1. Data Types

All information in CGL can be represented as one of seven fundamental data types. A card game in CGL can be represented entirely by initializing variables of these data types and by manipulating their values.

The seven data types are integers, doubles, booleans, strings, cards, lists, and players.

1.1 Integer

An integer is a 32-bit signed integer. All integers are represented in decimal; CGL does not provide support for octal or hexadecimal representations of integers. An integer is declared with the `int` keyword.

Examples of valid integers:

```
1
65
0
-149
```

1.2 Double

A double is a 64-bit signed floating point number. As with integers, all doubles are represented in decimal. Doubles must always contain exactly one decimal point, and they must always begin with an integer (i.e. you must write `0.5` instead of `.5`). A double is declared with the `double` keyword.

Examples of valid doubles:

```
0.01
0.0
12.
-32.1
```

Examples of invalid doubles:

```
.01
9.3.4
```

1.3 Boolean:

A boolean is a data type with two possible values: `true` or `false`. A boolean is a distinct type, like in Java, and is not comparable with integer values 0 or 1, as in C. A boolean is declared with the `bool` keyword.
1.4 String

A string is a sequence of characters enclosed in double quotation marks (" "). Strings in CGL also support for the following escape characters:

\n /* new line */
\t /* tab */
\" /* double quote */
\\ /* backslash */

In CGL strings, all backslash characters must be followed by a n, t, \, or /. A string is declared with the string keyword.

Examples of valid strings:

"string"
"123STRING123"
"str$_ing^&"
"string\n"
"\""
"\\\t\n\""
"

Examples of invalid strings:

"string"
"string\"
"string\o"
"\\\""
"""

Note: CGL does not utilize single quotes, and does not use single quotes to identify a character as a distinct data type.

1.5 Card

A card is a data type that represents a specific card in the standard 52-card deck. Each card is declared in three parts: an identifying $ sign, a card value (2, 3, 4, 5, 6, 7, 8, 9, J, Q, K, or A), and a suit (C, D, H, or S). The values of the cards within CGL range from 2-14. For games that use different card values, the programmer may use conditionals to simulate the different rankings. (See Appendix A for an example.)
An exhaustive list of valid cards is shown below:


There are some other, more flexible ways to represent cards in CGL. If you do not want to specify a card value or suit, you can replace that attribute with an asterisk (*). An exhaustive list of cards with asterisks is shown below:

Suit-less valued cards:

$2*, \,$$3*, \,$$4*, \,$$5*, \,$$6*, \,$$7*, \,$$8*, \,$$9*, \,$$10*, \,$$J*, \,$$Q*, \,$$K*, \,$$A*

Value-less suited cards:

$*C, \,$$*D, \,$$*H, \,$$*S

Any card:

$**

You may also declare the values of CGL cards using integer variables. For example, if you have previously defined the identifier a to equal 5, then you may represent a five of diamonds in the following way: $(a)D. You could also represent a six of diamonds like this: $(a+1)D. The parentheses are crucial when representing card values with a variable, as demonstrated in the following example:

```java
int J = 8;
card a = $JS; /* jack of spades */
card b = $(J)S; /* 8 of spades */
```

In card declarations employing a variable, the variable must be of type Integer and have a value in the range of 2-14. Otherwise, an error will be thrown.

Within a card declaration, the characters J, Q, K, and A correspond to the values 11, 12, 13, and 14. Therefore, the following card variable declaration examples are identical:

```java
int i = 12;
card queenOfHearts1 = $(i)H;
card queenOfHearts2 = $12H;
card queenOfHearts3 = $QH;
```

A card is declared using the card keyword.
1.6 List

A list is a data type representing an ordered collection of the seven fundamental data types: i.e., integers, doubles, booleans, strings, cards, players, and lists themselves. CGL does not constrain the size of a list, nor does CGL limit the number of data types a single list can hold. A list begins with [ and ends with ], and each element within the list is separated by commas. A list is declared with the list keyword.

Examples of valid lists:

[]
[1]
[1, true]
[“string”, 3.]
[“foo”, 12, [$AH, $KH, $QH, $JH, $10H]]

Examples of invalid lists:

[1; 9]
”string” [

1.7 Player

The player data type is the most complex data type in CGL. A player represents a collection of other data types, referred to as its “subtypes”. These subtypes are declared at the very beginning of a program, in a block of code labeled PLAYER { } (see section 4.1 for more information on CGL program layout). Each player in a CGL game has exactly the same set of subtypes, although those subtypes may hold different values for different players. Furthermore, the number and names of the player subtypes cannot be changed after they are initially declared. Here is an example subtype declaration:

PLAYER
{
    list hand = [];
    int score = 0;
}

This piece of code states that every player data type now has two subtypes: a list called “hand” and an integer called “score”. These values of these different subtypes can be accessed and modified with the dot (.) operator. For example:

player1.score = 5; /* sets player1’s score to 5 */
player1.hand <+ <- deck; /* takes the top card of the deck and adds it to the back of player1’s hand */
player2.score = 7;               // sets player2’s score to 7; has no affect on player1’s score */
player2.hand = player1.hand;      // sets player2’s hand to be a copy of the list found in player1’s hand */

In addition to the subtypes declared in the **PLAYER** block, each player in CGL starts the game with two default subtypes: a string **name** and an integer **turnID**. (More on turn IDs can be found in section 4.1.3.) In order to declare a player for the first time, you need to declare the value for **name** and the value for **turnID**, both surrounded by `<` and `>`, as in this example:

```
PLAYER { /* there is nothing here */ } 
SETUP 
{
    player p1 = <“john”, 1>;    // name is “john”; turnID is 1 */
    p1.name = “jane”;            // changes p1’s name to “jane” */
    p1.turnID = 2;               // changes p1’s turnID to 2 */
}
```

As indicated in the above code, a player is declared with the **player** keyword.  

**Note**: capitalization is important! **PLAYER** is used to denote the block of code where player subtypes are specified, but **player** is used to declare a player variable.

### 1.8 Anytype

Anytype is a pseudo-type that may not be instantiated, but may be used as a function parameter type or a function return type. (See 3.5 for information on function declarations.) In CGL, anytype is used exclusively with lists, in order to handle the fact that lists can hold any data type. As such, it is the type of the **ele** keyword (see 3.4), as well as the return type of the list remove operator (see 2.2.6).

Example of anytype and the list remove operator:

```
list bets = [1, 2, 3, 4, 5];
bool bet = bets ->;     // throws an error */
int bet = bets ->;       // succeeds */
```

For examples of anytype being used in functions, and for examples anytype and the **ele** keyword, see the **find()**, **in()**, **insert()**, **get()**, and **remove()** functions in Appendix B. The **anytype** keyword is only ever used in functions of that sort.
2. Lexical Conventions

2.1 Identifiers

CGL uses identifiers in order to represent variables and functions. An identifier is a sequence of letters, digits, and the underscore character “_”. The first character of an identifier must be a letter. CGL is case sensitive; i.e., upper and lower case letters are considered different.

Examples of valid identifiers:
hello
myHand
test4
a_b_c
identifier
IDENTIFIER
IdEnTiFiEr

Note: CGL treats the last three identifiers as distinct.

Examples of invalid identifiers:
67
my-hand
_bad
$%@^&

2.2 Operators

An operator is used to manipulate the values of data types, or to assign a value to an identifier.

2.2.1 Assignment Operator

The assignment operator is the equals sign (=). This is used to assign a value to an identifier.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
<td>int i = 0; int r = random(1, 52); card a = $4S;</td>
</tr>
</tbody>
</table>
### 2.2.2 Arithmetic Operators

Arithmetic operators are used to manipulate integers and doubles.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
</table>
| -        | Negative sign | int a = -4; /* -4 */  
double b = -4.0; /* -4.0 */ |
| +        | Addition | int a = 9 + 4; /* 13 */  
double b = 9. + 4.; /* 13.0 */ |
| -        | Subtraction | int a = 5 - 2; /* 3 */  
double b = 5.2 - 9.1; /* -3.9 */ |
| *        | Multiplication | int a = 65 * 0; /* 0 */  
double b = 7.2 * 1.1; /* 7.92 */ |
| /        | Division | int a = 30 / 5; /* 6 */  
double b = 6. / 0.5; /* 12.0 */ |
| %        | Modulo (integers only) | int a = 6 % 4; /* 2 */ |

Note: When you combine a double and an integer in an arithmetic expression, the result becomes a double. For example, 5.6 + 5 = 10.6, or 3.0 * 7 = 21.0. Additionally, when the division operator is used with two integers, it is treated as integer division; for example, 7 / 2 gives you 3, not 3.5. For double division, at least one of the two arguments must be a double; for example, 7.0 / 2 gives you 3.5.

### 2.2.3 Relational Operators

Relational operators take two data types and return a boolean value describing their relation. For all relational operators except for `===` and `!==`, the data types you're comparing must be of the same type; otherwise, you will get an error. Furthermore, the `>=`, `<=`, `>`, and `<` operators can only be used to compare integers and doubles; otherwise, you will get an error.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
</table>
| >=       | Greater than or equal to | bool a = 5 >= 3; /* true */  
bool b = 6.5 >= 9.2; /* false */ |
| <=       | Less than or equal to | bool a = 5 <= 6; /* true */  
bool b = 7.2 <= 6.0; /* false */ |
| >        | Greater than | bool a = 5 > 4; /* true */  
bool b = 4 > 4; /* false */ |
| <        | Less than | bool a = 5. < 6.; /* true */  
bool b = 6. < 6.; /* false */ |
| ==       | Equal to | bool a = 5 == 5 /* true */  
bool b = 6 == 5 /* false */  
bool c = 5 == "5" /* error */  
bool d = "a" == "a" /* true */  
bool e = $JS == $*S /* true */  
bool f = [] == [[]] /* false */  
bool g = true == false /* false */ |
## 2.2.4 Boolean Operators

Boolean operators are used to perform logic operations on boolean expressions. As in Java, CGL uses the short-circuit evaluation method to generate the behaviour of the various operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Not</td>
<td>bool a = true; a = !a; /* false */</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>And</td>
<td>bool a = true &amp;&amp; true; /* true */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.5 String Operator

String operators are used to manipulate strings.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Concatenate</td>
<td>string a = “foo” ^ “bar”; /* “foobar” */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>string b = a ^ “baz”; /* “foobaz” */</td>
</tr>
</tbody>
</table>
2.2.6 List Operators

List operators are used for adding and removing elements from the beginning and the end of lists. These operators allow lists to be implemented as both queues and stacks.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
</table>
| `e` `+>` `l` | Add element `e` to the front of list `l`. | `list l = [3, 2, 1];`  
`4 +> l;`  
/* `l` becomes [4, 3, 2, 1] */ |
| `l` `<+` `e` | Add element `e` to the end of list `l`. | `list h = [$AC, $AD, $AH]`  
`h <+ $AS;`  
/* `h` becomes [$AC, $AD, $AH, $AS] */ |
| `<-` `l` | Remove an element from the front of list `l`. | `list b = [true, false];`  
`<- b;`  
/* `b` becomes [false] */ |
| `l` `<-` | Remove an element from the end of list `l`. | `list f = [1, 1, 2, 3, 5, 8];`  
`int i = f <-;`  
/* `f` becomes [1, 1, 2, 3]`  
`i` becomes 8 */ |

Note the importance of the direction of the add operator when dealing with two lists. For example:

```
list a = [1, 2, 3] +> [4, 5]; /* `a` becomes [[1, 2, 3], 4, 5] */
list b = [1, 2, 3] <+ [4, 5]; /* `b` becomes [1, 2, 3, [4, 5]] */
```

2.2.7 Order of Operations

The following examples illustrate the order of operations for arithmetic operators (2.2.2), boolean operators (2.2.4), and list operators (2.2.6) in CGL.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Results</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1+2*3</code></td>
<td>7</td>
<td>The multiplication (*) operator takes precedence over the addition (+) operator.</td>
</tr>
<tr>
<td><code>(1-2)*3</code></td>
<td>-3</td>
<td>The expression within the parentheses is evaluated first before the multiplication operator.</td>
</tr>
<tr>
<td><code>1. / 2. * -4.</code></td>
<td>-2.0</td>
<td>The order of operations is left to right for division, multiplication, and a negative value. Therefore, / takes precedence over *, followed by - in this expression.</td>
</tr>
<tr>
<td>`false &amp;&amp; true</td>
<td></td>
<td>false`</td>
</tr>
<tr>
<td>`true &amp;&amp; (false</td>
<td></td>
<td>true)`</td>
</tr>
</tbody>
</table>
The top card of the deck is removed and placed in the back of the hand.

The remove (→ or ←) operator always evaluates before the add (→+ or ←+) operator.

2.3 Punctuators

Punctuators are used to handle program flow. They also help to make the program more readable.

<table>
<thead>
<tr>
<th>Punctuator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Ends a line of code for a statement.</td>
<td>int a = 2;</td>
</tr>
<tr>
<td>( )</td>
<td>Declares the parameters of a conditional, loop, or function.</td>
<td>if (a &lt; 3) { /* conditional body <em>/ } while (a &lt; 3) { /</em> loop body <em>/ } foreach (l) { /</em> foreach body <em>/ } def int len(list l) { /</em> function body */ }</td>
</tr>
<tr>
<td>{ }</td>
<td>Declares the parameters of a conditional, loop, or function. Also used to delineate the PLAYER, SETUP, TURN, and WIN blocks.</td>
<td>PLAYER { /* player info <em>/ } SETUP { /</em> setup <em>/ } TURN { /</em> turn info <em>/ } WIN { /</em> win conditions */ }</td>
</tr>
</tbody>
</table>

2.4 Comments

Comments within CGL begin with /* and end with *//, as in C. Multi-line comments are supported in CGL. Also, as in C, comments are not nested.

/* This is a comment, and it is multiline. */

/* This is all /* part of the /* same comment. */

/* This is a comment. */ This is an error! */

2.5 Preprocessing

Inclusion of external library functions is performed by using the preprocessing macro #include. For global scope, this must be done in the SETUP block, although CGL allows you to use #include in other blocks for local scope. After the #include statement, the CGL
external library filename must be included as a string, and the whole expression must end in a semicolon.

Example:

```cgl
#include "library.cgl1";
```

For more information on external libraries, see section 4.3

### 2.6 Keywords

The following list contains all eighteen keywords in CGL, which are restricted from being using for other purposes within a program:

```cgl
anytype
bool
card
def
double
ele
else
false
foreach
if
int
list
return
player
string
true
while
your
```
3. Control Flow

CGL executes different expressions in different orders, depending on how those expressions are laid out. This is broadly termed “control flow”. The following are five mechanisms CGL uses to handle control flow.

3.1 Statements

A statement is a piece of code followed by a semicolon. Statements are evaluated sequentially.

Example:
```c
list deck = [$AC, $2C, $3C, $4C, $5C, $6C, $7C, $8C, $9C, $10C];
dock ->;
dock = shuffle(deck);
```

3.2 Conditionals

Conditionals, or “if / else” statements, consist of an if statement, an optional number of else if statements, and then an optional concluding else statement.

```c
if (condition1)
{
    statement1;
}
else if (condition2) /* optional */
{
    statement2;
}
else if (condition3) /* optional */
{
    statement3;
}
...
else /* optional */
{
    statementk;
}
```

Each condition must be a boolean. statement1 will execute if condition1 is true, otherwise statement2 will execute if condition2 is true, and so on until statementk, which will execute only if all of the previous condition k-1 boolean expressions are false.
Furthermore, CGL allows you to omit the curly braces after an \texttt{if} or \texttt{else} declaration if the succeeding statement is only one line of code. This convention is also used in languages like Java and C.

Example:

```c
/* prints a message corresponding to a random die roll */
int dieRoll = random(1, 6);
if (dieRoll == 1)
    print("you rolled a 1\n");
else if (dieRoll == 2)
    print("you rolled a 2\n");
else
    print("you rolled a number greater than 2\n");
```

### 3.3 While Loops

While loops will execute a statement until a certain condition is reached. The condition is checked for before the body of the loop is evaluated.

```c
while (condition)
{
    statement;
}
```

The \texttt{condition} must be a boolean, and the statement executes at each iteration until the condition returns false.

Examples:

```c
/* deals five cards from the deck into myHand */
int i = 0;
while (i < 5)
{
    myHand <+ <- deck;
    i = i + 1;
}

/* puts every card in my hand on the bottom of the deck */
while (myHand != [])
{
    deck <+ <- myHand;
}
```
3.4 Foreach Loops

The foreach keyword is used to iterate through lists. The ele keyword (short for “element”) represents the current item in the list. The foreach loop exits once all items in the list have been touched.

```c
foreach (listname)
{
    statement;
}
```

The listname must be a list, and the statement occurs once for each item in the list.

Examples:

```c
/* adds up the card values in a hand (see 5.1.1.4 for the meaning of value())*/
int totalValue = 0;
foreach (hand)
{
    totalValue = totalValue + value(ele);
}

/* reverses the elements in myList */
list reversed = [];
foreach (myList)
{
    ele很重要 reversed;
}
myList = reversed;
```

Note that foreach has nothing to do with the for( ; ; ) construction in Java or C. To accomplish something like for( ; ; ), use a while loop. (See the first example in 3.3.)

3.5 Function Calls

The def keyword is used to declare functions in CGL. The return type of the function follows the word def. All functions must conclude with a return statement, which consists of a return keyword, followed by some data of the type you declared after the def keyword. Furthermore, all functions in CGL are pass-by-value, not pass-by-reference. As a result, there is no void keyword in CGL, and you cannot alter a value that you pass in as a parameter.
Examples:

/* Returns the length of list l. */
def int length(list l)
{
    int length = 0;
    foreach (l)
    {
        length = length + 1;
    }
    return length;
}

/* Returns true if element e is in list l. */
def bool in(anytype e, list l)
{
    bool in = false;
    foreach(l)
    {
        if (ele === e && ele == e)
            in = true;
    }
    return in;
}

Both of these examples are found in Appendix B. For more examples of functions see Appendix A and the rest of Appendix B.
4. Program Layout and Scoping

The layout of CGL program consists of up to four labeled “blocks” of code, which run in a set order that resembles the flow of most common card games. A CGL source file uses the extension .cgl, and a CGL library file uses the extension .cgll.

4.1 The Four Blocks

Each CGL program contains up to four main types of blocks of code: PLAYER { }, SETUP { }, TURN n { }, and WIN { }. Only the SETUP block is required; everything else is optional, but will probably be used in all but the simplest CGL programs. Furthermore, the PLAYER, SETUP, and WIN blocks can only occur once in the program, while you can declare as many TURN blocks as you like. Finally, the four blocks must be declared in order: PLAYER, SETUP, some number of TURN n, and WIN.

The next four sections describe each of the four blocks. The examples in each section can be combined to construct a complete card game, which is only 38 lines of code!

4.1.1 PLAYER

The PLAYER block sets out the subtypes of the player datatype. This block can only have variable declarations; no function declarations. It is the very first block to run in a CGL program, and the only time it ever runs is at the beginning of a program.

If you do not include a PLAYER block in your program, all player datatypes will only include the two default subtypes: name and turnID (see 1.7 for more information).

Example:

/* This gives each player in the game a score, a turn count, and a next player. */
PLAYER
{
    int score = 0;
    int turnCount = 0;
    player next = NEMO;
}

4.1.2 SETUP

The SETUP block is the main block of code in a CGL program, and the only required block in CGL. It runs immediately after the PLAYER block has concluded if there is a PLAYER block; otherwise, it is the first block run in the program. Inside the SETUP block, you can declare global...
variables and define functions that are accessible to all other parts of the program, i.e. the TURN and WIN blocks. Once you leave a SETUP block, you cannot return to it—the rest of the program will execute solely using the TURN and WIN blocks.

Example:

/* This setup block declares two players, sets out the player order, creates a standard deck, shuffles it, and finally calls the turn function on the first player. */
SETUP
{
    string name1 = scan();
    string name2 = scan();
    player p1 = <name1, 1>;
    player p2 = <name2, 1>;
    p1.next = p2;
    p2.next = p1;
    list deck = STANDARD;
    deck = shuffle(deck);
    turn(p1);
    /* you need to include a turn() or win() function at the end of a SETUP block if you want to continue the program */
}

4.1.3 TURN n

The TURN block is executed whenever a piece of code calls the turn(player p) function. Each TURN block is followed by an integer n, which by convention should be a positive integer. The n corresponds to the default turnID subtype of the player datatype. If a player john has john.turnID equal to 1, then the turn(john) function will cause the TURN 1 block to run. If a player jim has jim.turnID equal to 2, then turn(jim) will cause the TURN 2 block to run. If a player has a turnID without a corresponding TURN n block, CGL will take turn(playerWithBadID) and run the WIN block instead.

Within the TURN block, the player whose turn it is can be accessed with the keyword your.

Example:

/* If the top card of the deck is a red card, give the player a point. Then, put the card on the bottom of the deck. If the player has moved five times, move to the win block. */
TURN 1
{
    if (your.turnCount >= 5)
        win();
    card c = <- deck;

    if (c == $*D || c == $*H)
        your.score = your.score + 1;
    deck <+ c;
    your.turnCount = your.turnCount + 1;
    turn(your.next);
}

4.1.4 WIN

The WIN block is a block of code that runs whenever the win() function is called. It is generally used at the end of the program to check win conditions. Unlike TURN n, there is no designated player whose subtypes are being modified when you call win(); rather, WIN just deals with variables and functions, like in the SETUP block.

Example:

/* Tests to see which player drew more red cards, and declares that player the winner. */
WIN
{
    if (p1.score > p2.score)
        print(p1.name ^ “ wins\n”);
    else if (p1.score < p2.score)
        print(p2.name ^ “ wins\n”);
    else
        print(“draw\n”);
}

4.2 Scoping

All identifiers declared in the SETUP block are global, and accessible to the TURN n and WIN function of the program. Otherwise, the CGL language uses block level scoping; the variables and functions declared within TURN n and WIN are local to those blocks of the program. Furthermore, variables declared within conditionals, while loops, foreach loops, and function definitions are local to those blocks. In each scope, identifiers must be declared before use; in other words, it is not allowed for a statement with an identifier to precede the statement in which the identifier is declared.
Example:

/* note that any underlined code indicates that that code will throw an error */

PLAYER
{
    /* Any variables declared in the PLAYER block will be considered subtypes of the player datatype, not global or local variables */
    list hand = [];
    player next = NEMO;
}

SETUP
{
    /* any variables declared in the SETUP block will be usable in the rest of the SETUP body, as well as the TURN n and WIN blocks. */
    int x = 4;
    card c = $2H;
    list deck = STANDARD;

    /* This throws an error, as z has not yet been defined. */
    int y = 3 + z;

    int z = 9;
    int y = 3 + z;  /* Now this will work fine. */

    /* This throws an error, as cut() has not yet been defined. */
    deck = cut(deck);

    /* Any functions declared in the SETUP block will be usable in the rest of the SETUP body, as well as the TURN and WIN blocks. */
    def list cut(list deck)
    {
        /* function body */
    }

    /* Now this will work fine. */
    deck = cut(deck);
TURN 1
{
    /* This uses the values for x and y declared in SETUP. */
    int n = x + y;

    /* This throws an error, because y was already declared in SETUP. */
    int y = 2;

    /* This changes the value of y for the whole program. */
    y = 2;

    /* This only changes the value of n for the rest of the TURN block. The next time the TURN block is called, n will revert back to x + y, as expressed in the first line. */
    n = 40;

    /* Here, the parameters x and y are NOT the same x and y from SETUP. If you want to change the global x and y in this function, you would need to choose different parameter names. */
    def int power(int x, int y)
    {
        /* function body */
    }

    /* i becomes 16 (4 to the power of 2) */
    int i = power(x, y);
}

TURN 2
{
    /* This does not work, because n was declared in TURN 1, which TURN 2 does not have access to. */
    n = n + 1;
}
WIN
{
   /* This works fine because deck and cut() were declared in
      SETUP. */
   deck = cut(deck);

   /* This does not work because power() was declared in
      TURN 1, which WIN does not have access to. */
   int j = power(x, y);
}

4.3 External Libraries

CGL supports the creation and implementation of external libraries. A library is simply a
collection of functions that a CGL program can call at anytime. A CGL library file does not
contain any blocks of code; it is merely a set of function declarations without any extra syntax.
The `#include` preprocessing macro (discussed in section 2.5) literally copies the exact text of
the CGL library into the CGL program. Therefore, any extra syntax in the CGL library file would
mess up the overall syntax of the CGL program.

CGL library files use the extension `.cgll`, and an example library file can be found in Appendix
B.
5. CGL Core Library

The core library within CGL contains a number of functions for converting data types, printing to
and scanning from the command line, managing control flow, and randomization. There are also
two constants, one list and one player, which CGL will always recognize. The core library is
always loaded into every CGL program file; you do not need to include an #include statement
to get access to these functions.

5.1 Functions

5.1.1 Data Conversion Functions

5.1.1.1 string intToString(int i)

This function takes in an integer value as an argument, and returns it as a string. The integer
can be either positive or negative.

Examples:

string a = intToString(5); /* "5" */
string b = intToString(-8); /* "-8" */

5.1.1.2 int stringToInt(string s)

This function takes in a string that represents a numeric value as an argument, and returns it as
an integer. The string can include either a positive or a negative integer value. If the string
contains anything other than digits and a minus sign, this function will throw an error.

Examples:

int a = stringToInt("10"); /* 10 */
int b = stringToInt("-12"); /* -12 */

5.1.1.3 string suit(card c)

This function takes in a card type, and returns a string which represents a suit. The suit
function only accepts allowable cards, and returns the respective suit: “C”, “D”, “H”, or “S”. If
you call suit() on a card with a * as a suit, this function will throw an error.

Examples:

string a = suit($JH); /* "H" */
string b = suit($2C); /* "C" */
5.1.1.4 int value(card c)

This function takes in a card type, and returns an integer which represents the value of the card. Values for each card range from 2 through 14. If you call value() on a card with * as a value, this function will throw an error.

Examples:

```plaintext
int a = value($5S);       /* 5 */
int b = value($KD);        /* 13 */
```

5.1.2 Input/Output

5.1.2.1 int print(string s)

This function prints the given string to the command line. It returns the value 0.

Example:

```plaintext
print("hello world");    /* Prints hello world to the command line. */
```

5.1.2.2 string scan()

This function reads in everything on the command line up to a newline \n character, and stores the characters as a string, minus the newline character.

Example:

```plaintext
string input = scan();    /* If the user types in “foo bar\n”, then input becomes “foo bar” */
```

5.1.3 Control Flow

5.1.3.1 int turn(player p)

This function will cause the current block of code to halt, and then will execute the code located with the TURN n block, where n is the value of p.turnID. If p.turnID does not correspond with an existing TURN n block, this function will execute the code located in the WIN block.

As with the print() function, the turn() function always returns 0. The examples in section 4.1 and section 6 give appropriate illustrations of the turn() function.
5.1.3.2 int win()

This function will cause the current block of code to halt, and then will execute the code located in the WIN block.

As with the print() function, the win() function always returns 0. The examples in section 4.1 and section 6 give appropriate illustrations of the win() function.

5.1.4 Randomization

5.1.4.1 int random(int lower, int higher)

The random() function takes in two integer arguments, and returns an integer that is a random number between the range of the two input integers lower and higher, inclusive with those bounds.

Example:

```c
int i = random(1, 52); /* i is an integer between 1 and 52, inclusive. */
```

5.1.4.2 list shuffle(list l)

The shuffle() function takes in a list, and randomizes the position of the elements within the list, and returns the shuffled list.

Example:

```c
list aces = [$AC, $AD, $AH, $AS];
aces = shuffle(aces); /* makes aces a version of the original list, but with the original positions randomly moved. */
```

5.2 Constants

5.2.1 NEMO

NEMO is the default player. The word “nemo” is Latin for “no one”, and so NEMO should be used in places where the player is not known. NEMO’s name subtype is “nemo”, and NEMO’s turnID is -1. Because TURN n blocks should not include negative numbers for n, calling turn(NEMO) will redirect you to the win() block. Should you create a TURN -1 block, calling turn(NEMO) will redirect you to that block, unless you want to change NEMO’s turnID to an integer without a corresponding TURN n block.
5.2.2 STANDARD

STANDARD is a list containing the standard 52-card deck, which holds the cards in this order:

```
STANDARD =
[$2C, $3C, $4C, $5C, $6C, $7C, $8C, $9C, $10C, $JC, $QC, $KC, $AC,
 $2D, $3D, $4D, $5D, $6D, $7D, $8D, $9D, $10D, $JD, $QD, $KD, $AD,
 $2H, $3H, $4H, $5H, $6H, $7H, $8H, $9H, $10H, $JH, $QH, $KH, $AH,
 $2S, $3S, $4S, $5S, $6S, $7S, $8S, $9S, $10S, $JS, $QS, $KS, $AS]
```

Calling STANDARD does not shuffle the deck for you. Generally speaking, you will always want to use the following programming idiom when using STANDARD:

```
list deck = STANDARD;
deed = shuffle(deck);
```
Appendix A. Sample Game: Simplified Blackjack

PLAYER
{
    player next = NEMO;
    list hand = [];
    int score = 0;
    bool bust = false;
}

SETUP
{
    #include "listlibrary.cgll";  /* contains in(), reverse(),
                                and get() functions */

    /* In this function, valid IDs must be a list of ints */
    def list addPlayers(list validIds)
    {
        bool done = false;
        list players = [];
        while(done == false)
        {
            print("please enter player name\n");
            string name = scan();
            print("please enter player id or \"d\" if done\n");
            string id = scan();
            if (id == "d")
                done = true;
            else
                {
                    int turnID = stringToInt(id);
                    if (in(turnID, validIds)) /* in list library */
                        players <+ <name, id>;
                    else
                        print("invalid input\n");
                }
        }
        return players;
    }
}
/** In this function, hand should be a list of cards. Also, for the sake of simplicity, we’ve made ace = 11 in all cases. */

def int score(list hand)
{
    int score = 0;
    foreach (hand)
    {
        if (ele == $A*)
            score = score + 11;
        else if (ele == $J* || ele == $Q* || ele == $K*)
            score = score + 10;
        else
            score = score + value(ele);
    }
    return score;
}

/* adds the players to the game */
print(“player ID is 1; AI ID is 2
”);
list players = addPlayers([1, 2]);

/* sets player order */
players = reverse(players); /* in list library */
player prev = NEMO;
foreach (players)
{
    ele.next = prev;
    prev = ele;
}
players = reverse(players);

/* sets bust number to 21 */
int maxScore = 21;

/* generates the deck and shuffles it */
list deck = STANDARD;
deck = shuffle(deck);

/* deals two cards to every player */
foreach (players)
{
    int i = 0;
while (i < 2)
{
    ele.hand += deck;
}

/* starts the game */
turn(get(1, players)); /* in list library */

/* human turn */
TURN 1
{
    your.score = score(your.hand);
    bool done = false;
    print("type \"h\" for hit; anything else for stay\n");
    string s = scan();
    if (s == "hit")
    {
        your.hand += deck;
        your.score = score(your.hand);
        if (your.score > maxScore)
        {
            your.bust = true;
            turn(your.next);
        }
        else
        {
            turn(your);
        }
    }
    else
    {
        turn(your.next);
    }
}

/* AI turn */
TURN 2
{
    your.score = score(your.hand);
    if (your.score <= 14) /* AI only hits if its score is lower than 14 */
    {
        your.hand += deck;
        your.score = score(your.hand);
        if (your.score > maxScore)
        {
        }
your.bust = true;
    turn(your.NEXT);

} else
    turn(your);

} else
    turn(your.NEXT);

}

WIN
{
    player best = NEMO;
    int bestScore = 0;
    foreach (players)
    {
        if (ele.bust == false)
        {
            if (best == NEMO)
            {
                best = ele;
                bestScore = ele.score;
            }
            else if (ele.score > bestScore)
            {
                best = ele;
                bestScore = ele.score;
            }
        }
    }
    print("the winner is " ^ best.name ^ ".
")
}
APPENDIX B. Sample Library: List Library

/* Returns the length of list l. Runs in O(n). */
def int length (list l)
{
    int length = 0;
    foreach(l)
    {
        length = length + 1;
    }
    return length;
}

/*/  
Returns a list of indices where element e appears in l. Runs in O(n).
Examples:
    find(0, [0, 1, 1, 0, 0, 1, 0]); returns [1, 4, 5, 7]
    find(1, [0, 1, 1, 0, 0, 1, 0]); returns [2, 3, 6]
    find(2, [0, 1, 1, 0, 0, 1, 0]); returns []
    find("0", [0, 1, 1, 0, 0, 1, 0]); returns []
Note: indices start at 1, not 0!
*/
def list find(anytype e, list l)
{
    list res = [];
    int index = 1;
    foreach(l)
    {
        if (e === ele && e == ele)
            res <+ index;
        index = index + 1;
    }
    return res;
}

/*/  
Returns the element at the specified index. If index is too high, returns -1. Runs in O(n) and not O(1) because CGL lists are not arrays.
Note: indices start at 1, not 0!
*/
def anytype get(int index, list l)
{
    if (index > length(1))
```python
print("index too high\n");
return -1;
}
else
{
    int i = 1;
    foreach(l)
    {
        if (i == index)
            return ele;
        i = i + 1;
    }
}

/*
Returns true if element e is in list l. Uses linear search, so runs in O(n).
*/
def bool in(anytype e, list l)
{
    bool in = false;
    foreach(l)
    {
        if (ele == e && ele == e)
            in = true;
        return in;
    }
}

/*
Returns a list identical to l, except that there is an additional element positioned after the given index. If the index is too high, returns l. Runs in O(n).
Note: indices start at 1, not 0!
*/
def list insert(int index, anytype e, list l)
{
    int length = length(l);
    if (index > length)
        print("index too high; nothing added\n");
    else if (index <= (length / 2))
    {
        list before = [];
        int i = 0;
        while (i < index)
```
def list_remove(index, l):
    if index > length(l):
        print("index too high; nothing added\n")
    else:
        before = []
        i = 0
        while i < index:
            before += l
            i += 1
        l = before + l
        while before != []
            l = l + before
            before = []
        after = []
        i = length(l)
        while i > index:
            after += l
            i -= 1
        l = l + after
        while after != []
            l = l + after
            after = []
        return l

def list_remove(index, l):
    """Returns a list identical to list l, except with the element at the specified index removed. If the index is too high, returns l. Runs in O(n). Note: indices start at 1, not 0!"""
    if index > length(l):
        print("index too high; nothing added\n")
    else:
        before = []
        i = 0
        while i < index:
            before += l
            i += 1
        l = before + l
        while before != []
            l = l + before
            before = []
        after = []
        i = length(l)
        while i > index:
            after += l
            i -= 1
        l = l + after
        while after != []
            l = l + after
            after = []
    return l
def list reverse(list l)
{
    list reversed = [];
    foreach (l)
    {
        ele -> reversed;
    }
    return reversed;
}