number <= 10)
    return new JkrString(Integer.toString(number));
  else if(number == 11)
    return new JkrString("J");
  else if(number == 12)
      return new JkrString("Q");
  else if(number == 13)
      return new JkrString("K");
  else if( number == 14)
      return new JkrString("A");
  else
  {
      System.err.println("This card has in invalid rank");
      System.exit(-1);
      return new JkrString("");
  }
}

/**
 * Accessor method to get card's suit
 *
 * @return card's suit
 */
public JkrSuit getSuit()
{
    return suit;
}

/**
 * Accessor method to get the value of this card according to the hierarchy
 *
 * @return card's value according to the hierarchy of its owning pack
 */
public boolean getViewability()
{
    return viewability;
}
```

```
/**
 * Setter method to set the viewability of this card in the given hierarchy
 *
 * @param viewability viewability to give to this card
 */
public void setViewability(boolean viewability)
{
    this.viewability = viewability;
}

/**
 * Getter method to get the value of this card in the given hierarchy
 *
 * @return value of this card
 */
public int getValue()
{
    return value;
}

/**
 * Setter method to set the value of this card in the given hierarchy
 *
 * @param value value to give to this card
 */
public void setValue(int value)
{
    this.value = value;
}

/**
 * Setter method to set the number of this card
 *
 * @param number number value this card should take on
 */
public void setNumber(int number)
{
  this.number = number;
}

/**
 * Setter method to set this card's suit
 *
 * @param suit the suit of this card
 */
public void setSuit(JkrSuit suit)
{
  this.suit = suit;
}

/**
 * Prints card's number and suit in form:  'number' of 'suit' ie KING of SPADES
 *
 * @return card's number and suit
 */
public String toString()
{
    if(number <= 10)
        return number + " of " + suit;
    else
    {
        switch(number)
        {
            case 11:
                return JkrCard.JACK + " of " + suit;
            case 12:
                return JkrCard.QUEEN + " of " + suit;
            case 13:
                return JkrCard.KING + " of " + suit;
            case 14:
                return JkrCard.ACE + " of " + suit;
```

```
            default:
                System.err.println("Error:  Invalid card");
                System.exit(-1);
                return "";
        }
    }
}

/**
 * Compares the values of two cards based on the hierarchy this card's owning pack
 *
 * @param j card to compared to this card
 * @return -1 if less than c, 0 if equal, 1 if greater than c
 */
public int compareTo(JkrDataType j)
{
    JkrCard c = (JkrCard) j;

    if(value > c.value)
        return 1;
    else if(value < c.value)
        return -1;
    else
        return 0;
}

/**
 * Checks if two Cards have equal value
 *
 * @param c card to be compared to this card
 * @return true if cards are equal, false otherwise
 */
public boolean valEquals(JkrCard c)
{
    if(value == c.value)
        return true;
    else
        return false;
}

/**
 * Gets the String representation of a card with the given 'number'
 * 11, 12, 13, 14 correspond to J, Q, K, A while the numbers retain their same
 * value as a String
 *
 * @param number card's number value 2-14
 * @return card's String value using J, Q, K, A for face cards
 */
public static String getStrNumber(int number)
{
    if(number < 2)
    {
        System.err.println("Error:  Invalid card");
        System.exit(-1);
        return "";
    }
    else if(number <= 10)
        return "" + number;
    else
    {
        switch(number)
        {
            case 11:
                return "J";
            case 12:
                return "Q";
            case 13:
                return "K";
            case 14:
                return "A";
            default:
                System.err.println("Error:  Invalid card");
```

```
                    System.exit(-1);
                    return "";
                }
            }
        }

        /**
         * Override the equals method to compare two cards
         *
         * @param obj card to compare to this card
         * @return true if cards have matching number and suit, false otherwise
         */
        public boolean equals(Object obj)
        {
            JkrCard aCard = (JkrCard) obj;

            if(aCard.getSuit().equals(this.suit) && aCard.getNumber() == this.number)
                return true;
            else
                return false;
        }
}
```

8.3.4 JkrDataType.java

```
/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 1, 2003<br>
 * Time: 1:00:10 AM<br>
 * <br>
 *
 * Parent class of all data types utilized in the Joker Programming Language.
 * All child classes extend this general class.  This class is only used as a
 * parent with certain functions to be overridden by child classes that implement
 * a given functionality.
 */
public class JkrDataType
{
    String name;

    /**
     * Construstor, initializes name of this datatype to null
     */
    public JkrDataType()
    {
        name = null;
    }

    /**
     * Construstor initializing the name of this datatype
     *
     * @param name name bound to this instance
     */
    public JkrDataType(String name)
    {
        this.name = name;
    }

    /**
     * Creates a new copy of this object
     *
     * @return clone of this object instance
     */
    public JkrDataType copy()
    {
        return new JkrDataType();
    }

    /**
     * Gets the type name of this object.  JkrDataType is an unknown type, should be
```

```
 * overriden by child class.
 *
 * @return object type
 */
public String typename()
{
    return "unknown";
}

/**
 * Sets the name of this object
 *
 * @param name name to be bound to this instance
 */
public void setName(String name)
{
    this.name = name;
}

/**
 * Throw an exception with the specified message passed to this method
 *
 * @param msg error message to display
 * @return exception to be thrown
 */
public JkrDataType error(String msg)
{
    throw new JkrException(msg);
}

/**
 * Throw an exception with the specified message passed to this method along with
 * the violating object j
 *
 * @param j object involved in error producing operation
 * @param msg error message to display
 * @return exception to be thrown
 */
public JkrDataType error(JkrDataType j, String msg)
{
    if(j == null)
        return error(msg);
    else
    {
        String errorMessage = "illegal operation: " + msg + " (type=" + typename() +
                              ":  " + (name != null ? name : "?name?") + ")"
                              + " and (type=" + j.typename() + ":  " +
                              (j.name != null ? j.name : "?name?") + ")";
        throw new JkrException(errorMessage);
    }
}

/**
 * To be overridden by child classes that implement this arithmetic operation
 *
 * @param j another instance of JkrDataType
 * @return result of arithmetic operation
 */
public JkrDataType plus(JkrDataType j)
{
    return error(j, "+");
}

/**
 * To be overridden by child classes that implement this arithmetic operation
 *
 * @param j another instance of JkrDataType
 * @return result of arithmetic operation
 */
public JkrDataType minus(JkrDataType j)
{
    return error(j, "-");
```

```
}

/**
 * To be overridden by child classes that implement this arithmetic operation
 *
 * @param j another instance of JkrDataType
 * @return result of arithmetic operation
 */
public JkrDataType mult(JkrDataType j)
{
    return error(j, "*");
}

/**
 * To be overridden by child classes that implement this comparison operation
 *
 * @param j another instance of JkrDataType
 * @return result of comparison operation
 */
public JkrDataType eq(JkrDataType j)
{
    return error(j, "==");
}

/**
 * To be overridden by child classes that implement this comparison operation
 *
 * @param j another instance of JkrDataType
 * @return result of comparison operation
 */
public JkrDataType notEquals(JkrDataType j)
{
    return error(j, "!=");
}

/**
 * To be overridden by child classes that implement this comparison operation
 *
 * @param j another instance of JkrDataType
 * @return result of comparison operation
 */
public int compareTo(JkrDataType j)
{
    error(j, "compareTo");
    return -10;
}

/**
 * To be overridden by child classes that implement this logical operator
 *
 * @param j another instance of JkrDataType
 * @return result of logical operation
 */
public JkrDataType and(JkrDataType j)
{
    return error(j, "and");
}

/**
 * To be overridden by child classes that implement this logical operator
 *
 * @param j another instance of JkrDataType
 * @return result of logical operation
 */
public JkrDataType or(JkrDataType j)
{
    return error(j, "or");
}

/**
 * To be overridden by child classes that implement this logical operator
 *
```

```
     * @return result of logical operation
     */
    public JkrDataType not()
    {
        return error("not");
    }

    /**
     * To be overridden by child classes that implement this arithmetic operation
     *
     * @param j another instance of JkrDataType
     * @return result of arithmetic operation
     */
    public JkrDataType mod(JkrDataType j)
    {
        return error(j, "mod");
    }

    /**
     * To be overridden by child classes that implement a string representation of the object
     * instance
     *
     * @return object value as a string
     */
    public String toString()
    {
        error("Illegal operation: Cannot print " +
              (this.name == null ? "variable" : "'" + this.name + "'"));
        return "";
    }
}
```

8.3.5 JkrException.java

```
/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 1, 2003<br>
 * Time: 1:00:45 AM<br>
 * <br>
 *
 * Exception class for the Joker Programming Language to be used to handle
 * language exceptions with an embedded message
 */
public class JkrException extends RuntimeException
{
    /**
     * Constructor with a specified error message
     *
     * @param msg error message associated with this exception
     */
    public JkrException(String msg)
    {
        System.err.println("Error: " + msg);
    }
}
```

8.3.6 JkrInt.java

```
/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 1, 2003<br>
 * Time: 12:59:43 AM<br>
 * <br>
 *
 * Represents an 'int' type in the Joker programming language
 */
public class JkrInt extends JkrDataType
{
    int myInt;
```

```
/**
 * Constructor
 *
 * @param x integer value of this object
 */
public JkrInt(int x)
{
    myInt = x;
}

/**
 * Constructor calling parent 'JkrDataType' constructor and initializing integer value to 0
 */
public JkrInt()
{
    super();
    myInt = 0;
}

/**
 * Creates a new copy of this object
 *
 * @return clone of this object instance
 */
public JkrDataType copy()
{
    return new JkrInt(myInt);
}

/**
 * Gets the type name of this object
 *
 * @return object type
 */
public String typename()
{
    return "int";
}

/**
 * Gets the integer value of a JkrInt object
 *
 * @param j JkrInt object from which to extract integer value
 * @return integer value of the specified JkrInt
 */
public static int intValue(JkrDataType j)
{
    if(j instanceof JkrInt)
        return ((JkrInt)j).myInt;
    else
        j.error("invalid cast to int");

    return 0;
}

/**
 * Arithmetic addition of two JkrInt objects
 *
 * @param j JkrInt object to add to this instance
 * @return sum of the JkrInt object values
 */
public JkrDataType plus(JkrDataType j)
{
    if(j instanceof JkrInt)
        return new JkrInt(myInt + intValue(j));
    else
        j.error("invalid cast to int");

    return null;
}
```

```
/**
 * Arithmetic subtraction of two JkrInt objects
 *
 * @param j JkrInt object to subtract from this instance
 * @return difference of the JkrInt object values
 */
public JkrDataType minus(JkrDataType j)
{
    if(j instanceof JkrInt)
        return new JkrInt(myInt - intValue(j));
    else
        j.error("invalid cast to int");

    return null;
}

/**
 * Arithmetic multiplication of two JkrInt objects
 *
 * @param j JkrInt object to multiply with this instance
 * @return product of the JkrInt object values
 */
public JkrDataType mult(JkrDataType j)
{
    if(j instanceof JkrInt)
        return new JkrInt(myInt * intValue(j));
    else
        j.error("invalid cast to int");

    return null;
}

/**
 * Modular operation
 *
 * @param j value to use as mod value
 * @return value of this JkrInt mod j
 */
public JkrDataType mod(JkrDataType j)
{
  if(j instanceof JkrInt)
    return new JkrInt(myInt % intValue(j));
  else
    j.error("invalid cast to int");

  return null;
}

/**
 * Checks if two JkrInt objects have the same value
 *
 * @param j another JkrInt to compare to this instance
 * @return true if values are equal, false otherwise
 */
public JkrDataType eq(JkrDataType j)
{
    if(j instanceof JkrInt)
        return new JkrBoolean(myInt == intValue(j));
    else
        j.error("invalid cast to int");

    return null;
}

/**
 * Compares two JkrInt objects
 *
 * @param j another JkrInt to compare to this instance
 * @return 0 if equal, -1 if this instance is < j, 1 if this instance > j
 */
public int compareTo(JkrDataType j)
{
```

```
        if(j instanceof JkrInt)
        {
            JkrInt temp = (JkrInt) j;

            if(myInt > temp.myInt)
                return 1;
            else if(myInt < temp.myInt)
                return -1;
            else
                return 0;
        }
        else
            j.error("invalid cast to int");

        return -10;
    }

    /**
     * String representation of this JkrInt
     *
     * @return integer value
     */
    public String toString()
    {
        return "" + this.myInt;
    }
}
```

8.3.7 JkrInterpreter.java

```
import antlr.collections.AST;
import antlr.RecognitionException;

/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 1, 2003<br>
 * Time: 1:33:17 PM<br>
 * <br>
 *
 * Utility used in the Joker Programming Language to connect the Antlr generated tree
 * walker for the Joker Programming language to the backend java classes.  Contains
 * common methods called from the frontend tree walker to facilitate the tree walking
 * task and to pass necessary information to the backend classes for processing
 */
public class JkrInterpreter
{
    JkrSymbolTable symbolTable;
    boolean cont = false;
    boolean brk = false;
    JokerWalker walker = null;
    AST cond = null;
    AST itr = null;

    /**
     * Constructor to initialize this instance with the given Antlr tree walker for the Joker
     * Programming Language along with a root symbol table
     *
     * @param jw Antlr tree walker for the Joker Programming Language
     */
    public JkrInterpreter(JokerWalker jw)
    {
        symbolTable = new JkrSymbolTable(null, null);
        walker = jw;
    }

    /**
     * Gets the integer value of this string
     *
     * @param s String to convert to integer
```

```
 * @return integer value of the String
 */
public static JkrDataType getInteger(String s)
{
    return new JkrInt(Integer.parseInt(s));
}

/**
 * Gets the value of an identifier in the current scope
 *
 * @param name identifier
 * @return value of this identifier in this scope, null otherwise
 */
public JkrDataType getVariable(String name)
{
    JkrDataType j = symbolTable.getValue(name);

    if(j == null)
    {
        return new JkrDataType(name);
    }

    return j;
}

/**
 * Passes the name, value pair to the <code>getValue</code> method within
 * JkrSymbolTable
 *
 * @param name name of variable to be bound to the given 'value'
 * @param value value to be bound to the given variable 'name'
 */
public void setValue(String name, JkrDataType value)
{
    symbolTable.setValue(name, value);
}

/**
 * Assigns the value of 'k' to the name of 'j'
 *
 * @param j binds the value to the name of this JkrDataType
 * @param k binds this value to the name of 'j'
 * @return the new variable
 */
public JkrDataType assign(JkrDataType j, JkrDataType k)
{
    if(j.name != null)
    {
        JkrDataType temp = k.copy();
        temp.setName(j.name);
        symbolTable.setValue(temp.name, temp);
        return temp;
    }
    else
        return j.error(k, "=");
}

/**
 * Provides ability to assign and declare arrays and insert them into the symbol
 * table for a Joker program
 *
 * @param j specifies the datetype of this array will store
 * @param name name of this variable to bind this array value to
 * @param k the array containing all objects within it
 */
public void assignArray(JkrDataType j, String name, JkrDataType[] k)
{
    JkrArray temp = null;
    JkrDataType x = symbolTable.getValue(name, true);

    if(j instanceof JkrInt)
    {
```

```
    if(x == null) //name not yet bound in this scope
    {
        if(k != null) //bind name and value
        {
            JkrInt[] i = new JkrInt[k.length];

            for(int l = 0; l < i.length; l++)
            {
                i[l] = new JkrInt();
            }

            temp = new JkrArray(i);
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            symbolTable.declSetValue(name, temp);
        }
    }
    else //name already bound in this scope
    {
        System.err.println("Invalid integer assignment: " + name +
                           " already bound in this scope");
        System.exit(-1);
    }
}
else if(j instanceof JkrBoolean)
{
    if(x == null) //name not yet bound in this scope
    {
        if(k != null) //bind name and value
        {
            JkrBoolean[] b = new JkrBoolean[k.length];

            for(int i = 0; i < b.length; i++)
            {
                b[i] = new JkrBoolean();

            }

            temp = new JkrArray(b);
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            symbolTable.declSetValue(name, temp);
        }
    }
    else //name already bound in this scope
    {
        System.err.println("Invalid boolean assignment: " + name +
                           " already bound in this scope");
        System.exit(-1);
    }
}
else if(j instanceof JkrCard)
{
    if(x == null) //name not yet bound in this scope
    {
        JkrCard[] c = new JkrCard[k.length];

        for(int i = 0; i < c.length; i++)
        {
            c[i] = new JkrCard();
        }

        temp = new JkrArray(c);
        symbolTable.declSetValue(name, temp);
    }
    else
    {
        System.err.println("Invalid card assignment: " + name +
```

```
                                " already bound in this scope");
            System.exit(-1);
        }
    }
    else if(j instanceof JkrPack)
    {
        if(x == null) //name not yet bound in this scope
        {
            JkrPack[] p = new JkrPack[k.length];

            for(int i = 0; i < p.length; i++)
            {
                p[i] = new JkrPack();
            }

            temp = new JkrArray(p);
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            System.err.println("Invalid pack assignment: " + name +
                                " already bound in this scope");
            System.exit(-1);
        }
    }
    else //invalid assign
    {
        System.err.println("Invalid assignment");
        System.exit(-1);
    }

}

/**
 * Provides the ability to assign a variable 'name' to a Joker datatype 'value'
 *
 * @param j which Joker datatype this variable is
 * @param name name of this variable to bind this array value to
 * @param k the value to be bound to the given 'name'
 * @return the value bound to the given 'name'
 */
public JkrDataType assign(JkrDataType j, String name, JkrDataType k)
{
    JkrDataType temp = null;
    JkrDataType x = symbolTable.getValue(name, true);

    if(j instanceof JkrInt)
    {
        if(x == null) //name not yet bound in this scope
        {
            if(k != null) //bind name and value
            {
                temp = k.copy();
                symbolTable.declSetValue(name, temp);
            }
            else
            {
                symbolTable.declSetValue(name, new JkrInt());
            }
        }
        else //name already bound in this scope
        {
            System.err.println("Invalid integer assignment: " + name +
                                " already bound in this scope");
            System.exit(-1);
        }
    }
    else if(j instanceof JkrBoolean)
    {
        if(x == null) //name not yet bound in this scope
        {
            if(k != null) //bind name and value
```

```
        {
            temp = k.copy();
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            symbolTable.declSetValue(name, new JkrBoolean());
        }
    }
    else //name already bound in this scope
    {
        System.err.println("Invalid boolean assignment: " + name +
                           " already bound in this scope");
        System.exit(-1);
    }
}
else if(j instanceof JkrString)
{
    if(x == null) //name not yet bound in this scope
    {
        if(k != null) //bind name and value
        {
            temp = k.copy();
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            symbolTable.declSetValue(name, new JkrString());
        }
    }
    else //name already bound in this scope
    {
        System.err.println("Invalid string assignment: " + name +
                           " already bound in this scope");
        System.exit(-1);
    }
}
else if(j instanceof JkrCard)
{
    if(x == null) //name not yet bound in this scope
    {
        if(k != null)
        {
            temp = k.copy();
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            symbolTable.declSetValue(name, new JkrCard());
        }
    }
    else
    {
        System.err.println("Invalid card assignment: " + name +
                           " already bound in this scope");
        System.exit(-1);
    }
}
else if(j instanceof JkrPack)
{
    if(x == null) //name not yet bound in this scope
    {
        if(k != null)
        {
            temp = k.copy();
            symbolTable.declSetValue(name, temp);
        }
        else
        {
            symbolTable.declSetValue(name, new JkrPack());
        }
    }
```

```
        else
        {
            System.err.println("Invalid pack assignment: " + name +
                               " already bound in this scope");
            System.exit(-1);
        }
    }
    else //invalid assign
    {
        System.err.println("Invalid assignment");
        System.exit(-1);
    }

    return temp;
}

/**
 * Create new scope (used within for, foreach, while and if, else)
 */
public void newScope()
{
    symbolTable = new JkrSymbolTable(symbolTable, symbolTable);
}

/**
 * Close a scope
 */
public void closeScope()
{
    symbolTable = symbolTable.parent();
}

/**
 * Sets the AST representing the conditional expression to be evaluated upon each
 * iteration through a for loop
 *
 * @param c AST representing the conditional expression of a for loop
 */
public void setForCond(AST c)
{
    cond = c;
}

/**
 * Sets the AST representing the iteration expression to be evaluated upon a
 * successful execution of a for loop
 *
 * @param i AST representing the iteration expression
 */
public void setForItr(AST i)
{
    itr = i;
}

/**
 * Evaluates the iteration expression for this 'for' loop
 */
public void execForItr()
{
    JkrDataType a = null;

    if(itr != null)
    {
        try
        {
            walker.expr(itr);
        }
        catch(RecognitionException e)
        {
            System.err.println("Error:  Could not execute 'for' iteration expression");
            e.printStackTrace();
            System.exit(-1);
```

```
        }
    }
}

/**
 * Evaluates the conditional expression for this 'for' loop and returns its value
 *
 * @return true if conditional expression evaluates to true, false otherwise
 */
public boolean forCondTRUE()
{
    JkrDataType a = null;
    boolean b = false;

    if(cond == null)
      return true;

    try
    {
        a = walker.expr(cond);
        if(!(a instanceof JkrBoolean))
        {
            System.err.println("Error:  'for' condition not a boolean expression");
            System.exit(-1);
        }
        else
            b = ((JkrBoolean)a).bool;
    }
    catch(RecognitionException e)
    {
        System.err.println("Error:  Could not execute 'for' conditional expression");
        e.printStackTrace();
        System.exit(-1);
    }

    return b;
}

/**
 * Executes a 'foreach' statement within a Joker program
 *
 * @param itr_id name of the varialbe to use to iterate through the given 'array'
 * @param array the array of values to iterate through within this 'foreach' loop
 * @param foreach_stmts_tree the statements to execute upon each iteration of
 * this 'foreach' loop
 */
public void execForeach(JkrDataType itr_id, JkrArray array, AST foreach_stmts_tree)
{
    if(array.array instanceof JkrInt[])
    {
        this.assign(new JkrInt(), itr_id.name, null);
    }
    else if(array.array instanceof JkrBoolean[])
    {
        this.assign(new JkrBoolean(), itr_id.name, null);
    }
    else if(array.array instanceof JkrCard[])
    {
        this.assign(new JkrCard(), itr_id.name, new JkrCard());
    }
    else if(array.array instanceof JkrPack[])
    {
        this.assign(new JkrPack(), itr_id.name, new JkrPack());
    }
    else
    {
        System.err.println("Invalid 'foreach' array argument '" + array.name + "'");
        System.exit(-1);
    }

    for(int i = 0; !this.brk && i < array.length(); i++)
    {
```

```
            this.assign(itr_id, array.getElementAt(i));

            try
            {
               this.newScope();
                walker.expr(foreach_stmts_tree);
               this.closeScope();
            }
            catch(RecognitionException e)
            {
                System.err.println("Error:  invalid statement within 'foreach' loop");
                e.printStackTrace();
                System.exit(-1);
            }
        }
    }

    /**
     * Set the break condition within a loop
     *
     * @param b value to set this break boolean variable
     */
    public void setBreak(boolean b)
    {
        this.brk = b;
    }

    /**
     * Set the continue condition within a loop
     *
     * @param b value to set this continue boolean variable
     */
    public void setContinue(boolean b)
    {
        this.cont = b;
    }

    /**
     * Used to print the symbols in the symbol table structure of a Joker program
     */
    public void printSymbols()
    {
        System.out.println(symbolTable.toString());
    }
}
```

8.3.8 JkrOptions.java

```
import antlr.collections.AST;
import antlr.RecognitionException;
import java.util.Hashtable;
import java.util.Vector;
import java.util.Iterator;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 2, 2003<br>
 * Time: 12:22:57 AM<br>
 * <br>
 *
 * Utility class to process the 'option' block in the Joker Programming Language.
 * Stores the option name bound to AST representing the condition to evaluate to check if this
 * option is available to the user and also stores the AST representing the statements to execute
 * if the user were to choose this option given this option was available.
 *
 * This class allows for evaluating the option condition to determine if this option is available,
 * then outputing available options to the user.  Upon the user choosing an available option, the
 * statements associated with that option will be executed.  This can be done repetitively such is
 * within a card game environment where each player takes a turn and his only certain options
 * available due to the current game state.
 */
public class JkrOptions
{
    public static String DEFAULT = "DEFAULT";

    Hashtable optConditions = null;
    Hashtable optStatements = null;
    Vector available = null;
    JokerWalker walker = null;
    boolean optionsInitialized;

    /**
     * Constructor to initialize this object
     *
     * @param j Antlr tree walker generated for the Joker Programming Language to be used to
     * evaluate AST trees
     */
    public JkrOptions(JokerWalker j)
    {
        optConditions = new Hashtable();
        optStatements = new Hashtable();
        available = new Vector();
        walker = j;
        optionsInitialized = false;
    }

    /**
     * Adds this AST representing the option condition bound to the name of the option.
     * This option condition determines if this option is available or not after being evaluated
     *
     * @param name name of the option within a Joker program
     * @param condition AST tree to be evaluated to determine if this option is available
     */
    public void addOptionCondition(String name, AST condition)
    {
        optConditions.put(name, condition);
    }

    /**
     * Adds this AST representing the statements to be executed if the option with the given
     * 'name' is available and chosen by the user to be executed
     *
     * @param name name of the option within a Joker program
```

```
 * @param statements statements AST tree to be evaluated if this option were available
 * and chosen
 */
public void addOptionStatement(String name, AST statements)
{
    optStatements.put(name, statements);
}

/**
 * Adds an option with the given 'name' to the available options list
 *
 * @param name name of this option
 */
public void addAvailableOption(String name)
{
    available.addElement(name);
}

/**
 * Call when an option block is initialized within a Joker program
 */
public void init()
{
    clearOptions();
    optionsInitialized = true;
}

/**
 * Call when an option block has been completed within a Joker program
 */
public void end()
{
    clearOptions();
    optionsInitialized = false;
}

/**
 * Check if an options block has been initialized
 *
 * @return true if the Joker program is currently executing within an option block,
 * false otherwise
 */
public boolean isInitialized()
{
    return optionsInitialized;
}

/**
 * Called to clear the tables containing option block information that has completed execution
 * within a Joker program
 */
public void clearOptions()
{
    optConditions.clear();
    optStatements.clear();
    available.clear();
}

/**
 * Called after all option conditions and statements have been added and can begin processing
 */
public void execOptions()
{
    this.checkOptionConditions(); //compiles the available options list
    this.showAvailableOptions(); //shows the user the available options

    available.clear(); //clears teh available options for the next iteration if any exists
}

/**
 * Evaluates each option condition using the given Antlr tree walker for the Joker
 * Programming Language and if the condition evaluate to true, adds this option to the
```

```
 * list of available options
 */
public void checkOptionConditions()
{
    for(Iterator iterator = optConditions.keySet().iterator(); iterator.hasNext();)
    {
        String optName = (String) iterator.next();

        AST condTree = (AST) optConditions.get(optName);
        JkrDataType j = null;

        try
        {
            j = walker.expr(condTree);
        }
        catch(RecognitionException e)
        {
            System.err.println("Illegal option condition at option name '" + optName + "'");
            e.printStackTrace();
            System.exit(-1);
        }

        if(!(j instanceof JkrBoolean))
        {
            System.err.println("Illegal option condition for option name '" + optName + "'");
            System.exit(-1);
        }
        else if(((JkrBoolean) j).bool)
        {
            this.addAvailableOption(optName);
        }
        else
        {
        }
    }
}

/**
 * Outputs to the user the available options as determined by
 * <code>checkOptionConditions()</code> and prompts the user to select one of the available
 * options
 */
public void showAvailableOptions()
{
    AST optStmtsTree = null;

    while(true)
    {
        if(available.size() == 0)
        {
            optStmtsTree = (AST) optStatements.get(JkrOptions.DEFAULT);
            break;
        }


        System.out.println("AVAILABLE OPTIONS:");
        System.out.println("==================");

        for(Iterator iterator = available.iterator(); iterator.hasNext();)
        {
            String optName = (String) iterator.next();
            System.out.println("==> " + optName);
        }

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter option name:  ");
        String input = "";

        try
        {
            input = br.readLine();
        }
```

```
            catch(IOException e)
            {
                System.err.println("Error reading user input from console");
                e.printStackTrace();
            }

            boolean optionFound = false;

            for(Iterator iterator = available.iterator(); iterator.hasNext();)
            {
                String optName = (String) iterator.next();

                if(input.compareTo(optName) == 0)
                {
                    optionFound = true;
                    break;
                }
            }

            if(optionFound)
                optStmtsTree = (AST) optStatements.get(input);
            else
                optStmtsTree = null;

            if(optStmtsTree == null)
            {
                System.out.println("Your option '" + input + "' is NOT available");
                System.out.println("Please try again selecting an option name from the list
below\n");
                continue;
            }
            else
            {
                break;
            }
        }

        this.execSelectedOption(optStmtsTree);
    }

    /**
     * Evalutes the AST tree representing the statements to execute if a given option is chosen.
     * Evaluates using the Antlr tree walker for the Joker Programming Language
     *
     * @param statementsTree AST tree representing the statements to evaluate for this option
     */
    public void execSelectedOption(AST statementsTree)
    {
        try
        {
            walker.expr(statementsTree);
        }
        catch(RecognitionException e)
        {
            System.out.println("Error in option statements");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

8.3.9 JkrPack.java

```
import java.util.Vector;
import java.util.Hashtable;
import java.util.Iterator;

/**
 * Name:  Timothy SooHoo<br>
 * Date: Nov 23, 2003<br>
 * Time: 6:08:38 PM<br>
 * <br>
 *
 * Represents a 'pack' type in the Joker Programming Language.  A pack contains 0 or
 * more 'cards'.  This 'pack' type stores a card hierarchy giving each card a value
 * which allows cards to be compared in ways different from just their rank and suit.
 */
public class JkrPack extends JkrDataType
{
    public static final int DEFAULT_NUM_CARDS = 52;
    private static final int NUM_SHUFFLES = 500;

    private Vector pack;
    private Hashtable hierarchy = new Hashtable();

    /**
     * Default constructor which creates a standard 52-card deck includiing standard
     * 4 suits with 13 cards of each
     */
    public JkrPack()
    {
        this.pack = new Vector();
    }

    /**
     * Constructor which creates a custom pack
     *
     * @param pack custom created vector of cards
     */
    public JkrPack(Vector pack)
    {
        this.pack = pack;
    }

    /**
     * Constructor which creates a custom pack out of the cartesian product of the
     * numbers and suits
     *
     * @param numbers numbers a card in the pack can have
     * @param suits suits a card in the pack can have
     */
    public JkrPack(int[] numbers, char[] suits)
    {
        pack = new Vector(numbers.length * suits.length);
    }

    /**
     * Allows a single card to be wrapped into a pack for easier
     * pack opertions
     *
     * @param card card to be wrapped in this pack
     */
    public JkrPack(JkrCard card)
    {
       pack = new Vector();
       pack.addElement(card);
    }

    /**
     * Creates a new copy of this object
```

```
 *
 * @return clone of this object instance
 */
public JkrDataType copy()
{
    return new JkrPack(pack);
}

/**
 * Gets the type name of this object
 *
 * @return object type
 */
public String typename()
{
    return "pack";
}

/**
 * Initializes a standard 52 card pack of cards with 4 suits and 13 cards in each
 */
public void initDefaultPack()
{
    int[] numbers = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 };
    char[] suits = { 'D', 'H', 'S', 'C' };

    for(int i = 0; i < numbers.length; i++)
    {
        for(int j=0; j < suits.length; j++)
        {
            pack.addElement(new JkrCard(numbers[i], suits[j],
                    this.getHierarchyValue(JkrCard.getStrNumber(numbers[i]), "" +
                                                                        suits[j])));
        }
    }
}

/**
 * Initializes a pack of cards with the specified suits and card ranks.
 * Takes the cartesian product of the two vectors 'numbers' and 'suits'.
 * Hierarchy for this pack should be created before initializing the pack to ensure
 * that each card is given its proper 'value'
 *
 * @param numbers list of ranks to be in this pack
 * @param suits list of suits to be in this pack
 */
public void initPack(Vector numbers, Vector suits)
{
    for(int i = 0; i < numbers.size(); i++)
    {
        for(int j = 0; j < suits.size(); j++)
        {
            pack.addElement(new JkrCard(((Integer)numbers.elementAt(i)).intValue(),
                    ((Character)suits.elementAt(j)).charValue(),
                    this.getHierarchyValue("" + numbers.elementAt(i), "" +
                                                        suits.elementAt(j))));
        }
    }
}

/**
 * Removes the first element in the JkrPack and returns that element
 *
 * @return top element in the pack
 */
private JkrCard top()
{
    if(pack.size() < 1)
        return null;
    else
    {
        JkrCard top = (JkrCard) pack.firstElement();
```

```
        pack.removeElementAt(0);
        return top;
    }
}

/**
 * Shuffles the pack of cards
 */
public void shuffle()
{
    JkrCard temp;
    int randomCard1, randomCard2;

    for(int i = 0; i < JkrPack.NUM_SHUFFLES; i++)
    {
        randomCard1 = (int) (Math.random() * pack.size());
        randomCard2 = (int) (Math.random() * pack.size());

        temp = (JkrCard) pack.elementAt(randomCard1);
        pack.setElementAt(pack.elementAt(randomCard2), randomCard1);
        pack.setElementAt(temp, randomCard2);
    }
}

/**
 * Pop pack operator removes the 'num' of elements from the top of the pack and
 * returns the removed elements
 *
 * @param num number of elements to remove
 * @return JkrPack of removed elements
 */
public JkrPack pop(int num)
{
    Vector temp = new Vector(num);

    for(int i=0; i < num; i++)
    {
        temp.addElement(this.top());
    }

    return new JkrPack(temp);
}

/**
 * Removes given 'num' of elements from the back of this pack and returns the
 * removed elements
 *
 * @param num number of elements to remove from the back of this pack
 * @return the removed elements
 */
public JkrPack backPop(int num)
{
    Vector temp = new Vector(num);

    for(int i=0; i < num; i++)
    {
        temp.addElement(pack.lastElement());
        pack.removeElementAt(pack.size()-1);
    }

    return new JkrPack(temp);
}

/**
 * Push pack operator removes all elements from source pack and moves them as a whole
 * maintaining order placing them on the top of this pack
 *
 * @param source source pack to be added to this pack
 */
public void push(JkrPack source)
{
    Vector sourcePack = source.getPack();
```

```
    for(int i=0; i < sourcePack.size(); i++)
    {
        pack.insertElementAt(sourcePack.elementAt(i), i);
    }

    sourcePack.clear();
}

/**
 * Adds the source pack to the end of this pack and removes all elements in the
 * source pack
 *
 * @param source source pack to be added to this pack, all elements removed
 * @return this pack with the newly added elements
 */
public JkrPack enqueue(JkrPack source)
{
    Vector sourcePack = source.getPack();

    for(int i=0; i < sourcePack.size(); i++)
    {
        pack.add(sourcePack.elementAt(i));
    }

    sourcePack.clear();

    return this;
}

/**
 * Remove a card that has the same suit and number as specified
 *
 * @param suit
 * @param number
 * @return the card object removed, null if fail
 */
public JkrDataType remove(JkrSuit suit, int number)
{
    JkrCard ret = new JkrCard(number, suit);

    if(this.pack.size() > 0)
    {
        if(this.pack.remove(ret))
        {
            return ret;
        }
        else
        {
            return null;
        }
    }

    // by default
    return null;
}

/**
 * Pack index operator returns the nth element in the pack
 *
 * @param index into the pack vector
 * @return the card at the specified index, null otherwise
 */
public JkrDataType getElementAt(int index)
{
    if(index <= pack.size())
        return (JkrDataType) pack.elementAt(index);
    else
        return null;
}

/**
```

```
 * Accessor method to get cards in a pack
 *
 * @return vector of JkrCard objects
 */
public Vector getPack()
{
    return pack;
}

/**
 * Accessor method to get the cards in a pack
 *
 * @return array of card objects
 */
public JkrCard[] getArray()
{
    JkrCard[] j = new JkrCard[pack.size()];

    for(int i = 0; i < pack.size(); i++)
    {
         j[i] = (JkrCard) pack.elementAt(i);
    }

    return j;
}

/**
 * Accessor method to get the size of the pack ie the number of cards in this pack
 *
 * @return number of cards in this pack
 */
public int getSize()
{
    return pack.size();
}

/**
 * Accessor method to get the nth JkrCard element from the pack
 *
 * @param index index in the pack
 * @return JkrCard object at that index in the pack
 */
public JkrCard getCard(int index)
{
    return (JkrCard) pack.elementAt(index);
}

/**
 * Gets the nth card from the pack and returns a new instance of this card
 *
 * @param index the card at the specified index in the pack
 * @return a new instance of JkrCard representing this card at the given index
 * in the pack
 */
public JkrCard getNewCard(int index)
{
    if(index <= pack.size())
        return (JkrCard)(((JkrCard)(pack.elementAt(index))).copy());
    else
        return null;
}

/**
 * Adds a rule to this pack's hierarchy for a given card rank
 *
 * @param key card number
 * @param value the value a card of this rank used to calculate the total value
 * of this card
 */
public void addHierarchyCardRule(String key, String value)
{
    String k = key;
```

```
    int i = -1;

    try
    {
        Integer.parseInt(k);
    }
    catch(NumberFormatException e)
    {
        if(k.equals("J"))
            k = "11";
        else if(k.equals("Q"))
            k = "12";
        else if(k.equals("K"))
            k = "13";
        else if(k.equals("A"))
            k = "14";
        else if(k.equals("ten"))
          k = "10";
        else if(k.equals("nine"))
          k = "9";
        else if(k.equals("eight"))
          k = "8";
        else if(k.equals("seven"))
          k = "7";
        else if(k.equals("six"))
          k = "6";
        else if(k.equals("five"))
          k = "5";
        else if(k.equals("four"))
          k = "4";
        else if(k.equals("three"))
          k = "3";
        else if(k.equals("two"))
          k = "2";
        else
        {
            System.err.println("Error:  Invalid hierarchy card '" + key + "'");
            System.exit(-1);
        }
    }

    hierarchy.put(k, value);
}

/**
 * Adds a rule to this pack's hierarchy for a given card suit
 *
 * @param key card suit
 * @param value the value a card of this suit used to calculate the total value
 * of this card
 */
public void addHierarchySuitRule(String key, String value)
{
    if(key.equals("S") || key.equals("D") || key.equals("C") || key.equals("H"))
        hierarchy.put(key, value);
    else
    {
        System.err.println("Error:  Invalid hierarchy suit '" + key + "'");
        System.exit(-1);
    }
}


/**
 * Gets the value of a given card according to the specified hierarchy
 *
 * @param number number of this card 2-14
 * @param suit suit of this card
 * @return value of the number * value of suit according to the hierarcy for this pack
 */
public int getHierarchyValue(String number, String suit)
{
```

```
        int value1 = Integer.parseInt((String) this.hierarchy.get(number));
        int value2 = Integer.parseInt((String) this.hierarchy.get(suit));

        return value1*value2;
    }

    /**
     * Prints the hierarchy rules for this pack.
     */
    public void printHierarchy()
    {
        String str = "";
        String key = "";

        str = "PACK HIERARCHY\n";
        str +=  "==============\n";

        for(Iterator it = hierarchy.keySet().iterator(); it.hasNext();)
        {
            key = (String) it.next();
            str += "key: " + key + "\tvalue: " + ((String) hierarchy.get(key)) + "\n";
        }

        System.out.println(str);
    }

    /**
     * Prints all the cards in this pack
     */
    public void printPack()
    {
        String str = "";
        JkrCard card;

        if(pack.isEmpty())
            str += "Empty Pack";
        else
        {
            for(int i = 0; i < pack.size(); i++)
            {
                card = (JkrCard) pack.elementAt(i);
                str += "Card " + (i + 1) + ": " + card + "\tvalue: " +
                        card.getValue() + "\n";
            }
        }

        System.out.println(str);
    }

    /**
     * Get a String representation of all the cards in the pack
     *
     * @return all cards in pack
     */
    public String toString()
    {
        String str = "{ ";
        JkrCard card;

        for(int i = 0; i < pack.size(); i++)
        {
            card = (JkrCard) pack.elementAt(i);
            str += ((i != 0) ? ", " : "") + card.toString();
        }

        str += " }";
        return str;
    }
}

/**
 * Name:  Timothy SooHoo<br>
```

```
 * Date: Dec 1, 2003<br>
 * Time: 1:01:16 AM<br>
 * <br>
 *
 * Represents a 'string' type in the Joker Programming Language
 */
public class JkrString extends JkrDataType
{
    String str;

    /**
     * Construstor
     *
     * @param s value of this object
     */
    public JkrString(String s)
    {
        str = s;
    }

    /**
     * Constructor calling parent 'JkrDataType' constructor
     */
    public JkrString()
    {
        super();
    }

    /**
     * Creates a new copy of this object
     *
     * @return clone of this object instance
     */
    public JkrDataType copy()
    {
        return new JkrString(str);
    }

    /**
     * Gets the type name of this object
     *
     * @return object type
     */
    public String typename()
    {
        return "string";
    }

    /**
     * Access the string representation of this object
     *
     * @return string value of this object
     */
    public String toString()
    {
        return this.str;
    }

    /**
     * Compare two JkrString objects for equality
     *
     * @param j another JkrString object to compare to this instance
     * @return true if both objects contain the same character sequence, false otherwise
     */
    public boolean equals(JkrString j)
    {
        return str.equals(j.str);
    }
}
/**
 * Name:  Timothy SooHoo<br>
 * Date: Nov 23, 2003<br>
```

```
 * Time: 5:47:42 PM<br>
 * <br>
 *
 * Represents a 'suit' type within the Joker Programming Language.  Suits consist
 * of the four standard suits 'SPADES', 'CLUBS, 'HEARTS', and 'DIAMONDS'.
 */
public class JkrSuit extends JkrDataType
{
    public static final char HEART = 'H';
    public static final char DIAMOND = 'D';
    public static final char CLUB = 'C';
    public static final char SPADE = 'S';

    private char suit;

    /**
     * Constructor of JkrSuit class
     * Should use static class variables HEART, DIAMOND, CLUB, SPADE
     *
     * @param suit value should be 'H', 'D', 'C', or 'S' otherwise error thrown
     */
    public JkrSuit(char suit)
    {
        if(suit == JkrSuit.HEART || suit == JkrSuit.DIAMOND ||
                suit == JkrSuit.CLUB || suit == JkrSuit.SPADE)
            this.suit = suit;
        else
        {
            System.err.println("Error:  Invalid suit, must be 'H', 'D', 'C', or 'S'");
            System.exit(-1);
        }
    }

    /**
     * Get the suit associated with the string representation
     * Throws error if the string representation of the suit is invalid
     *
     * @param s string representation of a desired suit (ie hearts, diamonds, spades, clubs)
     * @return suit value as a character
     */
    public static char getSuitValue(String s)
    {
        if(s.equals("hearts"))
            return JkrSuit.HEART;
        else if(s.equals("diamonds"))
            return JkrSuit.DIAMOND;
        else if(s.equals("spades"))
            return JkrSuit.SPADE;
        else if(s.equals("clubs"))
            return JkrSuit.CLUB;
        else
        {
            System.err.println("Error:  Invalid suit type '" + s + "'");
            System.exit(-1);
            return '.';
        }
    }

    /**
     * Creates a new copy of this object
     *
     * @return clone of this object instance
     */
    public JkrDataType copy()
    {
        return new JkrSuit(suit);
    }

    /**
     * Gets the type name of this object
     *
     * @return object type
```

```
     */
    public String typename()
    {
        return "suit";
    }

    /**
     * Accessor method to get JkrSuit value
     *
     * @return suit value
     */
    public char getSuit()
    {
        return suit;
    }

    /**
     * Prints the full suit name
     *
     * @return full suit name or throws error if invalid
     */
    public String toString()
    {
        switch(this.suit)
        {
            case JkrSuit.HEART: return "HEARTS";
            case JkrSuit.DIAMOND: return "DIAMONDS";
            case JkrSuit.CLUB: return "CLUBS";
            case JkrSuit.SPADE: return "SPADES";
            default: System.err.println("Error:  Cannot print this suit, invalid suit type");
                     System.exit(-1);
                     return "";
        }
    }

    /**
     * Checks if two JkrSuit objects are equal
     *
     * @param obj another JkrSuit object to compare to this instance
     * @return true if both JkrSuit objects are equal, false otherwise
     */
    public boolean equals(Object obj)
    {
        JkrSuit s = (JkrSuit) obj;

        if(s.getSuit() == this.getSuit())
            return true;
        else
            return false;
    }
}
```

8.3.10 JkrSymbolTable.java

```
import java.util.HashMap;
import java.util.Iterator;

/**
 * Name:  Timothy SooHoo<br>
 * Date: Dec 1, 2003<br>
 * Time: 1:01:02 AM<br>
 * <br>
 *
 * Symbol table used in the Joker Programmign Language to store variable names and
 * their bindings to specific values in a given program.  Implements a hierarchical
 * parent, child relationship between symbol tables in different scopes similar
 * to java.  Symbol tables implemented using hashtables to store key, value pairs
 * binding variable names to their associated values in the table for easy look up
 * and modification.
 */
public class JkrSymbolTable extends HashMap
{
    JkrSymbolTable static_table, parent_table;

    /**
     * Constructor to create a symbol table with the given relationship to a parent
     * table and static table
     *
     * @param static_table the table that is located as a parent of all symbol
     * tables
     * @param parent_table the table located one scope above this symbol table
     */
    public JkrSymbolTable(JkrSymbolTable static_table, JkrSymbolTable parent_table)
    {
        this.static_table = static_table;
        this.parent_table = parent_table;
    }

    /**
     * Gets the parent symbol table of this symbol table unless parameter
     * 'isStatic' is set in which this will return the static symbol table
     *
     * @param isStatic set to retrieve static symbol table, otherwise return this
     * table's parent symbol table
     * @return parent symbol table
     */
    public JkrSymbolTable parent(boolean isStatic)
    {
        if(isStatic)
            return static_table;
        else
            return parent_table;
    }

    /**
     * Gets the parent symbol table of this symbol table
     *
     * @return parent symbol table
     */
    public JkrSymbolTable parent()
    {
        return parent_table;
    }

    /**
     * Gets the static symbol table
     *
     * @return static symbol table
     */
    public JkrSymbolTable getStatic_table()
    {
```

```
    return static_table;
}

/**
 * Check if this symbol table contains the given variable name
 *
 * @param name name of this variable to look for in the symbol table
 * @return true if a variable by the given 'name' exists in this symbol table,
 * false otherwise
 */
public boolean containsVariable(String name)
{
    return containsKey(name);
}

/**
 * Gets the value of the variable bound to this 'name'.  Searches this symbol
 * table first and proceeds to search the parent of the symbol table until
 * either a value for this 'name' is found, or there are no more symbol tables
 * that are above this symbol table.
 *
 * @param name name of the variable to look for in this symbol table, or in
 * parent symbol tables
 * @return value of the variable bound to this 'name', otherwise null if
 * variable name could not be found
 */
public JkrDataType getValue(String name)
{
    JkrSymbolTable table = this;
    Object temp = table.get(name);

    while(temp == null && table.parent() != null)
    {
        table = table.parent();
        temp = table.get(name);
    }

    //default, check the static table of global variables
    if(temp == null && this.parent() == null)
    {
        table = this.static_table;

        if(table != null)
            temp = table.get(name);
    }

    return (JkrDataType) temp;
}

/**
 * Gets the value of the variable bound to this 'name'.  Searches only
 * this symbol table if the 'onlyThisScope' parameter is set, otherwise it
 * calls <code>getValue(String name)</code> to search this symbol table and
 * its parents
 *
 * @param name name of the variable to look up a value for
 * @param onlyThisScope true specifies to search only this table,
 * otherwise seach all of its parents as well
 * @return value of the variable bound to this 'name', otherwise null
 * if variable name could not be found
 */
public JkrDataType getValue(String name, boolean onlyThisScope)
{
    if(onlyThisScope)
        return (JkrDataType) this.get(name);
    else
        return this.getValue(name);
}

/**
 * Sets the value of a variable with the given 'name' within this symbol
 * table.  Used in declaration of variables.
```

```
 *
 * @param name name to bind the 'value' to within the symbol table
 * @param value value to bind to this variable 'name' within this symbol table
 */
public void declSetValue(String name, JkrDataType value)
{
    value.name = name;
    this.put(name, value);
}

/**
 * Sets the value of a symbol with the given 'name' to the 'value'
 * within this symbol table.  First finds the scope (ie which symbol table)
 * this variable exists in searching this symbol table and its parents until
 * it finds a matching 'name'.  The variable is then given a new 'value'.
 *
 * @param name name of symbol to bind to this 'value'
 * @param value value to bind to this variable 'name' within this symbol table
 */
public void setValue(String name, JkrDataType value)
{
    JkrSymbolTable table = this.getScope(name);

    table.remove(name);
    value.name = name;
    table.put(name, value);
}

/**
 * Gets the symbol table a variable with the given 'name' exists in
 *
 * @param name name of variable to search for
 * @return the symbol table containing the variable by this name
 */
public JkrSymbolTable getScope(String name)
{
    JkrSymbolTable table = this;
    Object temp = table.get(name);

    while(temp == null && table.parent() != null)
    {
        table = table.parent();
        temp = table.get(name);
    }

    return table;
}

/**
 * Prints out the values stored in this symbol table and recursively
 * prints its parent symbol table
 *
 * @return string representation of the entire symbol table structure
 */
public String toString()
{
    String str = "========================\n";
    str += "SYMBOL TABLE\n";
    str += "========================\n";

    JkrSymbolTable table = this;
    String key = "";

    while(table != null)
    {
        str += "# SYMBOLS: " + table.keySet().size() + "\n";
        for(Iterator it = table.keySet().iterator(); it.hasNext();)
        {
            key = (String) it.next();
            str += "key: " + key + "\tvalue: " + table.getValue(key) + "\n";
        }
        table = table.parent();
```

```
            str += "========================\n";
        }
        return str;
    }
}
```

8.5 Main Program – JokerMain.java

```
/**
 * Main program that takes in Joker Source files
*
 * @author Jeffrey Eng and Jonathan Tse
 */


import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

class JokerMain
{
    static boolean verbose = false;

    // default constructor
    public JokerMain()
    {
    }

    public static void execFile( String filename )
    {
        InputStream input = null;


        try
        {
            if(!(new File(filename)).isFile())
            {
                System.out.println("File '" + filename + "' does not exist");
                System.exit(-1);
            }
            else
                input = (InputStream) new FileInputStream(filename);

            JokerLexer lexer = new JokerLexer( input );
            JokerParser parser = new JokerParser( lexer );

            // Parse the input program
            parser.game();


            CommonAST tree = (CommonAST)parser.getAST();

            if ( verbose )
            {
                // Print the resulting tree out in LISP notation
                System.out.println(
                    "================ tree structure ======================" );
                System.out.println( tree.toStringList() );
                System.out.println(
                    "================ end tree structure = ================\n" );
            }

            JokerWalker jwalker = new JokerWalker();
            jwalker.expr(tree);

        }
        catch( IOException e )
        {
```

```
            System.err.println( "Error: I/O: " + e );
            System.exit(-1);
        }
        catch( RecognitionException e )
        {
            System.err.println( "Error: Recognition: " + e );
            System.exit(-1);
        }
        catch( TokenStreamException e )
        {
            System.err.println( "Error: Token stream: " + e );
            System.exit(-1);
        }
        catch( Exception e )
        {
            System.err.println( "Error: " + e );
            System.exit(-1);
        }
    }

    public static void main( String[] args )
    {
        verbose = args.length >= 1 && args[0].equals( "-v" );

        if((verbose && args.length != 2) || (!verbose && args.length !=1))
        {
            System.out.println("Must specify a '.jkr' file");
            System.exit(-1);
        }
        else if(verbose)
        {
            int end = args[1].length();
            int start = end - 4;


            String ext = args[1].substring(start, end);

            if(!ext.equals(".jkr"))
            {
                System.out.println("Invalid program file '" + args[1] + "'");
                System.out.println("Must specify a '.jkr' file");
                System.exit(-1);
            }
            else
                execFile(args[1]);
        }
        else if(!verbose)
        {
            int end = args[0].length();
            int start = end - 4;


            String ext = args[0].substring(start, end);

            if(!ext.equals(".jkr"))
            {
                System.out.println("Invalid program file '" + args[0] + "'");
                System.out.println("Must specify a '.jkr' file");
                System.exit(-1);
            }
            else
                execFile(args[0]);
        }

        System.exit( 0 );
    }
}
```

8.6 Sample Program – blackjack.jkr

```
/**
 * Implements a standard 21 blackjack game with 4 players.
 * There is no dealer and the winner(s) is/are the player
 * to achieve a score closest to 21 without exceeding 21.
 *
 * @author Timothy SooHoo
 */
game BlackJack
{
    init
    {
        int numPlayers = 4; //number of players in the game

        //value of cards used to calculate hand value within main
        hierarchy : { A(11), K(10).Q.J.ten, nine(9), eight(8),
                  seven(7), six(6), five(5), four(4), three(3), two(2) }
        by { spades(1).hearts.clubs.diamonds } into DECK;

        DECK@; //shuffle the deck

        pack[numPlayers] playerHands; //create the array of player hands
    }

    main
    {
        int goalScore = 21; //goal score
        int counter;

        print "Dealing cards...";

        for (int n = 1; n <= 2; n++) //dealing loop
        {
            //deal one card to each player's hand
            foreach playerHands as hand
            {
                hand += (DECK >> 1);
            }
        }

        counter = 1;

        //print player hands
        foreach playerHands as hand
        {
            print "PLAYER " + counter + ":";

            for(int i=0; i < hand.size; i++)
            {
                int x = i+1;

                print "Card " + x + ": " + hand[i];
            }

            print "";
            counter++;
        }

        counter = 1;
        boolean validScore = false;
        int[numPlayers] handSums; //store each player's hand value

        //loop to execute main game functionality
        foreach playerHands as hand
        {
            print "";
            print "Player " + counter + "'s turn:";
```

```
print "Your hand: " + hand;

while(true)
{
    int sum = 0;

    //calculate sum of card values
    for(int y = 0; y < hand.size; y++)
    {
        sum = sum + hand[y].value;
    }

   //check for aces if the hand value is > 21
   if(sum > goalScore)
   {
       int numAces = 0;

       //count up number of aces before recalculating score
       for(int y = 0; y < hand.size; y++)
       {
           if(hand[y].rank == "A")
           {
               numAces++;
           }
       }

       int temp = 1;
       int tmpNumAces;

       //recalculate score by setting one ace's
       //value to 1, then proceed to set 2 ace's value to
       //1 if they exist until score is less than 21
       while(numAces > 0)
       {
           sum = 0;
           tmpNumAces = numAces;

           for(int y = 0; y < hand.size; y++)
           {
               if(hand[y].rank == "A" && tmpNumAces != 0)
               {
                   sum = sum + 1;
                   tmpNumAces--;
               }
               else
               {
                   sum = sum + hand[y].value;
               }
           }

           //score is below 21 continue
           if(sum < goalScore)
           {
               break;
           }
           else
           {
               numAces--; //score above 21 continue
           }
       }
   }

   handSums[counter-1] = sum; //store hand value

   print "";
   print "Hand value is " + sum;
   print "";

   //prompt user for input based upon their displayed hand
    option(hand)
   {
       //give the player another card to be
```

```
                //added to the player's hand value
                hit (handSums[counter-1] <= goalScore)
                {
                    hand += (DECK >> 1);
                   print "";
                    print "Current hand: " + hand; //print new hand
                }
                //proceed to the next player
                stand (handSums[counter-1] <= goalScore)
                {
                    print "STAND";
                   break;
                }
                //the player's hand value > 21, they lose by default
                default
                {
                    print "You BUST!!!";
                    break;
                }
            }
        }
        counter++;
    }

    print "";
    print "FINAL HANDS";

    counter = 1;

    //print final hands
    foreach playerHands as hand
    {
        print "Player " + counter + " : " + hand;
        counter++;
    }

    int max = 0;

    //find the maximum score < 21 achieved in this game
    for(int i=0; i < handSums.size; i++)
    {
        if(handSums[i] > max && handSums[i] <= goalScore)
        {
           max = handSums[i];
        }
    }

    print "";

    if(max == 0) //if everyone busted
    {
        print "All players busted";
        print "No winners";
    }
    else
    {
        //find all players with a hand value equal to the max
        for(int i=0; i < playerHands.size; i++)
        {
           int x = i+1;

           //declare the player a winner
           if(handSums[i] == max)
           {
               print "Player " + x + " WINS with score " + max + "!!!";
           }
        }
    }

    print "";
  }
}
```

8.7 Sample Program – war.jkr

```
/**
 * The game of War for 2 players implemented in the Joker
 * Programming Language.
 *
 * @author Timothy SooHoo
 */
game War
{
    init
    {
       pack theDeck;

       hierarchy :  { A(14), K(13), Q(12), J(11), ten(10), nine(9),
              eight(8), seven(7), six(6), five(5), four(4), three(3), two(2) } by
              { spades(1) . hearts . clubs . diamonds } into theDeck;

       // shuffle the deck
       theDeck@;

       pack player1Hand; //player 1's hand
       pack player2Hand; //player 2's hand

       int deckSize = theDeck.size; //store the deck size

       print "Dealing cards...";

       //deal the deck evenly between both players alternating
       for(int i=0; i < deckSize; i++)
       {
           if( i % 2 == 1)
              player1Hand += (theDeck >> 1);
           else
               player2Hand += (theDeck >> 1);
       }
    }

    main
    {
       int winner; //stores winner

       //game continues until winner declared
       for(int turn=0; true; turn++)
       {
           boolean isDone = false; //condition of while loop
           boolean endGame = false; //breaks out of the game for loop
           card oneCard, twoCard;
           pack prize;

           //print game status with number of cards for each player
           print "";
           print "DECK SIZE:";
           print "Player 1: " + player1Hand.size;
           print "Player 2: " + player2Hand.size;
           print "";

           while(isDone != true)
           {
              // IF they run out out of cards, other player is winner
              if(player1Hand.size == 0)
              {
                  winner = 2;
                  endGame = true;
                  break;
              }
              else if (player2Hand.size == 0)
              {
                  winner = 1;
                  endGame = true;
```

```
    break;
}

print "Player 1 turn:";
//prompt player1 to flip his card
option(player1Hand)
{
    flip(true)
    {
        oneCard = (player1Hand >> 1)[0]; //player1 show top card of hand
    }
    default
    {
    }
}

print "";

print "Player 2 turn:";
option(player2Hand)
{
    flip(true)
    {
        twoCard = (player2Hand >> 1)[0]; //player2 show top card of hand
    }
    default
    {
    }
 }

print "";
print "Player 1's card: " + oneCard;
print "Player 2's card: " + twoCard;
print "";

//player1 wins this turn, and gets both cards added to this hand
if(oneCard.value > twoCard.value)
{
    print "Player 1 wins round";

    player1Hand += oneCard;
    player1Hand += twoCard;
    player1Hand += prize;

    isDone = true;
}
//player2 wins this turn, and gets both cards added to this hand
else if(oneCard.value < twoCard.value)
{
    print "Player 2 wins round";

    player2Hand += oneCard;
    player2Hand += twoCard;
    player2Hand += prize;

    isDone = true;
}
else
{
   print "***************";
   print "**DECLARE WAR**";
   print "***************";

   // IF they turn out of cards, other player is winner
   if( player1Hand.size < 4 )
   {
       winner = 2;
       endGame = true;
       break;
   }
   else if ( player2Hand.size < 4 )
   {
```

```
                    winner = 1;
                    endGame = true;
                    break;
                }

                // I declare war
                prize += oneCard;
                prize += twoCard;
                prize += (player1Hand >> 3);
                prize += (player2Hand >> 3);
            }
        }

        //check if game over
        if(endGame == true)
            break;
    }

    //output winner
    print "The winner is:  Player " + winner + "!!!";
  }

}
```

8.8 Tester Program – Tester.java

```
/**
 * Tester program running test suite - concept based on CILY's test program (thanks!).
 *
 * @author Jonathan Tse
 */

import java.io.*;

public class Tester
{
    public static void main(String args[])
    {
        JokerMain jm = new JokerMain();

        // other variables
        boolean seenError = false;
        boolean atLeastOneFile = false;
        String expectedText = "";
        boolean genExp = false;
        boolean test = false;

        // file variables
        int upperLimit = 4000;
        String dirname = "test/cases/";
        String filename = "";
        PrintStream ps = null;

        if (args.length >= 1)
            {
                // genExp would automatically generate exp files
                genExp = args[0].equals( "-g" );

                // test would actually test them
                test = args[0].equals( "-t" );

                if (args.length == 2)
                    dirname = args[1];
            }

        // going through test cases
        for (int i = 1001; i < upperLimit; i++) {

            filename = dirname + i + ".jkr";
```

```
if ((new File(filename).isFile()) && test)
   {
       // testing the existing jkr files against the exps (only if exp exists)
       if (new File(dirname + i + ".exp").isFile())
          {
              atLeastOneFile = true;
              try
                 {
                     ps = new PrintStream(new FileOutputStream(dirname + i + ".out"));

                     // saving old stream and setting file output stream
                     PrintStream oldOut = System.out;
                     System.setOut(ps);
                     System.setErr(ps);

                     // running language and setting stream back
                     jm.execFile(filename);
                     System.setOut(oldOut);
                     System.setErr(oldOut);

                     ps.close();

                     expectedText = getFileContents(dirname + i + ".exp");

                     if (!expectedText.equals(getFileContents(dirname + i + ".out")))
                        {
                            System.out.println("[" + i + " FAIL] ");
                            seenError = true;
                        }
                     else
                        {
                            System.out.print("[" + i + " PASS] ");
                        }
                     System.out.println((new BufferedReader(new
                     FileReader(filename))).readLine().substring(3));
                 }
              catch (IOException e)
                 {
                     System.out.println("Cannot write to file " + i + ".out");
                 }
          }
   }
else if ((new File(filename).isFile()) && genExp)
   {
       // a call to generate the exp files for jkr files that don't have exp yet
       if (!(new File(dirname + i + ".exp").isFile()))
          {
              atLeastOneFile = true;
              // only if exp doesn't exist yet
              try
                 {
                     ps = new PrintStream(new FileOutputStream(dirname + i + ".exp"));

                     // saving old stream and setting file output stream
                     PrintStream oldOut = System.out;
                     System.setOut(ps);
                     System.setErr(ps);

                     // running language and setting stream back
                     jm.execFile(filename);
                     System.setOut(oldOut);
                     System.setErr(oldOut);

                     ps.close();

                     System.out.println(i + ".exp generated.");
                 }
              catch (IOException e)
                 {
                     System.out.println("Cannot write to file " + i + ".exp");
                 }
              catch (Exception e)
```

```
                            {
                                System.out.println("Cannot write to file " + i + ".exp");
                            }
                        }
                }
        }

        if (test)
            {
                if (seenError) {
                    System.out.println("Test cases have errors!");
                }
                else {
                    if (atLeastOneFile)
                        System.out.println("Test cases passed!");
                    else
                        System.out.println("No test cases found.");
                }
            }

        if (genExp)
            {
                if (atLeastOneFile)
                    System.out.println("Test expected files generated.");
                else
                    System.out.println("No new files to generate.");
            }
    }

    // retrieves contents of file
    public static String getFileContents(String fileName)
    {
        String line, retVal = "";
        BufferedReader in;

        try
            {
                in = new BufferedReader(new FileReader(fileName));
                while ((line = in.readLine()) != null)
                    retVal += line;
        }
        catch (IOException e)
            {
                System.err.println("cannot open output file " + fileName + " for test case
                comparisons");
            }

        return retVal;
    }
}
```

Test Cases

1001.jkr

```
//  Empty program. One line. game Test {  init {  }  main {}   }
```

1002.jkr

```
// Empty program. Multi-line

game Test {
  init {}
  main {}
}
```

1003.jkr

[unable to add due to time constraints and Office limitations]