Joker

a Card Game Programming Language
Language Reference Manual

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 October 27 – Document Created

Index

1 Lexical Conventions

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.

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.

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.

1.4 Keywords

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

```
int                 continue
boolean             init
card                main
pack                game
string              option
if                  case
else                default
for                 true
while               false
break
```

2 *Meaning of Identifiers*

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

2.1 Basic Types

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

Variables of the *integer* type are signed integers. They have a maximum value of 2147483647 and a minimum value of -2147483648.

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

Variables of the *card* type represent playing cards. See section 2.1.1.

Variables of the *pack* type represent groups of playing cards. See section 2.1.2.

2.1.1 Card type

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

The **number** is the number value of a card. On a playing card, this is the corner numeral. In the case of face cards,

```
foo.number = 10;
```

The **suit** is the suit of a card.

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

The **viewability** is a boolean attribute of whether or not a card is viewable to everyone. It can be accessed by:

```
foo.private = true;
```

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 (see section 5.3). Like the *card* type, *pack* objects also have a number of attributes, which consist of: **size**.

The **size** of a pack is the number of elements in the pack. The “bottom” or last element of the pack can be found at **size**-1.

```
if ( foo.size == 5)
```

2.2 Derived Types

In addition to the basic types, programmers may specify types derived from these basic types. Users have the ability to make an array of *int* and *boolean* values. For declaration of the array derived type, please check section 5.1. For example, declaring an array of five integers is:

int[5] foo;

2.3 Configuration Type

There is a special object from the special *configuration* type. This object allows the passing of parameter constants into the game logic. It is the equivalent of C’s and Java’s argv parameter, whereas in C and Java, string values of argv are obtained from the command line execution, *configuration* parameters are abstractly obtained through the Joker special runtime.

To access *configuration* parameters, programmers may access attributes through the special singleton variable `config`. For example:

```
int[config.NUM_PLAYERS] foo;
```

where the Joker file may have been executed using:

```
$ joker Blackjack.jkr –config NUM_PLAYERS=4
```

The actual passing of configuration values depends on the runtime implementation. Possibilities include command line name-value pairs (as shown above), GUI dialog boxes, or even a parsed configuration file.

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^{TM}, the order of evaluation of expressions is left-to-right.

3.1 Primary Expressions

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

> *primary-expression:* *identifier*
> | *string*
> | *(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.

3.2 Arithmetic Expressions

Arithmetic expressions take primary expressions as operands.

3.2.1 Postfix expressions

> *postfix-expression:*
> *primary-expression*
> | *postfix-expression++*
> | *postfix-expression--*

Unary arithmetic operators include “++” and “--”. These operators are post-fix. The operand is either incremented (++) or decremented (--) and only applies to *int*. The value of the expression is the value of the operand.

3.2.2 Multiplicative operators

> *multiplicative-expression:* *multiplicative-expression * postfix-expression*
> | *multiplicative-expression / postfix-expression*
> | *multiplicative-expression % postfix-expression*
> | *primary-expression*

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

Please note that truncation of the result may occur when performing division.

3.2.3 Additive operators

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

The additive operators “+”, “-” indicate addition and subtraction, respectively and are grouped from left to right. They are only applicable to *int* .

3.3 Pack Operators

Pack expressions are grouped from right-to-left.

pack-expression :

pack-expression >> integer-expression
pack-expression << pack-expression
pack-expression += pack-expression
pack-expression -= integer-expression
pack-expression @

3.3.1 Stack-style operators

The *pack* type supports a number of stack operations, including pop, push, enqueue, backpop (which is popping from the rear).

3.3.1.1 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.

3.3.1.2 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.

3.3.1.3 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 (+=).

3.3.1.4 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 (+=).

3.3.2 Other pack expressions

3.3.2.1 Shuffle operators

pack-expression @

The shuffle operator is a unary operator that randomly reorders the elements of a pack. This value of this expression is a reference to the reordered pack.

For example, the expression

```
deck @
```

reorders deck randomly and returns deck itself.

3.3.2.2 Pack index operators

pack-expression < integer-expression >

Pack index operators allow the extraction of the nth element in a pack. The value returned by this expression is a reference to the *card* element at the nth slot.

3.4 Relational Expressions

relational-expression : *integer-relational-expression*
| *card-relational-expression*

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.

3.4.1 Integer Relational Expressions

integer-relational-expression: integer-expression < integer-expression
| integer-expression > integer-expression
| integer-expression <= integer-expression
| integer-expression >= integer-expression

Integer relational expressions return 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)
```

3.4.2 Card Relational Expressions

card-relational-expression : card-expression < card-expression
| card-expression > card-expression
| card-expression <= card-expression
| card-expression >= card-expression

Card relational expressions return a boolean true if the specified relation is evaluated to true. Otherwise, they return a boolean false. A card is greater than (>) another when the product of its number-value and suit-value is greater than the other. The same applies for the other three binary card relational operators (< less than, <= less than or equal to, and >= greater than or equal to). For example,

```
card1 < card2
```

returns true if the value of card1 as defined in the card hierarchy is higher than the value of card2.

The hierarchy of the cards is defined in the init section of each program (see Section bla).

3.5 Equality Expressions

These equality expressions are binary and require two operands. They include equal to (“==”) and not equal to (“!=”). For example,

integer-expression == integer-expression
integer-expression != integer-expression

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

3.6 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 && conditional-expression
conditional-expression || conditional-expression

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

3.7 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.

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.

lvalue += expression

This assignment will replace the value of the lvalue by the result of the addition operator applied to lvalue and expression. Both the lvalue and the expression are of the same type.

lvalue -= expression

This assignment will replace the value of the lvalue by the result of the subtraction operator applied to lvalue and expression. Both the lvalue and the expression are of the same type.

*lvalue *= expression*

This assignment will replace the value of the lvalue by the result of the multiplication operator applied to lvalue and expression. Both the lvalue and the expression are of the same type.

lvalue /= expression

This assignment will replace the value of the lvalue by the result of the division operator applied to lvalue and expression. The decimal portion of quotient will be truncated. (Refer to Section 3.2.2) Both the lvalue and the expression are of the same type.

4 Statements

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.

4.2 Conditional Statements

Begins with the `if` keyword and followed by an optional `else`. The `if` is followed by an *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. For example,

`if` (*expression*)
    statement

or

`if` (*expression*)
    statement
`else`
    statement

4.3 Iterative Statements

iterative-statement:

`while` (*expression*) *statement*
`for` (*expression$_{opt}$* ; *expression$_{opt}$* ; *expression$_{opt}$*) *statement*
`foreach` *(array-expression | pack-expression)* `as` *identifier statement*

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

4.3.1 for statement

This statement has the form:

for (*expression$_{opt}$*; *expression$_{opt}$*; *expression$_{opt}$) 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.

4.3.2 foreach statement

`foreach` (*array-expression* | *pack-expression*) 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*.

4.3.3 while statement

`while` (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.

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;
```

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;
```

4.6 Option statement

option-statement: `option` { *option-statement-list* }

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

option-statement-item: `case` *(identifier)*? : (*conditional-expression*) *statements*
`default` : *statements*

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.

4.7 Game statement

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

```
game identifier {
    init statements
    main statements
}
```

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.

5 Declarations

declaration: type-specifier identifier-list
type-specifier: `int`
`boolean`
`pack`
`card`
identifier-list: *identifier-init*
identifier-list, identifier-init
identifier-init: *identifier*
identifier = (*integer-expression* | *boolean-expression*
| pack-*expression* | *card-expression*)

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.

5.1 Array Declarations for *int* and *boolean*

array-declaration: *type-specifier array-identifier;*

array-identifier: *bracket-list identifier*
| *empty-bracket-list identifier*
| *empty-bracket-list identifier init-array-identifier*

bracket-list: [*integer-expression*]
| *bracket-list* [*integer-expression*]

empty-bracket-list: []
| *empty-bracket-list* [*integer-expression*]

init-array-identifier: = { *integer-expression-list* }
| = { *boolean-expression-list* }

integer-expression-list: *integer-expression*
| *integer-expression-list, integer-expression*

boolean-expression-list: *boolean-expression*
| *boolean-expression-list, boolean-expression*

5.2 Hierarchy Declaration

hierarchy-declaration: `hierarchy`: *number* by *suit*

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

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

element-rule: name (*value*) (. *name*)*

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.

A card will be created by taking an element-rule from number and suit. Thus, the pack will be a Cartesian product of all numbers 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 number and suit are equal.

For example,

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

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.

Code Sample- Blackjack

The following code sample describes a single game of Blackjack without dealer rules.

```
game Blackjack {
    init {
        pack DECK;
        pack [ config.NUM_PLAYERS ].playerHands;
    }

    main {

        // the deal
        // (one card up, one card down to each player)
        boolean flipMe = true;

        for (int n = 1; n < = 2; n++) {
            foreach playerHands as hand{
                //one care to each hand
                hand += (DECK >> 1);
                hand<bottom>.private = filpMe;
            }

            flipMe = false;
        }

        // the action
        foreach playerHands as hand {
            while(true) {
                int sum = 0;

                // find the player total
                foreach hand as card {
                    sum += card.value;
                }
```

```
                        option:
                                hit (if sum <= 21) {
                                        hand += (DECK >> 1);
                                        hand <bottom>.private = false;
                                }

                                stand (if sum <= 21) {
                                        break;
                                }

                                bust (true) {
                                        lost( hand );
                                        break;
                                }
                }
        }

        // at this point every player has gone.
        Int maxPlayer = 0;

        foreach playerHands as hand {
                if (hand.sum > playerHands[maxPlayer].sum) {
                        maxPlayer = i;
                }
        }
    }
}
```