COMS W4115 Fall 2012

Course Project

CGL (Card Game Language) Final Report

Kevin Henrick (kph2115)

Ryan Jones (rlj2122)

Mark Micchelli (mm3710)

Hebo Yang (hy2326)

Columbia University

December 19, 2012

1. Introduction

1.1 Motivation

Our interest in the card game domain developed from our perceived contrast between the widespread popularity and rich history of card games, and the lack of simple, flexible languages for describing them. Hundreds of game variants have been developed over the half-millenia that the standard deck 52-card deck (http://en.wikipedia.org/wiki/Standard_52-card_deck) has been used, but only a handful are commonly found on an average home computer (solitaire, poker, hearts, etc).

Although the data requirements of card games are minimal, in our case requiring only the tracking of 52 symbols and simple player information, algorithms that emerge from various rule sets can be more difficult to define with current languages. We aim to simplify this process.

1.2 Overview

Card Game Language (CGL) is a programming language that was designed for compiling turn-based card game variants which employ a standard 52-card deck. Our language consists of a set of blocks: PLAYER, SETUP, TURN n, and WIN, which each have unique requirements. The language is translated from CGL into Java, which in turn will create a program that the user can use to play a turn-based card game. The language is intended for programmers to quickly create and play card games with simple, structured, source code.

Thus, CGL makes it easier to translate popular card games to a digital form than it would with general-purpose languages such as Java or C. The language is intended to be elegant enough that even the invention of new card games will be quick and fun. The Core Library, which consists of useful card-game related functions such as shuffle, is implemented in Java and available for use by the generated Java code. Therefore, a card game enthusiast who is not very familiar with programming can use CGL as a means to create her desired digital game.

1.3 Design Goals

1.3.1 Structured

To simplify the complexity of card game logic, we wanted CGL programs to be well structured using a simple, turn-based framework. This would make programs easier to read, and also allow the user to focus on programming the specifics of their game, rather than high-level logic shared by most card games.

1.3.2 Intuitive

In order for people who enjoy card games to enjoy CGL, the connection between a physical card game and a program must be simple and clear. Restricting our problem space allows us to map CGL's features more directly to the requirements of physical card games, and allows CGL programming to be more accessible than programming in a general-purpose language.

1.3.3 Concise

One simple measure for our language's usefulness relative to a general purpose language would be comparing the amount of code necessary to implement a specific game. Our language uses special syntax for game logic and list operations to allow for short, powerful statements and smaller programs.

1.3.4 Modular

Since card games have many variations as well as many strategies, we designed CGL to allow easy swapping of blocks of game logic or of AI strategies. This is possible because of the structured, intuitive organization of the game definition in the source program.

2. Language Tutorial

2.1 Getting Started with the CGL Compiler

2.1.1 Compiler Requirements

Card Game Language (CGL) can create Java source code without the Java compiler being installed. However, in order to run a working program in CGL, the most recent Java compiler needs to be installed. The following commands are used to compile and run a CGL program called source:

```
$ .cgl/ -j source.cgl

$ javac *.java

$ java Main
```

If the program fails to run, the programmer can validate the CGL code by using the semantic analyzer, which will return potential errors, such as undefined variables, undefined functions, improper argument types of a function, blocks not in correct order, etc. The -s action is used to call the semantic analyzer on the program as follows:

```
$ .cgl/ -s source.cgl
```

2.1.2 Installing the CGL Compiler

To install the compiler, all of the files should be installed to a single directory. The necessary files required to run a CGL program are scanner.mll, parser.mly, ast.mli, semantic_analyzer.ml, corelibrary.ml, javaclasses.ml, cgl.ml, and Makefile. By using the

`$ Make` command, each of the component files will compile.

2.2 Example 1: HighLow

A simple, single player card game that employs the 52-card deck. After a standard deck is shuffled, a player will guess whether the next card will be higher or lower. The game ends when the player has made an incorrect guess. Coincidentally enough, CGL creates this game in 52 lines of code.

2.2.1 The SETUP block

In order for CGL to compile, each program must have a SETUP{ } block. It is the only mandatory block of code that is required by CGL. For this example of the game HighLow, the SETUP block can be seen below:

```
SETUP
{
 int score = 0;
 list deck = STANDARD;
 deck = shuffle(deck);
 player p = <"", 1>;

 card c = <- deck;
 int lastValue = value(c);
print("the first card has value " ^ intToString(lastValue) ^ "\n");
deck <+ c;
turn(p);
 }
```

2.2.1 The TURN n Blocks

The TURN blocks can be created any number of times (i.e. TURN 1, … , TURN n) within CGL. A turn block is accessed when the turn() function is called with a player as an argument. The numeral after TURN corresponds to the turnID of the player. Within the TURN blocks, game rules and player strategies are declared, whether the player be a human or an artificial intelligence agent. An example of using multiple TURN blocks would be if the programmer wished to create a game with a human, conservative AI player, and an aggressive AI player. In this case, three TURN blocks may be declared. Within any TURN block, the player with whom the block was called can be accessed with the keyword "your". For example, variables declared in the PLAYER block can be accessed in a TURN block by using a "your.variable_name" expression. (The keyword “your” was actually chosen because it creates some nice phrases when paired with variables, like “your.name” and “your.turnID”.)

```
TURN 1
{
bool properInput = false;
bool high = true;
while (!properInput)
{
    print("will the next card be (h)igher or (l)ower?\n");
    string guess = scan();
    properInput = true;
    if (guess == "l")
        high = false;
    else if (guess != "h")
    {
        print("invalid input\n");
        properInput = false;
    }
}

c = <- deck;
int thisValue = value(c);
deck <+ c;
print("new card's value is " ^ thisValue ^ "\n");
if ((thisValue > lastValue && high) || (thisValue < lastValue && !high))
{
      print("correct prediction\n");
      score = score + 1;
      lastValue = thisValue;
      turn(p);
}
else
{
      print("incorrect prediction; game over\n");
      print("total score = " ^ intToString(score) ^ "\n");
}}
```

Note that the PLAYER and WIN blocks were not called in this example of HighLow. This is because the only required block in CGL is the SETUP block, and simple programs such as HighLow can be executed with two blocks, or even just the SETUP block. For more complex games, the PLAYER and WIN blocks can be used to store player information and more complex winning conditions.

2.2.2 The Playable HighLow Program

The following shows the player interface for HighLow. The player is first shown a card with a value of 10, and guesses that the next card will be lower by clicking (**l**). The next card has a value of 2, so she correctly predicted, and is able to guess again. Since 2 is the lowest possible card value, she guesses that the next card will be higher by clicking (**h**). The next card is an 8, so she goes again. She guesses higher again, but the next card is 5 (lower than 8). Therefore, the game ends, and the user sees her overall score (2 = number of correct guesses).

```
the first card has value 10

will the next card be (h)igher or (l)ower?

l

new card's value is 2

correct prediction

will the next card be (h)igher or (l)ower?

h

new card's value is 8

correct prediction

will the next card be (h)igher or (l)ower?

h

new card's value is 5

incorrect prediction; game over

total score = 2
```

2.3 Example 2: RedCardGame

Another simple card game created in CGL. Two players are dealt cards from the standard, shuffled 52-card deck. If a player receives a card that's a heart or a diamond, then he receives a point for being dealt a red card. After 5 cards are dealt to each player, the player with the higher score (number of red cards) wins the game.

2.3.1 The PLAYER Block

The PLAYER block can be used to define data fields for all players in a game. By deafult, each player has a string for its name, and an int for its turnID (which is used to call the corresponding TURN block using the turn() function). Like the TURN and WIN blocks, the PLAYER block is optional, but is necessary for non-trivial games, such as blackjack. Player data can be accessed in other blocks by using the dot operator (e.g. p**.**varName, where p is a player reference).

```
/* 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;
}
```

2.3.2 The WIN Block

The WIN block in CGL is an optional block in which the programmer can create win conditions for a game, and terminate the program. This block is run whenever the win() function is called within previous blocks. It has access to each player reference (via the **.** binary operator), global variables and functions. The WIN block for the RedCardGame can be seen below:

```
/* 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");

      }
```

2.3.3 The Rest of the Source Code for RedCardGame

The two remaining blocks for the RedCardGame (SETUP and TURN 1) can be seen below. Please note that for RedCardGame to work correctly, the blocks must be in the following order: PLAYER, SETUP, TURN 1, WIN.

```
/* 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);

    }


/* 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;

    print(your.name ^ " drew " ^ intToString(value(c)) ^ suit(c) ^ "\n");

    if (c == $*D || c == $*H)

    your.score = your.score + 1;

    print(your.name ^ "'s score is " ^ intToString(your.score) ^ "\n");

    deck <+ c;

    your.turnCount = your.turnCount + 1;

    turn(your.next);

    }
```

2.3.4 The Playable RedCardGame Program

The following shows the player interface for RedCardGame. The program first asks for two player names , then automatically draws a card for each player showing the card as well the the score for five turns, and finally shows the winner who has the highest score.

```
Kevin

Mark

Kevin drew 11D

Kevin's score is 1

Mark drew 8H

Mark's score is 1

Kevin drew 3S

Kevin's score is 1

Mark drew 11H

Mark's score is 2

Kevin drew 13C

Kevin's score is 1

Mark drew 5C

Mark's score is 2

Kevin drew 13S

Kevin's score is 1

Mark drew 11S

Mark's score is 2

Kevin drew 8S

Kevin's score is 1

Mark drew 10H

Mark's score is 3

Mark wins
```

2.4 Simplified Blackjack

A more complex card game created in CGL, Blackjack. Four players (AI or humans) are dealt two cards, with each card having an assigned score value from the range 1-11. Each player is shown her cards, and then asked if she would like to hit, which means receive another card, or stay, which means end her turn. The player(s) with the highest score without going over 21 wins the game.

2.4.1 The Playable Blackjack Game: Setup Portion

This is an example of the output for the initial setup portion of a Blackjack CGL game. In this game, three human players enter their names (**Professor Edwards**, **Mark**, and **Kevin**) along with the specification **1** to show that they're humans (note that this corresponds to the corresponding TURN 1 block that specifies their turns within CGL source code. The fourth player is an AI (called **dealer**) with a specification **2** to show that it's an AI (with corresponding TURN 2 block in the CGL source code).

```
Please enter Player name

Professor Edwards

Please enter 1 if human, or 2 if AI

1

Please enter Player name

Mark

Please enter 1 if human, or 2 if AI

1

Please enter Player name

Kevin

Please enter 1 if human, or 2 if AI

1

Please enter Player name

Dealer

Please enter 1 if human, or 2 if AI

2
```

2.4.2 The Playable Blackjack Game: Play Portion

The end of a playable part of the Black Jack CGL game is shown below. Kevin is dealt the King of Diamonds and the 4 of Hearts, and then asked if he would like to **"h"** or stay (by clicking anything else). He decides to hit, and is dealt a 2 of Spades. He now has a 16, and decides to hit again. He is now dealt the 3 of Hearts, and clicks **"s"** to stay (although anything else that is not an **"h"** would have also worked. At the end, each player's score is shown. Since Professor Edwards received a 21, he wins the game. Please note that the dealer's hand isn't revealed until the end. Since the dealer has a score of 0, the AI busted by receiving a total score greater than 21.

```
Kevin’s turn; press enter to continue

you have KD 4H

type "h" for hit; anything else for stay

h

you got a 2S

Kevin's turn; press enter to continue

you have KD 4H 2S

type "h" for hit; anything else for stay

h

you got a 3H

you have KD 4H 2S 3H

Type “h” for hit; anything else for stay

s

Professor Edwards scored 21

Mark scored 16

Kevin scored 19

Dealer scored 0

Professor Edwards wins
```

3. Language Manual

3.1 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.

3.1.1 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
```

3.1.2 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
```

3.1.3 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, unlike in C. A boolean is declared with the `bool` keyword.

3.1.4 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.

3.1.5 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`). Since most games that we've encountered use the Ace card as high, the values of the cards within CGL range from 2-14. For games like solitaire and Blackjack, conditionals may be used to alter the use of the Ace card. An exhaustive list of valid cards is shown below:

```
$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
```

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:

```
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.

A card is declared using the `card` keyword.

3.1.6 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, but CGL only allows lists to contain elements of a single type. 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., 2.]
[“string”, “foo”]
[$2D, $QC, [$AH, $KH, $QH, $JH, $10H]]
```

Examples of invalid lists:

```
[
[1; 9]
] “string” [
```

3.1.7 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). Once declared, the number and names of the subtypes is identical for every player created, and cannot be changed. However, the values of these types may be changed for each player individually. Here is an example 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 order to declare a player for the first time, you need to use the following format:

```
<"player_name", 1>;
```

The player declaration must always include two subtypes, surrounded by < and >. The first subtype is a string representing the player's name, and the second subtype is an integer representing the player's turn ID. (More on turn IDs can be found in section 4.1.3.) These two subtypes are found in *every* single player declaration by default, regardless of what's in PLAYER, and they can be accessed through the identifiers name and turnID. For example:

```
PLAYER { /* there is nothing here */ }

SETUP

{

  player p1 = <"john", 1>;

  p1.name = "jane";             /* changes p1's name to "jane" */

  p1.turnID = 2;                /* changes p1's turnID to 2 */

}
```

As indicated in the sample 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.

3.1.8 Anytype

Anytype is a pseudo-type that may not be instantiated, but may describe a function parameter type or a function return type. It is also the return type of the List remove operator (see 2.2.6).

Anytype represents a type that is strict but not known, and may be cast to its original type through assignment to a variable of that same type. Assignment of an anytype to a variable of a type that was not the anytype's original type will throw a cast error.

Example of anytype casting:

```
int bet = 5;              /* sets player1's score to 5 */

allBets <+ bet;

...

bool val = allBets ->;   /* Throws an error "invalid cast" */

int  val = allBets ->;   /* Good cast */
```

```
OR

int val = 0;
if (get(1,allBets) === val) val = get(1,allBets);
/* type check before cast */
/* get function is not included in core library */
```

3.2 Lexical Conventions

3.2.1 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

$%@^&
```

3.2.2 Operators

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

3.2.2.1 Assignment Operator

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

Operator	Meaning	Examples

=	Assignment	`int i = 0;`
		`int r = random(1, 52);`
		`card a = $4S;`

3.2.2.2 Arithmetic Operators

Arithmetic operators are used to manipulate integers and doubles.

Operator	Meaning	Examples

-	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`.

3.2.2.3 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.

Operator	Meaning	Examples

>=	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 */
!=	Not equal to	bool a = 5 != 6 /* true */ bool b = 6 != 6 /* false */ bool c = 5 != “x” /* error */ bool d = “a” != “b”; /* true */ bool e = $AC != $**; /* false */ bool f = [] != [5] /* true */ bool g = true != true /* false */
===	Type equal to	bool a = 6 === 7 /* true */ bool b = 6 === “6” /* false */ bool c = [6] === [“6”] /* true */ bool d = [] === [[]] /* true */
!==	Type not equal to	bool a = “a” !== “b” /* false */ bool b = [] !== $6* /* true */ bool c = [9] !== [9.] /* false */ bool d = 9 !== 9. /* true */

3.2.2.4 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.

Operator	Meaning	Examples

!	Not	`bool a = true;`
		`a = !a;            /* false */`

&&	And	`bool a = true && true;     /* true */`
		`bool b = true && false;    /* false */`

||	Or	`bool a = true || false;    /* true */`
		`bool b = false || false;   /* false */`

3.2.2.5 String Operator

String operators are used to manipulate strings.

Operator	Meaning	Examples

^	Concatenate	`string a = "foo" ^ "bar";   /* "foobar" */`
		`string b = a ^ "baz";    /* "foobarbaz" */`

3.2.2.6 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.

Operator	Meaning	Examples

`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, 5]`
		`   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]] */
```

3.2.2.7 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.

Expression	Results	Explanation
`1+2*3`	`7`	The multiplication (`*`) operator takes precedence over the addition (+) operator.
`(1-2)*3`	`-3`	The expression within the parentheses is evaluated first before the multiplication operator.
`1. / 2. * -4.`	`-2.0`	The order of operations is left to right for division, multiplication, and a negative value. Therefore, / takes precedence over `*`, followed by − in this expression.
`false && true || false`	`false`	The and (`&&`) operator takes precedence over the or (`||`) operator.
`true && (false || true)`	`true`	The expression within the parentheses is evaluated first before the and operator.
`hand <+ <- deck`	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.

3.2.3 Punctuators

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

Punctuator	Meaning	Examples
`;`	Ends a line of code for a statement.	`int a = 2;`

`( )`	Declares the parameters of a conditional, loop, or function.	`if (a < 3) { /* conditional body */ }`
		`while (a < 3) { /* loop body */ }`
		`foreach (l) { /* foreach body */ }`
		`def int len(list l)`
		`{`
		`    /* function body */`
		`}`

`{ }`	Declares the parameters of a conditional, loop, or function. Also used to deliniate the `PLAYER`, `SETUP`, `TURN`, and `WIN` blocks	`PLAYER { /* player info */ }`
		`SETUP { /* setup */ }`
		`TURN { /* turn info */ }`
		`WIN { /* win conditions */ }`

3.2.4 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! */
```

3.2.5 Keywords

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

```
bool

card

def

double

ele

else

false

foreach

if

int

list

return

player

string

true

while

your
```

3.2.6 External Libraries

Inclusion of external library functions is performed by writing the following pre-processing macro in the `SETUP` block:

```
SETUP

{

      #include library.cgll
```

```
        …
    }
```

This statement copies the source code from the .cgll file directly into the setup block. The .cgll library file is written in CGL and must include only function definitions. See Appendix B for an example library file.

3.3 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.

3.3.1 Statements

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

Example:

```
list deck = [$AC, $2C, $3C, $4C, $5C, $6C, $7C, $8C, $9C, $10C];

deck ->;

deck = shuffle(deck);
```

3.3.2 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.

```
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 if or else declaration if the following statement is only one line of code. This convention is also used in languages like Java and C.

Example:

```
/* 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");
```

3.3.3 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.

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

The `condition` must be a boolean, and the statement executes at each iteration until the `condition` returns false.

Examples:

```
/* 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;
}
```

3.3.4 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.

```
foreach (listname) {

      statement;

}
```

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

Examples:

```
/* 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 +> 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.)

3.3.5 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.

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;
    }
```

3.4 Program Layout and Scoping

3.4.1 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!

3.4.1.1 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 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;

}
```

3.4.1.2 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 */

    }
```

3.4.1.3 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
as 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);

}
```

3.4.1.4 WIN

The WIN block 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");

}
```

3.4.2 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 declared within TURN n and WIN are local to those blocks of the program. Functions can only be declared at the beginning of the SETUP block. 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);
}
```

3.5 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.

3.5.1 Functions

3.5.1.1 Data Conversion Functions

3.5.1.1.1 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" */
```

3.5.1.1.2 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 */
```

3.5.1.1.3 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" */
```

3.5.1.1.4 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:

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

3.5.1.2 Input/Output

3.5.1.2.1 int print(string s)

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

Example:

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

3.5.1.2.2 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:

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

3.5.1.3 Control Flow

3.5.1.3.1 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.

3.5.1.3.2 `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.

3.5.1.4 Randomization

3.5.1.4.1 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:

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

3.5.1.4.2 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:

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

3.5.2 Constants

3.5.2.1 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.

3.5.2.2 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;
deck = shuffle(deck);
```

4 Project Plan

4.1 Planning

After the Language Reference Manual was created, our group met every Saturday to assess our progress and discuss the work at hand. Observing this meeting schedule rigorously proved key to organizing our work as a team, and provided time to discuss the project, plan each portion of the code, and divide up tasks for the week ahead. As a result, almost all of the functionality that we specified in the Language Reference Manual was implemented.

4.2 Specification

Although our end-of-project implementation matches closely with the LRM, we made careful note of the additions and changes to the LRM specification. These are noted in the Appendix E.

4.3 Development Process

The development process for our language consisted of three main phases. During the initial phase, the scanner, parser, and abstract syntax tree were coded based on the requirements that were established in the Language Reference Manual. During the second phase, the generator, core library, java library, and cgl executable were created under the assumption that a program would be correctly type checked with a yet to be implemented semantic analyzer and sast. During the third phase, the semantic analyzer was created alongside a semantically analyzed abstract syntax tree (sast), and then tested using multiple test files. Testing occurred concurrently with phases two and three. However, near the end of phase three, a bash script was used to perform multiple tests, and automate the checking process.

4.4 Style Guide

4.4.1 OCaml

Our team decided from the start to use 4-space indents in all code. Furthermore, all variables and functions would use all_lowercase_with_underscores, not camelCase. There is some inconsistency when it came to the let … in construction. For short declarations, the “in” was kept in the same line as the last word of the function. For longer declarations, in was given a whole new line. We acquired this rule-of-thumb from Professor Edwards’s MicroC code, which seems to follow the same rules.

4.4.2 Java

Because the Java code is created by the generator, it was difficult to get things like indenting and newlines to display in a readable fashion. However, all of the Java variables are in camelCase, as typical in Java code. Additionally, the Java code also sometimes needs to create new variables unrelated to the variables declared in the CGL code. We managed to avoid naming overlap by appending each variable name with a certain number of '$' characters, because '$' can be included in Java identifiers but not CGL ones.

4.4.3 CGL Source Code

For our CGL code, we decided on 4-space indents, as in our OCaml code. Variable names were in camelCase, and brackets were treated using Allman style. As CGL is somewhat Java-like, and because we wanted to compile to Java, we tended towards traditional Java code styles when designing CGL.

4.4.4 File Directory Arrangement

All components necessary for compilation of a CGL program are inside a "_build" folder. In addition, games written in CGL are inside the "games" folder and test cases along with their bash scripts are under the "test_cgl" folder. The cgl.bash script under the root folder may be used to compile and run given game inside the "games" folder.

Example File Tree

```
/CGL
  Readme.txt
  /_build
       primary_sources(*.ml, *.mli, *.mly, *.mll, …)
  Makefile
     cgl.bash
  /games
       gameA.cgl
       gameB.cgl
  /test_cgl
       analyzer.bash
       /analyzer_test
```

```
          test1.cgl

          test2.cgl

          ...

     parser.bash

     /parser_test

          test1.cgl

          test1.out

          test1.result
```

4.5 Project Timeline

Date	Project Milestone
September 5, 2012	Formed Project Team.
September 15, 2012	Decided on Card Game Language (CGL).
September 22, 2012	Created Project Proposal.
September 29, 2012	Began work on the LRM.
October 6, 2012	Created sample CGL code for War, Blackjack, and Five Card Poker.
October 13, 2012	Continued work on LRM and sample code.
October 20, 2012	Began work on scanner, parser, and test cases.
October 27, 2012	Completed LRM.
November 3, 2012	Continued work on scanner, parser, test cases,

	and began creating the Abstract Syntax Tree.
November 10, 2012	Finished scanner, parser, and AST. Began working on semantic analyzer, generator, Makefile, and executable.
November 17, 2012	Continued working on semantic analyzer, generator, test cases, and created the majority of the Make and executable.
November 24, 2012	Finished generator, continued working on semantic analyzer/SAST, and test cases.
December 1, 2012	Finished Core Library, and continued working on semantic analyzer/SAST, and test cases.
December 8, 2012	Added Java Library, continued working on semantic analyzer/SAST, and test cases.
December 15, 2012	Finished the Semantic Analyzer/SAST, and continued debugging using test cases.
December 19, 2012	Submitted Final Report, Project Files, and Presented.

4.6 Team Member Roles and Responsibilities

4.6.1 Kevin Henrick

Kevin was the team leader, and scheduled weekly meetings for the project. His main coding responsibility was to create the semantic analyzer and sast. He also helped with creating the makefile, a number of test cases, and a simple game (called First_Ace.cgl). With respect to creating the project proposal, LRM, presentation, and Final Report, he was involved with each phase of this process.

4.6.2 Ryan Jones

Ryan's primary responsibility was to create the semantic analyzer and sast alongside Kevin. He also made a number of contributions to the cgl executable file and the makefile. He contributed to each portion of the project proposal, presentation, and Final Report, and made significant contributions in developing the features of CGL within the Language Reference Manual.

4.6.3 Mark Micchelli

Mark created the scanner, parser, abstract syntax tree, generator, core library, java library, makefile, and cgl executable. He also wrote three of the sample CGL games, including the most complex game, simplified blackjack. Mark played a central role in developing the many unique features of CGL. For example, he came up with the card representations ($(value)(suit), with the asterisk representing "any"), the list operators (<+, +>, <-, and ->), the player structure with turnIDs to allow for AIs, and the PLAYER/SETUP/TURN n/WIN structure of the whole CGL language. Mark was also the primary strategist in constructing the LRM.

4.6.4 Hebo Yang

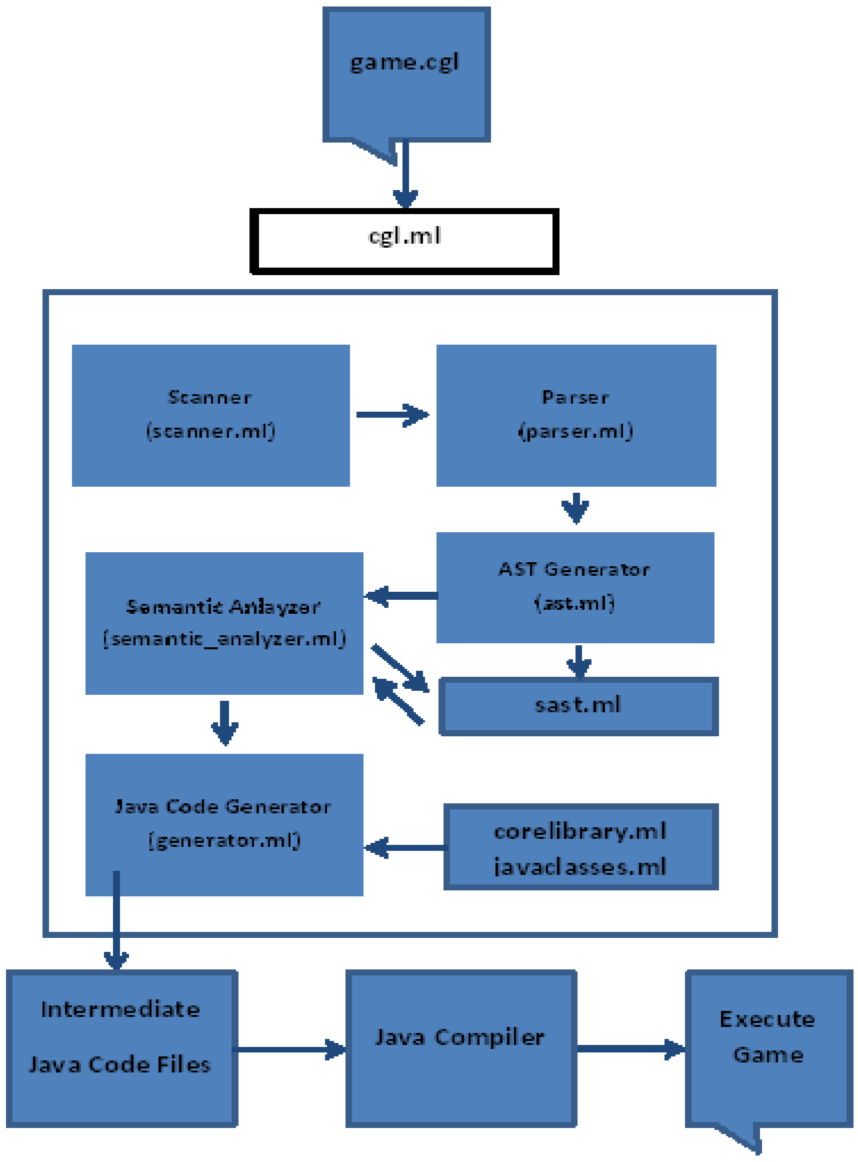
Hebo created a wide variety of test cases and two bash scripts to test inputted CGL programs. His test cases proved very effective in debugging the semantic analyzer, which allowed for greater transparency in developing the type checking system. He also created a game (War.cgl) and helped debug First_Ace.cgl. Hebo also contributed to every portion of the project, including the proposal, Language Reference Manual, presentation, and Final Report.

4.7 Software Development Environment

Our project group used Subclipse as a code sharing platform through the GoogleCode environment. Everyone on the team installed the subclipse software prior to completing the Language Reference Manual. Kevin originally created the working directory (called "_build") on the GoogleCode, and then the team quickly set-up their own accounts. It took some time to become familiar with the commands and features of Subclipse and GoogleCode, but it proved to be an effective platform for sharing OCaml, java, and cgl code throughout the project life. Mark did most of his work outside of Eclipse, just using vim and make, but he updated to the GoogleCode regularly.

5 Architectural Design

5.1 Block Diagram of Major Components



game.cgl
cgl.ml
Scanner (scanner.ml)
Parser (parser.ml)
AST Generator (ast.ml)
Semantic Anlayzer (semantic_analyzer.ml)
sast.ml
Java Code Generator (generator.ml)
corelibrary.ml javaclasses.ml
Intermediate Java Code Files
Java Compiler
Execute Game

5.2 Component Interface Interaction

5.2.1 Command Line Interface (cgl.ml – author: Mark Micchelli & Ryan Jones)

The command line interface is the executable used to compile CGL. It takes two arguments: a flag (either -s or -j) and a .cgl file. If you run the executable with -s, you will have the semantic analyzer check your program's semantics without actually generating anything. By default, you'll also check syntax errors that would be caught by the scanner or parser. If you run the executable with -j, you actually generate the Java code in the form of four .java files: Main, CGLList, Card, and Player. By compiling all of those with javac, and then running java Main, you can see your CGL program in action.

5.2.2 Scanner (scanner.mll - author: Mark Micchelli)

The scanner is used to determine the tokens that are recognizable within our language. The scanner processes the code that is written in the source.cgl file and transforms it into a series of tokens that are interpreted by the parser. The scanner is necessary to check whether valid characters are being used in a CGL program.

5.2.3 Parser (parser.mly - author: Mark Micchelli)

The parser's role is to use the series of tokens generated by the scanner and check whether the tokens can be used based on the context-free-grammar specified in our language. This is the part of the compiler that checks for syntax errors. After the parser.mly file processes the tokens generated from the scanner analyzing the source file, an abstract syntax tree is generated.

5.2.4 Abstract Syntax Tree (ast.mli - author: Mark Micchelli)

The Abstract Syntax Tree is used to define the primary relationships between the tokens within CGL. The AST is used as input in the two most important pieces code in our program: the semantic analyzer and the generator.

5.2.5 Semantic Analyzer / SAST (semantic_analyzer.ml, and sast.mli)

(author: Ryan Jones & Kevin Henrick)

The semantic analyzer traverses the nodes of the abstract syntax tree, and examines its structure. Throughout this process, the semantic analyzer type checks each node to make sure that the source code is written such that variables & functions used are previously defined with the proper return types, function calls have the proper number and types of arguments, that the block order (PLAYER, SETUP, TURN 1, … , TURN n, WIN) are in the proper order, functions are only defined at the beginning of the SETUP, and other checks specified in our Language Reference Manual. The semantic analyzer references the type checked abstract syntax tree (sast.mli) throughout this process. After the analysis is complete, and no exceptions (ie. Failure messages) are given back to the programmer, the compiler may proceed to the code generation phase of compiling a CGL program.

5.2.6 CGL Core Library and Java Classes (corelibrary.ml and javaclasses.ml- author: Mark Micchelli)

The core library functions for CGL are hard-coded Java. Our only criterion for deciding whether something should be a core function or not was: is this function impossible to write in pure CGL? If not, we made them part of the core library. As a result, core library functions are mostly used for casting, I/O, and random number generation. The one exception to this rule is the shuffle function, which *can* be written in CGL, but is so commonly used in card games that we elected to put it in the core library.

The java classes code produces the CGLList and Card functions, which stay the same regardless of the inputted CGL code. These classes are fed directly to the CGL executable for code generation.

5.2.7 Generator (generator.ml- author: Mark Micchelli)

The generator transforms the CGL code into compilable and runnable Java source code. This is the key step to actually creating working programs. Our generator reads from the AST to customize the Main and Player java files, which are then sent to the CGL executable.

5.2.8 Testing and Bash Script

(test cases and bash scripts- author: Hebo Yang)

The bash script cgl.bash under the root folder automates the compilation and running of given cgl games under the “games” folder. It also cleans up the intermediate files generated in the process. The script analyzer.bash and parser.bash under folder “test_cgl” respectfully run all the test cases under the folder “test_cgl/analyzer_test” and “test_cgl/parser_test” if no argument were given or a single test in the respective folder if a particular test name was given in the argument.

6 Test Plan

6.1 Phase 1 - Creating Scanner/Parser/AST

The process of creating test cases for the scanner, parser and AST began as soon as the LRM was finished. The style of tests resembled the one in the MicroC compiler, where programs covering all the functionalities in the LRM were written in the actual CGL language and the expected outputs were also written in .out files. The complete tested functionalities were: valid identifiers, assignment operators on all data types, arithmetic operators on int and double types, relational operators on all respective data types, boolean operators, string operators, list operators, order of operations, punctuators, comments, preprocessing, key words, all control flows, core library functions, block names, and existence of mandatory block components. Each of these test programs primarily focussed on a single functionality; complexed functionalities might depend on the successful implementation of the prior ones created.

The following program was used to test a number of features within CGL: control flow, function calls, which depend on type checked assignment operators, arithmetic operators, foreach loops, list data types, and the core library function `intToString`:

```
SETUP{
  def int length(list l)
  {
         int length = 0;
         foreach (l)
         {
                length = length + 1;
         }
         return length;
  }

  list l=[1,2];
  int a1=length(l);
```

```
    string a=intToString(a1);

    print(a);

}
```

6.2 Phase 2 - Creating the Generator assuming Correct Semantics

When the generator was completed, the test cases were able to be compiled and run, producing .result files for comparison with the corresponding .out files. During this phase, the test cases examined the integrity of the scanner, parser, AST, and the generator itself. It could first catch scanner as well as parser failures, and then successfully scanned and parsed programs would be compiled and run to further check if the generator allowed them to produce the correct results as the prewritten .out files. A bash script was written to automate the running of all test cases and compare actual outputs with expected ones. It could identity the failure cases and show the differences between actual and expected outputs for debugging purposes.

6.3 Phase 3 - Type Checking and Creating Semantic Analyzer

The test cases for semantic analyzer were written concurrently as the semantic analyzer was implemented. It was designed to catch semantic errors in the CGL programs before the generator compiled them into intermediate Java source code. The target at the end of this phase was to develop functionality to catch semantic errors. If the input programs triggered errors or warnings from the generator as well as Java compiler, the test cases would be used to find the exceptions caught by the semantic analyzer and sast. Common errors such as undeclared/duplicate variables and functions, invalid assignment due to type difference, invalid function parameters, as well as CGL specific errors like incorrect block order, referring to ele outside a foreach loop, changing an argument type inside a function, and specific block functionalities were all tested during this phase. A bash script was written to automate the running of all test cases and show the message generated by the semantic analyzer to identify the problems.

The following example code was used to test whether a function's argument type was implemented correctly based on it's original function call:

```
SETUP {

    def int check(bool c){

        if (c){

            return 0;
        }

        else {

              return 1;

        }

  }

  int a=check("true");

}

/* Please note that the above program should not run because "true"
is a String type, not a bool type as defined in the original check
function declaration. The semantic analyzer caught this exact
exception when the test case was run with the .cgl/ -s test.cgl
command. */
```

6.4 Testing CGL Programs

After all the components of CGL were implemented, and passed the respective tests, more complex CGL programs like formal playable card games written in CGL were tested. The bash scripts described in section 6.2 and 6.3 were also able to independently take the name of the CGL file, given it was saved on the corresponding folder, and test it. Furthermore, since the compilation process of CGL included semantic check, a bash script named cgl.bash could also take the name of the file and test it. Please note that all of the successfully compiled games (First_Ace.cgl, HighLow.cgl, RedCardGame.cgl, and BlackJack.cgl) all passed the semantic checking phase.

7 Lessons Learned

7.1 Kevin Henrick

7.1.1 Lessons Learned

I am admittedly a weak programmer. This was the most challenging, but also the most rewarding academic project that I have ever been apart of. While I frequently worried throughout the semester about whether our group could create a successfully functioning compiler, I very much enjoyed the process of programming with our strong team. In addition to the lessons that I learned from Professor Edwards (the simple logic behind the LR(0) automaton in parsing was particularly interesting to me), I learned a lot from my teammates throughout this project. I would conservatively estimate that I am about 100% better at programming than I was before this course.

Throughout the classroom time in Programming Languages, I learned a great deal about the intricacies of not only functional programming in OCaml, but many different types of programming languages (Java, C, C++, etc.). In particular, it was truly amazing to learn about the development of C by Dennis Ritchie and his colleagues at Bell Labs, and discover the rich history and development of modern programming languages.

Needless to say, before taking Programming Languages, I knew very little about compilers. It was really interesting to learn each phase of the process required for a compiler to work. Scanning the input, generating the characters into a sequence of tokens, parsing the tokens to build an abstract syntax tree, semantically checking each node of the tree by traversing it, generating code in a new language (ie. java), and outputting to the executable, is a process that I now say I can understand. I would have no idea what that previous run on sentence would mean before taking this class.

7.1.2 Advice for Future Teams

The standard word of advice holds true- start early. Our group formed on the first day of class, and we started brainstorming briefly thereafter. We spent a great deal of time in the beginning of the semester trying to create a solid Language Reference Manual, and that helped a lot. If I didn't know the answer to a particular feature of the language while coding, I would frequently consult the print-out of the Language Reference Manual that I kept with me.

I fortunately was able to know my teammates in advance of taking this class, and I knew that they were a great group. It's important to choose your team wisely, and stay updated on the progress being made at each weekly meeting- weekly meetings are necessary.

I had never used Google Code or Subclipse before this project, but those proved to be very effective in sharing code. Choosing a standard code-sharing environment like Google Code/ Subclipse is very beneficial.

7.2 Ryan Jones

7.2.1 Lessons Learned

Before talking about specific lessons learned, I feel it would worthwhile to describe some of the distinctive qualities I have found this project to have.

Firstly, it's a long project. We formed teams very early in the semester, and the project deadline is at the very end. Consequently it's a long time to be working with the same group people, which has great benefits but also some challenges.

Secondly, the project heavily utilizes the OCaml programming language, which for many in the class was not only a new language, but also a new paradigm, in that OCaml is a functional language. This was one of most challenging elements of the project, but also one of the most exciting and rewarding.

Thirdly, the project is a design project. The development of a programming language is rich with design choices, from language features, to intermediate representations, to supplemental languages used.

In light of the nature of the project, my primary lessons learned are the following:

Some measures of leadership can be very simple, like the yes/no answer to the following questions: Is the team meeting early and often? Are team members doing their work? Are team members communicating freely and regularly? Diagnosing and fixing problems with issues like these is not only very important, but also very doable.

Another lesson is how team members skills correspond to team roles. In this project some roles have to be more static, like having Kevin as the team leader and Mark as the technical leader, and some roles and responsibilities can be more dynamic.

ALso, it's important to consider what types of work can be done by multiple people, of how one person doing a section alone might impact other team members capacity to work on a future part, given team skills. It's important to understand your people-dependencies.

I've found it important that team members with less programming experience are still well integrated with the programming portions of the project. As an example of this, I think Kevin and I worked well together on the semantic analyzer, even though he had less familiarity. We got around this by employing partner programming and combining implementation with lots of higher-level discussion. Planning sections by blocking them out using pseudocode or interfaces and comments helped a lot as well. I consider our completion of a functional semantic analyzer to be one of my most satisfying moments in the project.

The last lesson is pretty simple, which is that OCaml is a great language for writing compilers. I really loved learning about the different variations of how functions can be created, as well as all the use of anonymous functions and chaining evaluations of functions. Pattern matching and recursion were great for picking apart and navigating data structures.

7.2.2 Advice for Future Teams

In light of the lessons that I've learned through this project, I would say the absolute most important advice I have for future teams would be the following:

"Phone calls are great"

Reliable, consistent communication is vital for this project. Guage which means of communication are most effective for individual teammates, whether it's e-mail, texting, phone calls or otherwise. When having troubles with communication, call by phone. I strongly feel that one of the most important ways our team leader's leadership skill manifested itself was through him making lots of phone calls.

"Don't be afraid of C"

The members in our group generally had more experience with Java than C, and this is one of the main reasons we chose early on to translate to Java. We have some regret over this though, because using translating certain language features was a bit complicated or ugly in Java. There's a reason that C is such a popular systems language.

A minor last bit of advice is that I found the official OCaml reference online to be a great resource in addition to the class lectures and slides. It's quite accessible, it more specifically it really helped my understanding of the nuances of type definition as well as the syntax of ocallex and ocalyacc.

7.3 Mark Micchelli

7.3.1 Lessons Learned

OCaml was my first exposure to functional programming, and I absolutely fell in love with it. It took me some time to get used to the functional way of thinking, but I got there, it felt just as natural as imperative or object-oriented programming. I also understand why Prof. Edwards just to have us use a functional language as opposed to Java or C; the step-by-step nature of compiler design (scanner ->

parser -> AST -> semantic analyzer -> generator) lends itself very naturally to functional programming. Also, type inferencing is pretty darn cool.

I did a lot of programming by myself in this project, which had a number of benefits as well as some drawbacks. Prof. Edwards said at the beginning of the project that smaller groups often have it easier than big groups, and you can't get a smaller group than a group of one. It was nice to work at my own pace, and to never really have to comprehend someone else's source code. (It's always easier to write code than to read it.) It feels great to be completely in control! On the other hand, there were plenty of instances when I wanted to communicate with my team members about a problem I was having, but couldn't do it effectively because I had spent hours analyzing my code, while they had only spent minutes. Furthermore, while all of my code works, a lot of it is pretty ugly. If my teammates were more involved in my sections of the project, I feel like they could have helped me spot simpler solutions and prettier optimizations for everything I was writing.

7.3.2 Advice for Future Teams

This is an amazingly fun project. First and foremost, get yourself excited that you're writing your own programming language, and that eventually, you'll get watch it work as if like magic.

There were a number of things I felt our team could have done better. For one, there was hardly ever a point when the team was working on more than one OCaml file at once. The scanner came first, then the parser, AST, generator, core library / java classes, and SAST / semantic analyzer, in that order. I feel like, if more of us were working on the same pieces of the code at once, we would have been better able to communicate bug fixes and improvements as we went along.

Another mistake was the choice of Java as the language to compile to; C would have been way easier. I was actually in favor of C from the start, but because our team was more familiar with Java, we eventually decided on Java within the first couple weeks. But ironically, it eventually turned out that I wrote the generator more-or-less single-handedly! So C would have been easier and more fun for me, but I stuck it out in Java because of that original decision.

Finally, and I'm sure this is the opposite of what many other teams must be saying, but don't be afraid to change the design spec late in the game. I spent about as much time trying to make lists in Java that can hold different classes as I did writing the entire rest of the generator. (Yes, the obvious answer is a LinkedList<Object>, but have fun trying to figure out how to handle casting.) If at some point we had said, "let's just have one class per list", that would have saved me hours of miserable and ugly coding.

7.4 Hebo Yang

7.4.1 Lessons Learned

First of all, this was a comprehensive project covering the motivation, design, implementation, and testing of a language, being developed in a functional programing language OCaml, which was relatively unfamiliar to everyone. Therefore, I think the learning curve was indeed steep although I felt like that I have learned a lot in this process. I also learned that it was really important to not only start early, but also invest more time as well as efforts in the beginning to have a better and more complete understanding of the project as a whole. In other words, being able to see the big picture right at start was crucial. We were very lucky in this sense for having a group member who not only had a lot of programing experiences in various languages, but also did make a huge commitment at the start to lead us towards a great initial design of the language. Though the design was not perfect, we realized in the end that it still helped us avoid many potential detours in the language development process.

Secondly, I learned that a strong leadership was indispensable for team work. There were many times when there was not an apparently correct or best choice and we just had to make a decision. Such leadership not only made the team more efficient, but also simplified the process. Moreover, it was also important to have a leader to oversee the progress, act as a liaison between team members and lastly make sure everyone was doing their job, particularly for a large project like this.

7.4.2 Advice for Future Teams

The first thing I would like to say was that don't be afraid to borrow an idea from a previous group and improve upon it. Looking through the past projects, we found the idea of creating a card game language interesting and promising. Learning from the experiences and more importantly, the mistakes from the previous group made our design more mature to start with at the beginning. Moreover, knowing the deficiency of prior work could give even a stronger motivation to get things right.

As mentioned in the lessons learned subsection as well as suggested by Professor Edwards in the beginning of the class, a strong leadership was really crucial for this project. Therefore, choosing a leader who can get everyone motivated and do their job was even more crucial than choosing a leader who would work on every details on his own. Furthermore, a strong team leader would also help to make sure that the team get started early, which is another important thing in this project.

Appendix A. Project Commit Log from Google Code

Rev	Commit log message	Date	Author
r216	empty	Today (89 minutes ago)	Hebo.Yang@gmail.com
r215	empty	Today (94 minutes ago)	Hebo.Yang@gmail.com
r214	fixed analyzer	Today (112 minutes ago)	Hebo.Yang@gmail.com
r213	remove scanner.mll in trunk	Today (2 hours ago)	Hebo.Yang@gmail.com
r212	remove scanner.mll in trunk	Today (2 hours ago)	Hebo.Yang@gmail.com
r211	remove test.ml in trunk	Today (2 hours ago)	Hebo.Yang@gmail.com
r210	it compiles now!	Today (3 hours ago)	markmicchelli@gmail.com
r209	updated author credit	Today (4 hours ago)	markmicchelli@gmail.com
r208	removed compile and execute, added ryan's name	Today (4 hours ago)	markmicchelli@gmail.com
r207	[No log message]	Today (4 hours ago)	rljones.314
r206	removed reverse funciton b/c unused	Today (4 hours ago)	markmicchelli@gmail.com
r205	fixed score var	Today (4 hours ago)	markmicchelli@gmail.com
r204	[No log message]	Today (4 hours ago)	rljones.314
r203	Substantial progress with semantic analyzer. Compiles most programs we already have. Still have problem where player fields share a namespace with all tmp variables.	Today (9 hours ago)	rljones.314
r202	lots of fixes, player._ should be visibile for all player data fields	Today (9 hours ago)	rljones.314

r201	made card printing prettier with something which probably should have been a core library function to begin with	Yesterday (18 hours ago)	markmicchelli@gmail.com
r200	made card fields Integer and String instead of int and char; made new Card constructor.	Yesterday (18 hours ago)	markmicchelli@gmail.com
r199	every function in the translator now takes an Object and casts; ugly, I know, but necessary for the ele to work.	Yesterday (18 hours ago)	markmicchelli@gmail.com
r198	ele works better, but still not perfect	Yesterday (18 hours ago)	markmicchelli@gmail.com
r197	First draft; set number of players.	Yesterday (19 hours ago)	markmicchelli@gmail.com
r196	added pvar checking stuff	Yesterday (19 hours ago)	rljones.314
r195	Bash Script Upload	Yesterday (20 hours ago)	Hebo.Yang@gmail.com
r194	Bash Script Upload	Yesterday (20 hours ago)	Hebo.Yang@gmail.com
r193	Bash Script	Yesterday (20 hours ago)	Hebo.Yang@gmail.com
r192	[No log message]	Yesterday (20 hours ago)	rljones.314
r191	remove semantic-blockname.cgl	Yesterday (21 hours ago)	Hebo.Yang@gmail.com
r190	update semantic tests	Yesterday (21 hours ago)	Hebo.Yang@gmail.com
r189	convert int in concat to string with intToString	Yesterday (21 hours ago)	rljones.314@gmail.com
r188	updated to latest	Yesterday (21 hours	rljones.314@gmail.com

		ago)	
r187	include semantic_analyzer dependency	Yesterday (22 hours ago)	rljones.314
r186	[No log message]	Dec 16 (41 hours ago)	rljones.314
r185	[No log message]	Dec 16 (41 hours ago)	rljones.314
r184	fixes + added type equality function to handle anytype	Dec 16 (41 hours ago)	rljones.314
r183	fixed semantic checking bugs	Dec 16 (41 hours ago)	rljones.314
r182	[No log message]	Dec 16 (42 hours ago)	rljones.314
r181	[No log message]	Dec 16 (42 hours ago)	rljones.314
r180	Removed "."	Dec 16 (43 hours ago)	Kevin.Henrick1@gmail.com
r179	took out unnecessary "."	Dec 16 (43 hours ago)	Kevin.Henrick1@gmail.com
r178	Draw the First Ace! Ignore the previous one and its folder..	Dec 16 (45 hours ago)	Hebo.Yang@gmail.com
r177	Find Ace!	Dec 16 (45 hours ago)	Hebo.Yang@gmail.com
r176	[No log message]	Dec 16 (45 hours ago)	rljones.314
r175	simpletests	Dec 16 (46 hours ago)	Hebo.Yang@gmail.com
r174	tests	Dec 16 (46 hours ago)	Hebo.Yang@gmail.com
r173	[No log message]	Dec 16 (46 hours ago)	rljones.314
r172	semantictest1	Dec 16 (46 hours	Hebo.Yang@gmail.com

		ago)	
r171	minor ele fix in | Ast.CardExpr	Dec 16 (47 hours ago)	rljones.314
r170	added formals to built-in functions and added assign to binop check	Dec 16 (47 hours ago)	rljones.314
r169	latest sast	Dec 16 (2 days ago)	rljones.314
r168	fixed the card.expr "ele" case	Dec 16 (2 days ago)	kevin.henrick1@gmail.com
r167	unified use of vdecls between sast and ast, and a few other fixes	Dec 16 (2 days ago)	rljones.314
r166	Finding the First Ace game- needs a little debugging.	Dec 16 (2 days ago)	Kevin.Henrick1@gmail.com
r165	cleaned some comments	Dec 16 (2 days ago)	kevin.henrick1@gmail.com
r164	last of the night	Dec 15 (2 days ago)	rljones.314
r163	added semantic check action	Dec 15 (2 days ago)	rljones.314
r162	block order checking	Dec 15 (2 days ago)	rljones.314
r161	dyn-209-2-236-141:CGL ryanjones$ ocamlc -c semantic_analyzer.ml dyn-209-2-236-141:CGL ryanjones$ (it compiles)	Dec 15 (2 days ago)	rljones.314
r160	[No log message]	Dec 15 (2 days ago)	rljones.314
r159	started a new semantic analyzer without sast references, and sending output back to ast	Dec 15 (2 days ago)	kevin.henrick1@gmail.com
r158	sast update	Dec 15 (2 days ago)	rljones.314
r157	fixed MINUS/SUB bug	Dec 15 (2 days ago)	markmicchelli@gmail.com
r156	fixed MINUS/SUB bug	Dec 15 (2 days ago)	markmicchelli@gmail.com

r155	updates	Dec 15 (2 days ago)	rljones.314
r154	major debugging of process statement	Dec 15 (2 days ago)	rljones.314
r153	statement processing debugging	Dec 15 (2 days ago)	rljones.314
r152	major debugging	Dec 15 (2 days ago)	rljones.314
r151	correct expr in sast- still needs some debugging	Dec 14 (3 days ago)	kevin.henrick1@gmail.com
r150	function ordering update	Dec 14 (3 days ago)	rljones.314
r149	Updated expreCheck for List, Player, and Call	Dec 14 (3 days ago)	rljones.314
r148	minor update	Dec 14 (3 days ago)	rljones.314
r147	progress on processStatement	Dec 14 (3 days ago)	rljones.314
r146	minor error fix + commenting	Dec 14 (3 days ago)	rljones.314
r145	[No log message]	Dec 14 (3 days ago)	kevin.henrick1@gmail.com
r144	added comment to the top of the file explaining the input/output of the semantic analyzer	Dec 14 (3 days ago)	kevin.henrick1@gmail.com
r143	added a question regarding unopr in exprCheck	Dec 14 (3 days ago)	kevin.henrick1@gmail.com
r142	[No log message]	Dec 13 (4 days ago)	kevin.henrick1@gmail.co
r141	changed comment for card value error	Dec 13 (4 days ago)	kevin.henrick1@gmail.com
r140	wrote comment for authorship of sast and semantic analyser (Ryan Jones and Kevin Henrick)	Dec 13 (4 days ago)	kevin.henrick1@gmail.com
r139	fixed typo in comment	Dec 13 (4 days ago)	markmicchelli@gmail.com
r138	Fixed syntax errors; added author credit	Dec 13 (4 days ago)	markmicchelli@gmail.com

r137	fixed a little issue where with making the win function	Dec 13 (4 days ago)	markmicchelli@gmail.com
r136	Gives a little more debugging info.	Dec 13 (4 days ago)	markmicchelli@gmail.com
r135	Holds the Card and CGLList classes.	Dec 13 (4 days ago)	markmicchelli@gmail.com
r134	Generator a little cleaner; removed all primitives; put card class in javaclasses.	Dec 13 (4 days ago)	markmicchelli@gmail.com
r133	Shuffle function fixed to handle new lists; eq removed; ugly syntax removed.	Dec 13 (4 days ago)	markmicchelli@gmail.com
r132	Use the -j flag for now; -s not implemented yet; -c and -e not functional.	Dec 13 (4 days ago)	markmicchelli@gmail.com
r131	The proper Makefile should not have the .ml extension.	Dec 13 (4 days ago)	markmicchelli@gmail.com
r130	updated id in expr check	Dec 11 (6 days ago)	rljones.314
r129	draft of binop type checking	Dec 11 (6 days ago)	rljones.314
r128	small cleanup	11-Dec-12	rljones.314
r127	updated the expression in vdecl to be typechecked expr (not Ast.expr as before)	10-Dec-12	kevin.henrick1@gmail.com
r126	minor update	8-Dec-12	rljones.314
r125	some function params/outputs updates	8-Dec-12	rljones.314
r124	Now compiles the java files! Of course, the generator is still broken, but we're on our way.	8-Dec-12	markmicchelli@gmail.com
r123	First draft of executable	8-Dec-12	markmicchelli@gmail.com

r122	High Low game, implemented in half the lines DesCartes used	8-Dec-12	markmicchelli@gmail.com
r121	fixed typo	8-Dec-12	markmicchelli@gmail.com
r120	Red card game from LRM	8-Dec-12	markmicchelli@gmail.com
r119	sast now compiles, but need to go over the different leaves.	6-Dec-12	kevin.henrick1@gmail.com
r118	cleaned up the test.ml to separate the tests into more readable blocks	6-Dec-12	kevin.henrick1@gmail.com
r117	fixed and tested find_function function-working! - Kevin and Ryan	6-Dec-12	kevin.henrick1@gmail.com
r116	find_variable function passed first "find" test-working correctly!	6-Dec-12	kevin.henrick1@gmail.com
r115	Added vdecls and a fdecl to test the find_variable, find_function, and test_scope functions.	6-Dec-12	kevin.henrick1@gmail.com
r114	list checking update in exprCheck	4-Dec-12	rljones.314
r113	added find_function, comments regarding processFdecl function	4-Dec-12	rljones.314
r112	small update	4-Dec-12	kevin.henrick1@gmail.com
r111	fixed method for main, updated exprCheck (CardExpr, Id, Ele)	4-Dec-12	kevin.henrick1@gmail.com
r110	moved main to top	4-Dec-12	rljones.314
r109	updated skeleton with validation comments	4-Dec-12	rljones.314
r108	added more cases for type checking the nodes in	4-Dec-12	kevin.henrick1@gmail.com

	ast.expr		
r107	[No log message]	4-Dec-12	rljones.314
r106	force commit (start of main that returns an Sast)	2-Dec-12	rljones.314@gmail.com
r105	updated the find_variable function. should only have one _ for datatype in List.find that checks scope of the variable by name.	2-Dec-12	kevin.henrick1@gmail.com
r104	[No log message]	2-Dec-12	kevin.henrick1@gmail.com
r103	updates to the sast	2-Dec-12	kevin.henrick1@gmail.com
r102	added type of types	2-Dec-12	kevin.henrick1@gmail.com
r101	replaced instances of expr with expr_detail	2-Dec-12	kevin.henrick1@gmail.com
r100	beginning of sast	2-Dec-12	kevin.henrick1@gmail.com
r99	minor changes in expr function	2-Dec-12	kevin.henrick1@gmail.com
r98	Started the list expr check.	2-Dec-12	kevin.henrick1@gmail.com
r97	Added Ast.StringType to the expr type checking function.	2-Dec-12	kevin.henrick1@gmail.com
r96	[No log message]	1-Dec-12	rljones.314
r95	another symbol table update	1-Dec-12	rljones.314
r94	building symbol table updates	1-Dec-12	rljones.314
r93	"fdecl list syntax update"	30-Nov-12	rljones.314
r92	merged marks fix with our symbol table update	30-Nov-12	rljones.314
r91	Lastest version	30-Nov-12	markmicchelli@gmail.com
r90	made the bdecl list become a block_name_list	30-Nov-12	markmicchelli@gmail.com

r89	got started with type checking	28-Nov-12	rljones.314
r88	simple dfa for block order checking	28-Nov-12	rljones.314
r87	Parser was reversing formal_opt in fdecl	28-Nov-12	markmicchelli@gmail.com
r86	Fixed fdecls to be a lot simpler.	27-Nov-12	markmicchelli@gmail.com
r85	creates 3 java files based on input of stdin	26-Nov-12	markmicchelli@gmail.com
r84	Now makes separate Main, Card, and Player Java classes.	26-Nov-12	markmicchelli@gmail.com
r83	Added constants, but not yet fillConst	25-Nov-12	markmicchelli@gmail.com
r82	string_of_fdecl getting out of control but it works	25-Nov-12	markmicchelli@gmail.com
r81	Fixed pass-by-value/pass-by-reference problem with many, many lines of code.	25-Nov-12	markmicchelli@gmail.com
r80	Fixed the checking of the block order function. Still need to figure out return type, and check multiple TURN{} functions in order from 1,...,n.	24-Nov-12	kevin.henrick1@gmail.com
r79	Totally messed up shuffle implementation :P Correct now, and source cited.	24-Nov-12	markmicchelli@gmail.com
r78	Now with shuffle! Note: these functions rely on java.import.Scanner, java.util.Random, and java.util.LinkedList, but I haven't included import statements.	24-Nov-12	markmicchelli@gmail.com

r77	Added some comments, and make PLAYER create a new class. If we decide against this, the previous version has PLAYER integrated into the main() function.	24-Nov-12	markmicchelli@gmail.com
r76	Everything but shuffle!	24-Nov-12	markmicchelli@gmail.com
r75	[No log message]	24-Nov-12	kevin.henrick1@gmail.com
r74	vdecls are now statements!	24-Nov-12	markmicchelli@gmail.com
r73	vdecls are now statements!	24-Nov-12	markmicchelli@gmail.com
r72	vdecls are now statements!	24-Nov-12	markmicchelli@gmail.com
r71	Changed any mention of Asttest to Ast	24-Nov-12	kevin.henrick1@gmail.com
r70	[No log message]	23-Nov-12	kevin.henrick1@gmail.com
r69	Made a couple corrections	23-Nov-12	kevin.henrick1@gmail.com
r68	Assign now a binop	22-Nov-12	markmicchelli@gmail.com
r67	Assign now a binop	22-Nov-12	markmicchelli@gmail.com
r66	Assign now a binop	22-Nov-12	markmicchelli@gmail.com
r65	Just a start, not even actual core library functions, only needed for operators to work right.	21-Nov-12	markmicchelli@gmail.com
r64	Added my name to the top....	21-Nov-12	markmicchelli@gmail.com
r63	Block decls now display funcs, bvars, and bbodys in right order.	21-Nov-12	markmicchelli@gmail.com
r62	AST now an mli and not ml---please don't use asttest anymore.	21-Nov-12	markmicchelli@gmail.com
r61	Teq and eq now call core library functions. Also, string_of_program works as	21-Nov-12	markmicchelli@gmail.com

	it should. -Mark		
r60	Now handles generator, corelibrary, and ast.mli (as opposed to ast.ml). -Mark	21-Nov-12	markmicchelli@gmail.com
r59	Changed asttest generated by ocamldep from .cmx to .cmi	21-Nov-12	Kevin.Henrick1@gmail.com
r58	Here are the corrected test cases.	21-Nov-12	Kevin.Henrick1@gmail.com
r57	First draft of generator!	21-Nov-12	markmicchelli@gmail.com
r56	Started the functions for converting the Ast.expr into our Java Abstract Syntax Tree (Jast.expr).	20-Nov-12	kevin.henrick1@gmail.com
r55	Deleted comments, and made changes to the symbol table.	20-Nov-12	kevin.henrick1@gmail.com
r54	ELE and YOUR now separate from ID -Mark	19-Nov-12	markmicchelli@gmail.com
r53	ELE and YOUR now separate from ID -Mark	19-Nov-12	markmicchelli@gmail.com
r52	Ele and Your now separate from Id -Mark	19-Nov-12	markmicchelli@gmail.com
r51	Updated the symbol table and function table to have bdecl (block declarations), fdecl (function declarations), and vdecl (variable declarations).	19-Nov-12	kevin.henrick1@gmail.com
r50	added newlines -Mark	19-Nov-12	markmicchelli@gmail.com
r49	Added '.' -Mark	19-Nov-12	markmicchelli@gmail.com
r48	Added '.' -Mark	19-Nov-12	markmicchelli@gmail.com
r47	Added the '.' character for subtypes. -Mark	19-Nov-12	markmicchelli@gmail.com

r46	Added comment about GT and LT in player declarations. -Mark	19-Nov-12	markmicchelli@gmail.com
r45	LIT split into individual types; PLAYER fixed. -Mark	19-Nov-12	markmicchelli@gmail.com
r44	Player fixed; ugly functions in the header fixed; LIT split into individual types. -Mark	19-Nov-12	markmicchelli@gmail.com
r43	Deleted unnecessary int_of_value function. -Mark	19-Nov-12	markmicchelli@gmail.com
r42	Lit broken up into individual types. _to_string functions commented out. -Mark	19-Nov-12	markmicchelli@gmail.com
r41	Makefile works! -Mark	19-Nov-12	markmicchelli@gmail.com
r40	Needs to open the Parser to work! -Mark	18-Nov-12	markmicchelli@gmail.com
r39	Actually handles lists now! Also handles cards and players better by passing them to the parser. -Mark	18-Nov-12	markmicchelli@gmail.com
r38	Updated for 3 new expr types: cardvar, listvar, and playervar -Mark	18-Nov-12	markmicchelli@gmail.com
r37	Parser updated: Lits are now ints, doubles, bools, strings, and SOME cards (w/o vars) Added CardVar, ListVar, and PlayerVar -Mark	18-Nov-12	markmicchelli@gmail.com
r36	Minor fixes.	18-Nov-12	kevin.henrick1@gmail.com
r35	Added more cases within the symbol table.	18-Nov-12	kevin.henrick1@gmail.com
r34	My mistake -- wrong version. This one is good. -Mark	18-Nov-12	markmicchelli@gmail.com

r33	Parser compiles! -Mark	18-Nov-12	markmicchelli@gmail.com
r32	This is the initial cgl file that tests whether the scanner and parser compile together properly.	18-Nov-12	kevin.henrick1@gmail.com
r31	Added the initial functions for creating a symbol table that will be used to check the scope of variables in CGL.	18-Nov-12	kevin.henrick1@gmail.com
r30	Added some more comments to the type declarations.	17-Nov-12	kevin.henrick1@gmail.com
r29	I added a couple comments, and changed a program declaration to simply be a list of blocks. This was changed, since a block list is already composed of function lists and variable lists.	17-Nov-12	kevin.henrick1@gmail.com
r28	Added the comments for the SETUP, PLAYER, TURN, and WIN blocks. Changed the comment for the REMR list operator.	17-Nov-12	kevin.henrick1@gmail.com
r27	This is the beginning of the semantic analyzer. I began a few incomplete functions, and described in comments what future type-checking functions will accomplish once we program them in OCAML.	17-Nov-12	kevin.henrick1@gmail.com
r26	test cases	17-Nov-12	Hebo.Yang@gmail.com
r25	new	17-Nov-12	Hebo.Yang@gmail.com
r24	AST compiles! -Mark	17-Nov-12	markmicchelli@gmail.com
r23	Scanner compiles! Also, list	17-Nov-12	markmicchelli@gmail.com

	is now no longer a literal. -Mark		
r22	MicroC compiler as a start.	17-Nov-12	Kevin.Henrick1@gmail.com
r21	/* Going to start the Makefile over from scratch */	17-Nov-12	Kevin.Henrick1@gmail.com
r20	[No log message]	13-Nov-12	kevin.henrick1@gmail.com
r19	This is a preliminary version of the Makefile.	12-Nov-12	kevin.henrick1@gmail.com
r18	Fixed the double quote escape character. -Mark	12-Nov-12	markmicchelli@gmail.com
r17	Minor fixes on the List Operator associativity declarations.	12-Nov-12	kevin.henrick1@gmail.com
r16	Also, changed the token for the int keyword to INT, not INTLIT. Remember: int is INT 0, 8, 90, -30, 123456 are INTLIT -Mark	12-Nov-12	markmicchelli@gmail.com
r15	Removed helper functions card_of_string and player_of_string, because I'm going to see if we can have OCaml interpret cards and players as strings up until the very last moment they're converted into Java.	12-Nov-12	markmicchelli@gmail.com
r14	Finished making changes as per my comments. -Mark	11-Nov-12	markmicchelli@gmail.com
r13	Doesn't work, but getting there...	11-Nov-12	markmicchelli@gmail.com
r12	[No log message]	11-Nov-12	kevin.henrick1@gmail.com
r11	The scanner has been updated with corrections.	11-Nov-12	kevin.henrick1@gmail.com

r10	I updated the tokens to reflect the changes that Mark would like to make for naming conventions. We need to check the calls of tokens for the literals to make sure that this makes sense.	11-Nov-12	kevin.henrick1@gmail.com
r9	I updated the scanner to reflect the changes that Mark made for the uses of lxm (ie intType, doubleType, boolType, stringType, cardType, listType, etc.).	10-Nov-12	kevin.henrick1@gmail.com
r8	scanner 2 (mark)	10-Nov-12	markmicchelli@gmail.com
r7	[No log message]	10-Nov-12	kevin.henrick1@gmail.com
r6	[No log message]	10-Nov-12	kevin.henrick1@gmail.com
r5	[No log message]	10-Nov-12	kevin.henrick1@gmail.com
r4	I also updated the MicroC scanner to reflect some of the features within cgl.	10-Nov-12	Kevin.Henrick1@gmail.com
r3	I made some minor changes to the parser to reflect the features of our language compared to the one in MicroC.	10-Nov-12	Kevin.Henrick1@gmail.com
r2	test	23-Oct-12	kevin.henrick1@gmail.com
r1	Initial directory structure.	23-Oct-12	---

Appendix B. Complete Code Listing

scanner.mll

```
{
(* Author: Mark Micchelli *)
open Parser

let card_of_string card =
    let int_of_value value =
          if value = 'A' then 14 else
          if value = 'K' then 13 else
          if value = 'Q' then 12 else
          if value = 'J' then 11 else
          if value = '*' then 0 else
              let valueStr = Char.escaped value in
          int_of_string valueStr in
    if String.length card = 3 then (int_of_value card.[1], card.[2])
    else (10, card.[3])
}

(* useful subsections of lits and ids *)
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let punc = ['~' '`' '!' '@' '#' '$' '%' '^' '&' '*' '(' ')' '-' '+'
'='
            ',' '.' '?' '/' '<' '>' ':' ''' ';' '{' '}' '[' ']' '|'
' ']
```

```
let escape = ("\\n" | "\\t" | "\\\"" | "\\\\")
let value = ('2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | "10" |
             'J' | 'Q' | 'K' | 'A' | '*')
let suit = ['C' 'D' 'H' 'S' '*']
let cardExprPt1 = '$' '('
let cardExprPt2 = ')' suit

(* regular expressions for identifiers *)
let identifier = (letter) (letter | digit | '_')*

(* regular expressions for 6 of our 7 main data types;
 * lists are not included because they are recursive and
 * thus not definable by a regular expression *)
let intLit = ('-')? (digit)+
let doubleLit = ('-')? (digit)+ '.' (digit)*
let boolLit = ("true" | "false")
let stringLit = '"' (letter|digit|punc|escape)* '"'
let cardLit = '$' value suit

rule token = parse

['\n' '\r' '\t' ' '] { token lexbuf }
| "^M"     { token lexbuf }
| "/*"     { comment lexbuf }

(* Assignment Operator -- LRM 2.2.1 *)
```

```
| '='       { ASSIGN }

(* Arithmetic Operators -- LRM 2.2.2 *)
| '+'       { PLUS }
| '-'       { SUB }
| '*'       { MULT }
| '/'       { DIV }
| '%'       { MOD }

(* Relational Operators -- LRM 2.2.3
(GT and LT also used for player -- LRM 1.7 *)
| ">="      { GEQ }
| "<="      { LEQ }
| ">"       { GT }
| '<'       { LT }
| "=="      { EQ }
| "!="      { NEQ }
| "==="     { TEQ }
| "!=="     { TNEQ  }

(* Boolean Operators -- LRM 2.2.4 *)
| "!"       { NOT }
| "&&"      { AND }
| "||"      { OR  }

(* String Operator -- LRM 2.2.5 *)
```

```
| '^'       { CONCAT }


(* List Operators -- LRM 2.2.6 *)

| "+>"      { ADDR }

| "<+"      { ADDL }

| "<-"      { REML }

| "->"      { REMR }


(* Punctuators -- LRM 2.3 *)

| ';'       { SEMI }

| '('       { LPAREN }

| ')'       { RPAREN }

| '{'       { LBRACE }

| '}'       { RBRACE }

| ','       { COMMA }


(* Card Stuff -- LRM 1.5 *)

| cardExprPt1 { DOLLAR }

| cardExprPt2 as lxm { SUIT(lxm.[1])  }


(* List Stuff -- LRM 1.6 *)

| '['       { LBRACK }

| ']'       { RBRACK }


(* Player Stuff -- LRM 1.7 *)

| '.'       { DOT }
```

```
(* Preprocessing -- LRM 2.5 *)
| "#include" { INCLUDE }

(* Keywords -- LRM 2.6 *)
| "anytype" { TYPE(Ast.AnytypeType) }
| "bool"    { TYPE(Ast.BoolType) }
| "card"    { TYPE(Ast.CardType) }
| "def"     { DEF }
| "double"  { TYPE(Ast.DoubleType) }
| "ele"     { ELE }
| "else"    { ELSE }
| "foreach" { FOREACH }
| "if"      { IF }
| "int"     { TYPE(Ast.IntType) }
| "list"    { TYPE(Ast.ListType) }
| "player"  { TYPE(Ast.PlayerType) }
| "PLAYER"  { PLAYER }
| "return"  { RETURN }
| "SETUP"   { SETUP }
| "string"  { TYPE(Ast.StringType) }
| "TURN"    { TURN }
| "while"   { WHILE }
| "WIN"     { WIN  }
| "your"    { YOUR }
```

```
(* Literals -- LRM 1 *)

| intLit    as lxm { INT(int_of_string lxm) }

| doubleLit as lxm { DOUBLE(float_of_string lxm) }

| boolLit   as lxm { BOOL(bool_of_string lxm) }

| stringLit as lxm { STRING(lxm) }

| cardLit   as lxm { CARD(card_of_string lxm) }


(* Identifiers -- LRM 2.1 *)

| identifier as lxm { ID(lxm) }


| eof { EOF }

| _ as char { raise (Failure("illegal character: " ^ Char.escaped
char)) }


and comment = parse

  "*/" { token lexbuf }

| _    { comment lexbuf }
```

parser.mly

```
%{

(* Author: Mark Micchelli *)

open Ast

%}
```

```
%token ASSIGN

%token PLUS SUB MULT DIV MOD

%token GEQ LEQ GT LT EQ NEQ TEQ TNEQ

%token AND OR NOT

%token CONCAT

%token ADDR ADDL REMR REML

%token DOT

%token SEMI COMMA LPAREN RPAREN LBRACE RBRACE

%token DOLLAR LBRACK RBRACK

%token INCLUDE

%token DEF ELSE ELE FOREACH IF PLAYER RETURN SETUP TURN WHILE WIN
YOUR

%token <Ast.datatype> TYPE

%token <int> INT

%token <float> DOUBLE

%token <bool> BOOL

%token <string> STRING

%token <int * char> CARD

%token <char> SUIT

%token <string> ID

%token EOF


%nonassoc NOELSE

%nonassoc ELSE

%right ASSIGN

%right ADDL

%left ADDR
```

```
%right REML

%left REMR

%left OR

%left AND

%left TEQ TNEQ

%left EQ NEQ

%left LT GT LEQ GEQ

%left CONCAT

%left PLUS SUB

%left MULT DIV MOD

%right NOT

%right DOT


%start program

%type <Ast.program> program


%%


/* program is a bdecl list */

program:

    bdecl_list { List.rev $1 }


bdecl_list:

      /* nothing */ { [] }

    | bdecl_list bdecl {$2 :: $1 }
```

```
bdecl:
      PLAYER LBRACE stmt_list RBRACE
        { { bname = "PLAYER"; bid = -1;
        funcs = []; bbody = List.rev $3; } }
    | SETUP LBRACE fdecl_list stmt_list RBRACE
        { { bname = "SETUP"; bid = -1; funcs = List.rev $3;
        bbody = List.rev $4; } }
    | TURN INT LBRACE fdecl_list stmt_list RBRACE
        { { bname = "TURN"; bid = $2; funcs = List.rev $4;
        bbody = List.rev $5; } }
    | WIN LBRACE fdecl_list stmt_list RBRACE
        { { bname = "WIN"; bid = -1; funcs = List.rev $3;
        bbody = List.rev $4; } }

fdecl_list:
      /* nothing */    { [] }
    | fdecl_list fdecl { $2 :: $1 }

fdecl:
    DEF TYPE ID LPAREN formal_opt RPAREN LBRACE stmt_list RBRACE
    { { fdt = $2;
      fname = $3;
      formals = $5;
      fbody = List.rev $8; }
    }
```

```
formal_opt:
      /* nothing */ { [] }
    | formal_list { List.rev $1 }

formal_list:
      formal { [$1] }
    | formal_list COMMA formal { ($3 :: $1) }

formal:
    TYPE ID { { pdt = $1; pname = $2; } }

vdecl:
    TYPE ID ASSIGN expr SEMI
    { { vdt = $1;
        vname = $2;
        value = $4; }
    }

stmt_list:
      /* nothing */  { [] }
    | stmt_list stmt { $2 :: $1 }

stmt:
      expr SEMI { Expr($1) }
    | RETURN expr SEMI { Return($2) }
    | vdecl { Vdecl($1) }
    | LBRACE stmt_list RBRACE { Block(List.rev $2) }

    | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
Block([])) }

    | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }

    | FOREACH LPAREN expr RPAREN stmt { Foreach($3, $5) }

    | WHILE LPAREN expr RPAREN stmt { While($3, $5) }


expr_opt:

      /* nothing */ { [] }

    | expr_list { List.rev $1 }


expr_list:

      expr { [$1] }

    | expr_list COMMA expr { $3 :: $1 }


expr:

      INT               { Int($1) }

    | DOUBLE           { Double($1) }

    | BOOL             { Bool($1) }

    | STRING           { String($1) }

    | CARD             { Card(fst $1, snd $1) }

    | DOLLAR expr SUIT { CardExpr($2, $3) }

    | LBRACK expr_opt RBRACK { List($2) }

    | LT expr COMMA expr GT  { Player($2, $4) }

    | ID               { Id($1) }

    | ELE              { Ele }

    | YOUR             { Your }
    | expr ASSIGN expr { Binop($1, Assign, $3) }
    | expr PLUS   expr { Binop($1, Plus,   $3) }
    | expr SUB    expr { Binop($1, Sub,    $3) }
    | expr MULT   expr { Binop($1, Mult,   $3) }
    | expr DIV    expr { Binop($1, Div,    $3) }
    | expr MOD    expr { Binop($1, Mod,    $3) }
    | expr EQ     expr { Binop($1, Eq,     $3) }
    | expr NEQ    expr { Binop($1, Neq,    $3) }
    | expr TEQ    expr { Binop($1, Teq,    $3) }
    | expr TNEQ   expr { Binop($1, Tneq,   $3) }
    | expr LT     expr { Binop($1, Lt,     $3) }
    | expr LEQ    expr { Binop($1, Leq,    $3) }
    | expr GT     expr { Binop($1, Gt,     $3) }
    | expr GEQ    expr { Binop($1, Geq,    $3) }
    | expr AND    expr { Binop($1, And,    $3) }
    | expr OR     expr { Binop($1, Or,     $3) }
    | expr CONCAT expr { Binop($1, Concat, $3) }
    | expr ADDL   expr { Binop($1, Addl,   $3) }
    | expr ADDR   expr { Binop($1, Addr,   $3) }
    | expr DOT    expr { Binop($1, Dot,    $3) }
    | NOT  expr        { Unopl(Not,  $2) }
    | REML expr        { Unopl(Reml, $2) }
    | expr REMR        { Unopr($1, Remr) }
    | ID LPAREN expr_opt RPAREN { Call($1, $3) }
    | LPAREN expr RPAREN { $2 }
```

ast.mli

```
(* Author: Mark Micchelli *)

type datatype = IntType | DoubleType | BoolType | StringType | 
CardType |
    ListType | PlayerType | AnytypeType

type binop = Assign | Plus | Sub | Mult | Div | Mod | Eq | Neq | 
Teq |
    Tneq | Lt | Leq | Gt | Geq | And | Or | Concat | Addl | Addr | 
Dot
type unopl = Not | Reml
type unopr = Remr

type expr =
    Int of int
    | Double of float
    | Bool of bool
    | String of string
    | Card of int * char
    | CardExpr of expr * char
    | List of expr list
    | Player of expr * expr
    | Id of string
    | Ele
    | Your

    | Binop of expr * binop * expr

    | Unopl of unopl * expr

    | Unopr of expr * unopr

    | Call of string * expr list

    | Noexpr



type vdecl =

{

    vdt : datatype;

    vname : string;

    value : expr;

}


type stmt =

      Block of stmt list

    | Expr of expr

    | Vdecl of vdecl

    | Return of expr

    | If of expr * stmt * stmt

    | Foreach of expr * stmt

    | While of expr * stmt


type formal =

{
    pdt : datatype;
    pname : string;
}

type fdecl =
{
    fdt : datatype;
    fname : string;
    formals : formal list;
    fbody : stmt list;
}

type bdecl =
{
    bname : string;
    bid : int;
    funcs : fdecl list;
    bbody : stmt list;
}

type program = bdecl list
```

semantic_analyzer.ml

```
(* Author Ryan Jones and Kevin Henrick *)

(* The semantic analyzer takes in an AST and type checks each node
of the tree,*)

(* and returns the correctly checked node to the semantically
analyzed abstract*)

(* syntax tree (SAST). *)


open Ast

open Corelibrary


(* module StringMap = Map.Make(String) *)


(* Creating the symbol_table *)

type symbol_table = {

  parent : symbol_table option;

  functions : Ast.fdecl list;

  variables : Ast.vdecl list;

}


let typeEq t1 t2 = match t1,t2 with

| AnytypeType, _ -> true

| _, AnytypeType -> true

| _, _ -> if (t1 = t2)

            then true

                        else false


let string_of_datatype = function
```

```
      IntType -> "IntType"
    | DoubleType -> "DoubleType"
    | BoolType -> "BoolType"
    | StringType -> "StringType"
    | CardType -> "CardType"
    | ListType -> "ListType"
    | PlayerType -> "PlayerType"
    | AnytypeType -> "AnytypeType"

let string_of_expr = function
  | _ -> "(not implemented yet)"

let string_of_binop = function
  | Assign -> "Assign"
  | Plus -> "Plus"
  | Sub -> "Sub"
  | Mult -> "Mult"
  | Div -> "Div"
  | Mod -> "Mod"
  | Eq -> "Eq"
  | Neq -> "Neq"
  | Teq -> "Teq"
  | Tneq -> "Tneq"
  | Lt -> "Lt"
  | Leq -> "Leq"
  | Gt -> "Gt"
  | Geq -> "Geq"

  | And -> "And"

  | Or -> "Or"

  | Concat -> "Concat"

  | Addl -> "Addl"

  | Addr -> "Addr"

  | Dot -> "Dot"


(* DEFAULT fdt is Ast.IntType *)


(* Core functions *)




let f1 = {pdt = Ast.IntType; pname = "int"}

let intToString =       { fdt = Ast.StringType; fname =
"intToString";          formals = [f1]; fbody = []; }


let f1 = {pdt = Ast.DoubleType; pname = "double"}

let doubleToString =    { fdt = Ast.StringType; fname =
"doubleToString"; formals = [f1]; fbody = []; }


let f1 = {pdt = Ast.StringType; pname = "string"}

let stringToInt =            { fdt = Ast.IntType;    fname =
"stringToInt";          formals = [f1]; fbody = []; }


let f1 = {pdt = Ast.StringType; pname = "double"}

let stringToDouble =    { fdt = Ast.DoubleType; fname =
"stringToDouble";  formals = [f1]; fbody = []; }
```

```
let f1 = {pdt = Ast.CardType; pname = "valueCard"}

let value =                             { fdt = Ast.IntType;
fname = "value";                        formals = [f1]; fbody =
[]; }



let scan =                              { fdt = Ast.StringType;
fname = "scan";                    formals = [];   fbody = []; }



let f1 = {pdt = Ast.StringType; pname = "printString"}

let print =                             { fdt = Ast.IntType;
fname = "print";                        formals = [f1]; fbody =
[]; }



let f1 = {pdt = Ast.CardType; pname = "suitString"}

let suit =                              { fdt = Ast.StringType;
fname = "suit";                              formals = [f1]; fbody
= []; }



let random =                       { fdt = Ast.ListType;   fname =
"random";                          formals = []; fbody = []; }



let f1 = {pdt = Ast.ListType; pname = "deck"}

let shuffle =                      { fdt = Ast.ListType;   fname =
"shuffle";                    formals = [f1]; fbody = []; }



let f1 = {pdt = Ast.PlayerType; pname = "player"}

let turn =                              { fdt = Ast.IntType;
fname = "turn";                              formals = [f1]; fbody
= []; }
```

```
let win =                                          { fdt = Ast.IntType;
fname = "win";                                     formals = [];    fbody
= []; }




(* Core variables *)

let standard = { vdt = Ast.ListType;    vname = "STANDARD";  value
= Ast.Noexpr }

let nemo     = { vdt = Ast.PlayerType;  vname = "NEMO";      value
= Ast.Noexpr }


(* Pre-defined player data fields *)

let name     = { vdt = Ast.StringType;  vname = "$PVARname";
value = Ast.Noexpr }

let turnID   = { vdt = Ast.IntType;     vname = "$PVARturnID";
value = Ast.Noexpr }


let global_scope = {

  parent = None;

  variables = [ standard; nemo; name; turnID];

  functions = [ intToString; doubleToString;

            stringToInt; stringToDouble;

            value;       suit;

                               scan;        print;

                               random;      shuffle;

                               turn;        win ];

}
```

```
(* Finding the variable in the scope by its name*)

let rec find_variable scope name =

  try

    List.find (fun {vdt=_; vname=s; value=_} -> s = name)
scope.variables

  with Not_found ->

         match scope.parent with

    | Some(parent) -> find_variable parent name

    | _ -> let pTrim st = ( if (String.length st) > 4

                                then (String.sub st 0 5)


else st                   )

                          in

                   if ( (pTrim name) <> "$PVAR")

                 then find_variable scope ("$PVAR"^name)

                          else

                                (* raise (Failure("Var not found in
scope (TMP message)")) *)

                                let outName = if (String.length
name) > 4

                                                then


        (String.sub name 5 (String.length name-5))


else name

                                in

             raise (Failure("Variable "^outName^" not found in
scope"))
```

```
            (* find-string starts with PVAR but still not found,
i.e. not a player var *)


(*

       let _ = (

            let rec getGlobalScope scope = match scope.parent with

          | None -> scope

          | Some(parent) -> (getGlobalScope parent)

            |

    let build_string tmpString nextString = tmpString^"
\n"^nextString in

    let func_names_string = List.fold_left build_string("")
(List.map (fun {fdt=_; fname=n; formals=_; fbody=_} -> n )
(getGlobalScope scope).functions) in

    let num_funcs = List.length (getGlobalScope scope).functions in



    let var_names_string = List.fold_left build_string("")
(List.map (fun {vdt=_; vname=n; value=_} -> n ) (getGlobalScope
scope).variables) in

    let num_vars = List.length (getGlobalScope scope).functions in

    raise(Failure("At end, vars found were "^((string_of_int
num_vars)^var_names_string)

               ^" \n\nand funcs found were "^((string_of_int
num_funcs)^func_names_string) ))

    ) in

*)




(* check if a function is present in global scope *)

let rec find_function scope name =
  let rec getGlobalScope scope = match scope.parent with

        | None -> scope

        | Some(parent) -> (getGlobalScope parent)

  in

  try

        List.find (fun {fdt=_; fname=s; formals=_; fbody=_} -> s =
name) (getGlobalScope scope).functions

  with Not_found ->

        let build_string tmpString nextString = tmpString^"
\n"^nextString in

        let func_names_string = List.fold_left build_string("")
(List.map (fun {fdt=_; fname=n; formals=_; fbody=_} -> n )
(getGlobalScope scope).functions) in

        let num_funcs = List.length (getGlobalScope
scope).functions in

        raise(Failure("Function "^name^" not found in global scope,
funcs found were "^(string_of_int num_funcs)^func_names_string))




(* Used identifiers must be defined: use find_variable function OR
find_func function *)

(* Identifier references must be variables or functions  *)

(* Function calls must refer to functions: use find_func function *)

(* Left side of assign binop must be existing variable of same type
as right side. *)



let rec exprCheck scope = function
```

```
  (* pass literals right through, already validated in the scanner
*)
  | Ast.Int i       -> Sast.Int(i),    Ast.IntType
  | Ast.Double d    -> Sast.Double(d), Ast.DoubleType
  | Ast.Bool b      -> Sast.Bool(b),   Ast.BoolType
  | Ast.String s    -> Sast.String(s), Ast.StringType
  | Ast.Card (i,c)  -> Sast.Card(i,c), Ast.CardType



  | Ast.CardExpr (e,c)  ->
             let is_valid expr = match expr with
            | Ast.Int i ->  if ( i>1 && i<15 ) then
                                           true (* value is valid *)
                                       else

  raise (Failure ("card integer value must be between 2 and 14
(inclusive), but found "^(string_of_int i)^".")     )
            | Ast.Id  id -> let curVdecl = find_variable scope id
in
                                       if( typeEq curVdecl.vdt
Ast.IntType)  (* id must refer to IntType in CardExpr*)

then true
                                                              else
raise (Failure ("Id in CardExpr must be type Int, found type " ^
string_of_datatype curVdecl.vdt))
                 | Ast.Ele   ->  let eleType = (find_variable
scope "ele").vdt (* lookup what ele currently refers to in local
symbol table *)
                                       in
```

```
                                                                      if
( typeEq eleType Ast.IntType)

then true
                                                                      else
raise (Failure ("Ele must refer to an element of type Int, found
type " ^ (string_of_datatype eleType)) )




                   | _         ->  raise (Failure ("card expression
is wrong type, found "^string_of_expr expr^" ."))
      in
             if ( is_valid e ) then
        Sast.CardExpr( exprCheck scope e, c), Ast.CardType  (*
SUCCESS RETURN VALUE *)
          else raise (Failure ("card expression value or type is
wrong"))


  | Ast.List (elements) -> Sast.List( List.map (exprCheck scope)
elements ), Ast.ListType


  | Ast.Player (e1, e2) ->
            let is_valid_player expr1 expr2 = match snd(exprCheck
scope e1), snd(exprCheck scope e2) with
                   | Ast.StringType, Ast.IntType -> true
                   | _ , _ -> false
                   in
                   if is_valid_player e1 e2
                     then Sast.Player(exprCheck scope e1 , exprCheck
scope e2), Ast.PlayerType
```

```
        else

                        raise(Failure("Invalid player name or turn
id, e1 type: "^(string_of_datatype (snd(exprCheck scope e1)) )^" e2
type:"^(string_of_datatype (snd(exprCheck scope e2))) ))


   | Ast.Id(s) -> (* use find_variable and find_function *)

     let newVdecl = find_variable scope s in

        Sast.Id(newVdecl.vname), newVdecl.vdt   (* look for it as a
variable with find_variable *)


   | Ast.Ele -> Sast.Ele, (find_variable scope "ele").vdt

   | Ast.Your ->

          if( (find_variable scope "your").vname = "your")

                then Sast.Your, Ast.PlayerType

          else

                raise(Failure("Your referenced outside of turn
block"))

                     (* MIGHT NOT BE REACHED EVER *)


   | Ast.Binop (expr1, op, expr2) ->  (* taken from MicroC Type
Checker*)

        (

        (* The types of operands for unary and binary operators
must be consistent. *)

                     let e1 = exprCheck scope expr1 in


                     (* if only operator is dot, than add look for
player.___ !!!!!!!!!!!!!*)

                     let e2 = exprCheck scope expr2 in
        let _, t1 = e1 (* Get the type of each child *) in

        let _, t2 = e2 in

                    match t1, op, t2 with

                    | (AnytypeType | BoolType), (And| Or),
(AnytypeType | BoolType) ->  (* Bool *)

                          Sast.Binop(e1, op, e2), Ast.BoolType



                    | (AnytypeType | IntType), Mod, (AnytypeType |
IntType) ->  (* Ints mod *)

                          Sast.Binop(e1, op, e2), Ast.IntType



                    | (AnytypeType | IntType | DoubleType), (Lt | Leq
| Gt | Geq), (AnytypeType | IntType| DoubleType) ->  (* Int
operations *)

                          Sast.Binop(e1, op, e2), Ast.BoolType



                    | (AnytypeType | IntType), (Plus | Sub | Mult |
Div), (AnytypeType | IntType) ->  (* Ints or Floats (with above
cases excluded) *)

                          Sast.Binop(e1, op, e2), Ast.IntType



                    | (AnytypeType | IntType | DoubleType), (Plus |
Sub | Mult | Div), (AnytypeType | IntType | DoubleType) ->  (* Ints
or Floats *)

                          Sast.Binop(e1, op, e2), Ast.DoubleType



                   | _, (Eq | Neq | Teq | Tneq), _  ->   (* Anything?
*)

                          Sast.Binop(e1, op, e2), Ast.BoolType
                   | (StringType | AnytypeType), Concat, (StringType
| AnytypeType) ->  (* Strings *)    (* DOUBLE CHECK ANYTYPE ALLOWED*)

                     Sast.Binop(e1, op, e2), Ast.StringType



                   | (ListType|AnytypeType), Addl, (IntType |
CardType | DoubleType | StringType | BoolType | ListType |
PlayerType | AnytypeType) -> (* expr lists *)

                     Sast.Binop(e1, op, e2), Ast.ListType



                   | (IntType | CardType | DoubleType | StringType |
BoolType | ListType | PlayerType | AnytypeType), Addr,
(ListType|AnytypeType) -> (* expr lists *)

                     Sast.Binop(e1, op, e2), Ast.ListType



                  | PlayerType, Dot, t  -> if (typeEq t AnytypeType)

                                                  then Sast.Binop(e1, op,
e2), Ast.AnytypeType(* left expr is expr * expr or Id, right expr
is Id *)


              else raise(Failure("player dotted with wrong type
(should never throw compile-time error?)"))



                   | _, Assign, _ -> if (typeEq t1 t2)

                       then Sast.Binop(e1, op, e2), Ast.AnytypeType
(* !?! DANGEROUS !?! --> FIX RETURN TYPE *)

                        else raise(Failure("Operand types must
match for Assign operator"))



                   | tA, op, tB -> raise(Failure("Binop "^
(string_of_binop op) ^" has improper operands, found "^
(string_of_datatype tA) ^", "^ (string_of_datatype tB) ^ "\n"))

                  )
```

```
  | Ast.Unopl (op, expr) ->
           (
               let e1 = exprCheck scope expr in
         let _, t1 = e1 in (* Get the type of each child *)
               match op, t1 with
           | Not, BoolType -> Sast.Unopl(op, e1), Ast.BoolType (*
Bool *)
           | Reml, ListType -> Sast.Unopl(op, e1), Ast.AnytypeType
         | _,_ -> raise(Failure("Invalid operand ("^string_of_datatype
t1^") for left-associative unary operator"))
               )

  | Ast.Unopr (expr, op) ->
           (
           let e1 = exprCheck scope expr in
         let _, t1 = e1 in (* Get the type of each child *)
               match t1, op with
           | ListType, Remr -> Sast.Unopr(e1, op), Ast.ListType
           | _ -> raise(Failure("Invalid operand
("^string_of_datatype t1^") for right-associative unary operator"))
               )

  | Ast.Call (string, exprList) ->
          (
          let funcFound = find_function scope string in
          let formalTypes = List.map (fun {pdt=s; pname=_} -> s)
funcFound.formals in
```

```
        let checkedExprs = List.map (exprCheck scope) exprList in

        let checkedTypes = List.map snd(checkedExprs) in


        (* check each type from checked list against fdecl param
types in scope's function list *)

        let rec typesMatch list1 list2 = match list1,list2 with

        | [], [] -> true

        | [], _ -> raise(Failure(" found parameters when expecting
none "))

        | _ , [] -> raise(Failure(" found no parameters when
expecting parameters"))

        | _ , _ ->

              try

          ( typeEq (List.hd(list1)) (List.hd(list2)) ) &&
typesMatch (List.tl(list1)) (List.tl(list2))

              with Failure("hd") ->

                   raise(Failure(" trying List.hd on [] in
exprCheck:Ast.Call"))

        in

        if (typesMatch formalTypes checkedTypes)

          then Sast.Call (string , checkedExprs) , funcFound.fdt

    else

              raise(Failure("Arguments for function: "^ string ^" do
not match function signature"))

        )


  | Ast.Noexpr -> Sast.Noexpr, Ast.AnytypeType
```

```
(* adds player data fields to global scope so that player.x is
visible in any turn block. *)

(* INS: curScope:symbol_table , prevScope:symbol_table ,
scopeStack:(symbol_table list) , vd:Vdecl  *)

(* OUTS: out:symbol_table*)


let empty_table =   (* empty table will be a flag to indicate the
first function call *)

(*Some(symbol_table) or None  (i.e. symbol_table Option *)

                   { parent = None;

                                                 functions = [];

                                                    variables =
[]; }

let rec addPvarGlobal curScope prevScope scopeStack vd = match
curScope, prevScope, scopeStack with


| cur, prev, [] ->  (* empty table will be a flag to indicate the
first function call *)

                    (* let _ = print_string " cur,prev,[] called in
addPvarGlobal \n" in *)

                                                let symNonOpt s =
match s with

                                                    |  Some(sym_tab)
-> sym_tab

                                                        | None ->
( let _ = print_string("hack should work around this (in
addPvarGlobal)") in empty_table)

                                                   in

                                                   let prevIsEmpty
prev = match prev with
```

```
                                                  | Some(sym_tab)
-> (sym_tab = empty_table)
                                                  | _ -> false
                                                  in
                    (* if (prev = empty_table) (*ascending*) *)
                                                  if (prevIsEmpty
prev) (*ascending*)
                    then
                                                           (*
let _ = print_string("ascending (stack+) \n") in *)

                                                           let
curParentNone cur = match cur with
                                                           |
Some(sym_tab) -> (sym_tab.parent = None)
                                                          | _ ->
false
                                                           in
                                                           (* if
( cur.parent = None) (*started right in the global*) *)
                                                           if
( curParentNone cur) (*started right in the global*)
                        then let newVd =
                                                      { vdt =
vd.vdt;
                                                       vname =
"$PVAR"^vd.vname;

value = vd.value; }
                                                            in
                                                         let
newGlobal =
                          { parent = None;
                                                            functions =
(symNonOpt cur).functions;

variables = newVd::((symNonOpt cur).variables) } in

Some(newGlobal)
                                                              else
(addPvarGlobal (symNonOpt cur).parent cur (cur::[]) vd)
                                                        else (*
cur.parent = prev , i.e. descending *)
                                                          (* let _ =
print_string "final descend \n" in *)
                                                          cur  (* FINAL
RETURN VALUE *)


| cur, prev, h::t ->
                    let _ = print_string " cur,prev,h::t called in
addPvarGlobal \n" in
                    let symNonOpt s = match s with
                                                         |  Some(sym_tab)
-> sym_tab
                                                             | None ->
( let _ = print_string("hack should work around this (in
addPvarGlobal)\n") in empty_table)
                                                        in
                     if (symNonOpt prev).parent = cur (*ascending*)
                       then
                                                                  let
_ = print_string( "ascending (stack+) \n") in
                           if (symNonOpt cur).parent = None  then
(*middle state, prev is global scope*)
```

```
let newVd =

{ vdt = vd.vdt;

vname = "$PVAR"^vd.vname;

value = vd.value; } in
                           let newGlobal =
                           { parent = None;
                                                          functions
= (symNonOpt cur).functions;

variables = newVd::( (symNonOpt cur).variables); } in
                                                        let newCur =
{ parent = Some newGlobal;

variables = (symNonOpt h).variables;

                             functions = (symNonOpt h).functions; }
                                                        in
addPvarGlobal (Some newCur) (Some newGlobal) t vd
                         else (addPvarGlobal (symNonOpt cur).parent
cur (cur::(h::t)) vd)   (*walking state*)
                                                      else
                                                           if
(symNonOpt cur).parent = prev (*descending*)
                                                               then
                                                               let _
= print_string ("descending (stack-) \n") in

  addPvarGlobal h cur t vd
```

```
                                                          else

                                                              let _
= print_string "shouldn't be reached \n" in None



(* The below function iteratively checks each statement in CGL by
folding over an intermediate scope/statement list tuple. *)


let rec processStatement (scope,stmtList) stmt =

  let newScope = {

    parent =  Some scope;  (* set parent to newglobalscope
parameter *)

    functions = [];

    variables = [];

  } in

  match stmt with

  | Expr(e1)  -> scope, Sast.Expr( exprCheck scope e1 )::stmtList

  | Vdecl(vd) ->

        let exprType = snd(exprCheck scope vd.value) in

        if (typeEq vd.vdt exprType)  (* TODO: Refactor and make
sequential *)

      then if (try (find_variable scope vd.vname).vname = vd.vname
with Failure(msg) -> false)  (* Failure is good because no
duplicate declaration *)

                  then raise(Failure("Duplicate variable
declaration for var:"^vd.vname))

             else

                  let updScope =

            { parent = scope.parent;
            functions = scope.functions;

                   variables = vd::(scope.variables) } in



                        let updScopePvar = (

                             let emptyScope = { parent = None;


functions = [];


variables = []; }

                                   (* serves as flag for
addPvarGlobal *) in



                              if (try let _ = find_variable scope
"IS_IN_PLAYER" in true with Failure(msg) -> false )  (* if current
block is player block*)

                            then

                                   (* let _ = print_string("is
IS_IN_PLAYER \n") in *)

                                   match ( addPvarGlobal (Some
updScope) (Some emptyScope) [] vd ) with   (* add player variable
to global scope as PVARname *)

                                   | Some(sym_tab) -> sym_tab

                                   | None -> let _ = print_string
"should never be reached (in process:stmt)" in emptyScope

                          else

                                   (* let _ = print_string("not
IS_IN_PLAYER \n") in *)

                                   updScope

                             )
```

```
                       in





                 let retVdecl = fst( vd, exprType ) in

            updScopePvar, Sast.Vdecl(retVdecl)::stmtList
(* !!!!!!! until updScopePvar WORKS *)

      else raise(Failure("Assignment variable
type:"^(string_of_datatype vd.vdt)^

                          " does not match expression
type:"^(string_of_datatype exprType)))


   | Block(stmts) -> scope, Sast.Block( snd( List.fold_left
processStatement(newScope,[]) stmts ) )::stmtList


   | If(e, s1, s2) ->

        let ec = exprCheck scope e in

     let _, t1 = ec in

        if (typeEq t1 Ast.BoolType) then

            let newS1, s1c = (processStatement(newScope, []) s1)
in

            let _, s2c = (processStatement (newScope, []) s2) in

            try

            newS1, Sast.If( ec, List.hd(s1c),
List.hd(s2c) )::stmtList

            with Failure("hd") ->

                raise(Failure(" trying List.hd on [] in
processStatement:If"))

        else raise(Failure("predicate of If must be boolean
expression, found "^(string_of_datatype t1)))
```

```
  | While(e, s) ->
    let ec = exprCheck scope e in
    let _, t1 = ec in
        if (typeEq t1 Ast.BoolType) then
          let newS1, sc = (processStatement(newScope, []) s) in
              try
          newS1, Sast.While( ec, List.hd(sc) )::stmtList
              with Failure("hd") ->
                raise(Failure(" trying List.hd on [] in
processStatement:While"))
        else raise(Failure("predicate of While must be boolean
expression, found "^(string_of_datatype t1)))


  | Foreach(e, s) -> (* Includes ELE *)
    let ec = exprCheck scope e in
    let _, t1 = ec in
        let ele = {
              vdt = Ast.AnytypeType; (* evaluate e and get type of
first element, b/c e should be a list *)
              vname = "ele";
              value = Ast.Ele;  (* evaluate e and get value of first
element, b/c e should be a list *)
        } in
        let eleScope =  (* eleScope extends newScope to include Ele
*)
    {
      parent = newScope.parent;  (* set parent to newglobalscope
parameter *)
```

```
      functions = newScope.functions;
      variables = ele::newScope.variables;
    } in
       if (typeEq t1 Ast.ListType) then
         let newS1, sc = (processStatement(eleScope, []) s) in
             try
         newS1, Sast.Foreach( ec, List.hd(sc))::stmtList
         with Failure("hd") ->
              raise(Failure(" trying List.hd on [] in
processStatement:Foreach"))
       else raise(Failure("predicate of Foreach must be ListType
expression, found "^(string_of_datatype t1)))


  | Return(e) ->
       let curReturnType = (find_variable scope "return").vdt  (*
store return type of current function in scope *)
       in
       let ec = exprCheck scope e in
    let _, t1 = ec in
       if (typeEq t1 curReturnType )
         then scope, Sast.Return(ec)::stmtList
       else raise(Failure("Return type of function body doesn't
match return type of function signature,
         found"^(string_of_datatype t1)^", expected
"^(string_of_datatype curReturnType)))
```

```
let processFdecl curScope fdecl =  match fdecl.fbody with
  | [] -> raise(Failure("Empty functions not allowed"))
  | x ->
        let retVdecl = {
          vdt = fdecl.fdt;
              vname = "return";
              value = Ast.Noexpr;
        } in
        let formalToVdecl = function
              | frml -> { vdt = frml.pdt;
                              vname = frml.pname;
                                                value = Ast.Noexpr }
        in
    let retScope = {
              parent = curScope.parent;
              functions = curScope.functions;
              variables = (List.map formalToVdecl
(fdecl.formals) )@(retVdecl::curScope.variables);
    } in
        let checkedFdecl =
        {
          fdt = fdecl.fdt;
          fname = fdecl.fname;
          formals = fdecl.formals;
          fbody = fst(x, (List.fold_left processStatement(retScope,
[]) fdecl.fbody ));   (* UGLY HACK (x) *)
        } in
        checkedFdecl




let processFdecls bname fdecls scope = match bname, fdecls with
| "SETUP", fdecl_list ->
  (
    (* LOAD all funcs in list into symbol table (b/c all funcs
should be able to "find" i.e. call each other) *)
      let addFunc scope fdecl =
                  { parent = scope.parent;
                    functions = fdecl::scope.functions;
                    variables = scope.variables }
             in
             let newGlobal = List.fold_left addFunc(scope) fdecls
(* new scope containing all funcs in setup *)
         in
      let subScope = {
              parent =  Some newGlobal;  (* set parent to
newglobalscope parameter *)
              functions = [];
              variables = [];
         }
         in
         (* MAP through each function and check *)
         (List.map (processFdecl subScope) fdecl_list), newGlobal
(* newFdecls, newScope *)
```

```
   )

| _ , [] ->   [], scope  (* Non-"SETUP" bname should have empty
fdecls list *)

| _ , _  ->   raise(Failure("Functions can only be declared in
SETUP (at beginning)"))







let processBdecl (scope, checkedBdecls) bdecl =

        let funcsAndScope = processFdecls bdecl.bname bdecl.funcs
scope in

    (* IN: bname, funcs list  --->>  OUT: func list, updated symbol
table  *)

        let updScope = snd(funcsAndScope) in



        let yourVdecl = {vdt = Ast.PlayerType; vname = "your";
value = Ast.Noexpr; } in  (* dummy "value" *)

        let updScopeYour = {

          parent = updScope.parent;

              functions = updScope.functions;

              variables = yourVdecl::updScope.variables;

        } in



        (* let _ = print_string ("\n\nbdecl.name is
"^(bdecl.bname)^"\n") in *)

    if bdecl.bname = "TURN"

          then
          let bbodyAndScope = List.fold_left
processStatement(updScopeYour,[]) bdecl.bbody

         in  (* IN: stmtList, scope, bbody   -->>   OUT: stmt list,
updated symbol table *)

          let returnBdecl = {

             bname = bdecl.bname;

             bid =   bdecl.bid;

             funcs = fst funcsAndScope;

             bbody = fst(bdecl.bbody, snd bbodyAndScope);   (* UGLY
HACK (ast instead of sast) *)

          }  in

          fst(bbodyAndScope), returnBdecl::checkedBdecls  (*
intermediate scope and bdecl list *)




        else if (bdecl.bname = "PLAYER")      (* HANDLE DIFFERENTLY
FROM OTHERS *)

          then

             let newV = {vdt = Ast.BoolType; vname="IS_IN_PLAYER";
value=Ast.Noexpr} in


             let flaggedUpdScope = {

                  parent = updScope.parent;

                  functions = updScope.functions;

                  variables = ( newV::updScope.variables;) } in  (*
flag is_in_player so that all player vdecls are added to global
with a special string prefix *)


             let unFlag scopeF = let newVars = (List.filter (fun x
-> (x.vname)<>"IS_IN_PLAYER") scopeF.variables) in
```

```
                              { parent = scopeF.parent;

functions = scopeF.functions;

  variables = newVars }
             in



             let bbodyAndScope = List.fold_left
processStatement(flaggedUpdScope,[]) bdecl.bbody

         in  (* IN: stmtList, scope, bbody   -->>   OUT: stmt list,
updated symbol table *)

             let rec isAllVdecls stmtList = match stmtList with

             | [] -> true

             | Ast.Vdecl(x)::tail -> isAllVdecls tail

             | _ -> false

             in

          let returnBdecl = {

             bname = bdecl.bname;

             bid =   bdecl.bid;

             funcs = fst funcsAndScope;

             bbody = fst(bdecl.bbody, snd bbodyAndScope);   (* UGLY
HACK (ast instead of sast) *)

          }  in

             if isAllVdecls bdecl.bbody  (* PLAYER bbody must only
have variable declarations *)

               then (unFlag (fst(bbodyAndScope))),
returnBdecl::checkedBdecls  (* intermediate scope and bdecl list *)

          else raise(Failure("Player block must consist only of
variable declarations"))
```

```
        else

          let bbodyAndScope = List.fold_left
processStatement(updScope,[]) bdecl.bbody

         in  (* IN: stmtList, scope, bbody   -->>   OUT: stmt list,
updated symbol table *)

          let returnBdecl = {

              bname = bdecl.bname;

              bid =   bdecl.bid;

              funcs = fst funcsAndScope;

              bbody = fst(bdecl.bbody, snd bbodyAndScope);   (* UGLY
HACK (ast instead of sast) *)

          }  in

          fst(bbodyAndScope), returnBdecl::checkedBdecls  (*
intermediate scope and bdecl list *)
```

```
(* START valid next blocks for each block - - - - - - - - - - - - -
- - - - - - - - - - - -*)

(* remove numbers from end of TURN blocks *)

let turnTrim string =

  if ( (String.length string > 3) && (String.sub string 0 4 =
"TURN") )

    then "TURN"

  else string
```

```
let curValid state block = match ( turnTrim state, turnTrim block)
with ("START", "PLAYER") -> true
  | ("START", "SETUP") -> true
  | ("PLAYER", "SETUP") -> true
  | ("SETUP", "TURN") -> true
  | ("SETUP", "WIN") -> true
  | ("SETUP", "END") -> true
  | ("TURN", "TURN") -> true
  | ("TURN", "WIN") -> true
  | ("TURN", "END") -> true
  | ("WIN", "END") -> true
  | _ -> false


(* check that blocks are in a valid order *)
let rec allValid state blocks = match blocks with
        | [] -> let _ = print_string "Order check finished \n" in
true
        | head::tail -> if curValid state head && allValid head
tail
                   then true
                 else raise(Failure("Blocks not in correct order"))


let startState = "START"


let check_order block_list =
    let extract_name bdecl = bdecl.bname in
    let block_name_list = List.map extract_name block_list in
        allValid startState block_name_list
(* END valid next blocks for each block - - - - - - - - - - - - -  -
- - - - - - - - - - -*)






(* MAIN *)
(* Parameters:  Ast.Program, Symbol_Table(global)  *)
(* Output:     Sast.Program *)
let checkProgram program globalTable =
  let _ = check_order program in
  let endScope, _ = List.fold_left processBdecl(globalTable, [])
program in


  (* endScope *)


  (*
  (* debug info below *)
  let rec getGlobalScope scope = match scope.parent with
        | None -> scope
        | Some(parent) -> (getGlobalScope parent)
  in
  let build_string tmpString nextString = tmpString^"
\n"^nextString in
```

```
  let func_names_string = List.fold_left build_string("") (List.map
(fun {fdt=_; fname=n; formals=_; fbody=_} -> n ) (getGlobalScope
endScope).functions) in

  let num_funcs = List.length (getGlobalScope endScope).functions
in


  let var_names_string = List.fold_left build_string("") (List.map
(fun {vdt=_; vname=n; value=_} -> n ) (getGlobalScope
endScope).variables) in

  let num_vars = List.length (getGlobalScope endScope).functions in

  let _ = print_string ("At end, funcs found were "^((string_of_int
num_vars)^var_names_string)^" \n"

                ^"and var found were "^((string_of_int
num_funcs)^func_names_string) ) in

  *)


  endScope
```

sast.mli

```
(* Author Ryan Jones and Kevin Henrick *)


open Ast


type expr_detail =
      Int of int
    | Double of float
    | Bool of bool
    | String of string
    | Card of int * char
    | CardExpr of expr * char
    | List of expr list
    | Player of expr * expr
    | Id of string
    | Ele
    | Your
    | Binop of expr * Ast.binop * expr
    | Unopl of unopl * expr
    | Unopr of expr * Ast.unopr
    | Call of string * expr list
    | Noexpr
  and expr = expr_detail * Ast.datatype



type stmt =
      Block of (stmt list) (* statement list and var list - only
line changed here*)
    | Expr of expr
    | Vdecl of Ast.vdecl
    | Return of expr
    | If of expr * stmt * stmt
    | Foreach of expr * stmt

    | While of expr * stmt


type formal =

{

    pdt : Ast.datatype;

    pname : string;

}


type fdecl =

{

    fdt : Ast.datatype;

    fname : string;

    formals : Ast.formal list;

    fbody : stmt list;

}


type bdecl =

{

    bname : string;

    bid : int;

    funcs : fdecl list;

    bbody : stmt list;

}


type program = bdecl list
```

generator.ml

```
(*
 * Author: Mark Micchelli
 * The generator takes in a CGL AST and converts it to a string of
a Java
 * program. The two functions from here you'd want to call are
 * string_of_player_class and string_of_main_class.
 *)

open Ast
open Corelibrary

let string_of_datatype = function
      IntType -> "Integer"
    | DoubleType -> "Double"
    | BoolType -> "Boolean"
    | StringType -> "String"
    | CardType -> "Card"
    | ListType -> "CGLList"
    | PlayerType -> "Player"
    | AnytypeType -> "Object"

let string_of_binop = function
```

```
      Assign -> "="
    | Plus -> "+"
      | Sub -> "-"
    | Mult -> "*"
    | Div -> "/"
    | Mod -> "%"
    | Eq -> "never reached"
    | Neq -> "never reached"
    | Teq -> "never reached"
    | Tneq -> "never reached"
    | Lt -> "<"
    | Leq -> "<="
    | Gt -> ">"
    | Geq -> ">="
    | And -> "&&"
    | Or -> "||"
    | Concat -> "+"
    | Addl -> "never reached"
    | Addr -> "never reached"
      | Dot -> "never reached"

let rec string_of_expr = function
      Int(i) -> "new Integer(" ^ string_of_int i ^ ")"
    | Double(d) -> "new Double(" ^ string_of_float d ^ ")"
    | Bool(b) -> "new Boolean(" ^ string_of_bool b ^ ")"
    | String(s) -> "new String(" ^ s ^ ")"
    | Card(i, c) ->
        "new Card(" ^ string_of_int i ^ ", '" ^ Char.escaped c ^
"')"
    | CardExpr(e, c) ->
        "new Card(" ^ string_of_expr e ^ ", '" ^ Char.escaped c ^
"')"
    | List(el) ->
        "new CGLList(" ^ String.concat ", "
        (List.map string_of_expr el) ^ ")"
    | Player(e1, e2) ->
        "new Player(" ^ string_of_expr e1 ^ ", " ^
        string_of_expr e2 ^ ")"
    | Id(s) -> s
    | Ele -> "ele"
    | Your -> "your"
    | Binop(e1, o, e2) ->
      let string_of_binop_expr e1 o e2 = match o with
          Eq -> string_of_expr e1 ^ ".equals(" ^ string_of_expr e2
^ ")"
        | Neq -> "!" ^ string_of_expr e1 ^ ".equals(" ^
string_of_expr e2 ^ ")"
      | Teq -> "teq(" ^ string_of_expr e1 ^ ", " ^ string_of_expr
e2 ^ ")"
        | Tneq -> "!teq(" ^ string_of_expr e1 ^ ", " ^
string_of_expr e2
            ^ ")"
        | Addl ->
            string_of_expr e1 ^ ".addLast(" ^ string_of_expr e2 ^
")"
```

```
        | Addr ->
            string_of_expr e2 ^ ".addFirst(" ^ string_of_expr e1 ^
")"
        | Dot -> string_of_expr e1 ^ "." ^ string_of_expr e2
        | _ -> string_of_expr e1 ^ " " ^ string_of_binop o ^ " " ^
                string_of_expr e2
      in string_of_binop_expr e1 o e2
    | Unopl(o, e) ->
      let string_of_unopl_expr o e = match o with
        Not -> "!" ^ string_of_expr e
      | Reml -> string_of_expr e ^ ".removeFirst()"
        in string_of_unopl_expr o e
    | Unopr(e, o) -> string_of_expr e ^ ".removeLast()"
  | Call(f, el) ->
        f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
")"
    | Noexpr -> ""


let rec string_of_stmt = function
      Block(stmts) ->
        "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
"}\n"
    | Expr(expr) ->
        string_of_expr expr ^ ";\n"
    | Vdecl(vdecl) ->
        string_of_datatype vdecl.vdt ^ " " ^ vdecl.vname ^ " = " ^
        string_of_expr vdecl.value ^ ";\n"
    | Return(expr) ->
```

```
        "return " ^ string_of_expr expr ^ ";\n";

    | If(e, s, Block([])) ->

        "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s

    | If(e, s1, s2) ->

        "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^

        "else\n" ^ string_of_stmt s2

    | Foreach(e, s) ->

        "list = " ^ string_of_expr e ^ ".toArray();

        for (Object ele : list)

            " ^ string_of_stmt s ^ "

        "


    | While(e, s) ->

        "while (" ^ string_of_expr e ^ ") \n" ^ string_of_stmt s


(*

 * Appends each formal with a "$". There is no risk of variable
name

 * overlap because "$" is a character that's allowed in Java
identifiers,

 * but not CGL ones.

 *)

let string_of_formal formal =

     "Object " ^ formal.pname ^ "$$"


(*

 * Makes Java function declarations, but this requires a lot of
care because
```

```
 * CGL is pass-by-value and Java is pass-by-reference. I get around
this

 * by cloning each formal at the beginning of each function, and by

 * referring only to the cloned variables throughout the rest of
the

 * function. Because each formal is appended with a "$", the cloned

 * variables can be referred to by their unaltered CGL names.

 *)

let string_of_fdecl fdecl =

    (* Clone all non-primitive formals into the original variables.
*)

    let clone formal =

        let name = formal.pname in

        match formal.pdt with

          IntType ->

            "

            Integer " ^ name ^ "$ = (Integer) " ^ name ^ "$$;

            Integer " ^ name ^ " = new Integer(" ^ name ^
"$.intValue());

            "

        | DoubleType ->

            "

            Double " ^ name ^ "$ = (Double) " ^ name ^ "$$;

            Double " ^ name ^ " = new Double(" ^ name ^
"$.doubleValue());

            "

        | BoolType ->

            "

            Boolean " ^ name ^ "$ = (Boolean) " ^ name ^ "$$;
            Boolean " ^ name ^ " = new Boolean(" ^ name ^
            "$.booleanValue());
            "
        | StringType ->
            "
            String " ^ name ^ "$ = (String) " ^ name ^ "$$;
            String " ^ name ^ " = new String(" ^ name ^ "$);
            "
        | CardType ->
            "
            Card " ^ name ^ "$ = (Card) " ^ name ^ "$$;
            Card " ^ name ^ " = new Card(value(" ^ name ^ "$),
suit(" ^
            name ^ "$));
            "
        | ListType ->
            "
            CGLList " ^ name ^ "$ = (CGLList) " ^ name ^ "$$;
            CGLList " ^ name ^ " = new CGLList();
            ListIterator iter$$$ = " ^ name ^ "$.listIterator(0);
            while (iter$$$.hasNext())
                " ^ name ^ ".addLast(iter$$$.next());
            "
        (* needs work -- how do you copy over other fields? *)
        | PlayerType ->
            "
            Player " ^ name ^ "$ = (Player) " ^ name ^ "$$;
            Player " ^ name ^ " = new Player(" ^ name ^ "$.name, " 
^ name ^

            "$.turnID);

            "

        (* also needs work -- probably won't clone properly *)

        | AnytypeType -> "Object " ^ name ^ " = " ^ name ^ "$$;"

    in

    (* Finally, the Java code proper. *)

    "private static " ^ string_of_datatype fdecl.fdt ^ " " ^ 
fdecl.fname

    ^ "(" ^ String.concat ", " (List.map string_of_formal 
fdecl.formals) ^

    ")\n{\n" ^

    String.concat "" (List.map clone fdecl.formals) ^

    String.concat "" (List.map string_of_stmt fdecl.fbody) ^

    "}\n"


(* Creates the Player class *)

let string_of_player_class bdecl_list =

    let first_block = List.hd bdecl_list in

    (* Creates the variables at the bottom of the class *)

    let make_player_vars bdecl =

        if bdecl.bname = "PLAYER" then

            let string_of_player_var stmt = match stmt with

              Vdecl(vdecl) ->

                "public " ^ string_of_datatype vdecl.vdt ^ " " ^

                vdecl.vname ^ ";\n"

            | _ -> ""
            in
            String.concat "" (List.map string_of_player_var
bdecl.bbody)
        else ""
    in
    (* Assigns values to those variables in the constructor *)
    let make_player_construct bdecl =
        if bdecl.bname = "PLAYER" then
            let string_of_player_assn stmt = match stmt with
              Vdecl(vdecl) ->
                  vdecl.vname ^ " = " ^ string_of_expr vdecl.value
^ ";\n"
            | _ -> ""
            in
            String.concat "\t" (List.map string_of_player_assn
bdecl.bbody)
        else ""
    in
    "
    public class Player
    {
        public Player(String name, Integer turnID)
        {
            this.name = name;
            this.turnID = turnID;
            " ^ make_player_construct first_block ^ "
        }
        public boolean equals(Object other)
        {
            if (other instanceof Player)
            {
                Player that = (Player) other;
                boolean sameName = this.name.equals(that.name);
                boolean sameID = this.turnID.equals(that.turnID);
                return sameName && sameID;
            }
            return false;
        }

        public static Player NEMO = new Player(\"NEMO\", -1);
        public String name;
        public Integer turnID;
        " ^ make_player_vars first_block ^ "
    }
    "

(* Writes the turn() function of the core library *)
let string_of_turn_block_list block_list =
    if List.length block_list > 0 then
        let string_of_turn_block block =
            "
            if (your.turnID == " ^ string_of_int block.bid ^ ")
            {" ^
            String.concat "" (List.map string_of_stmt block.bbody)
^
            "}
            else " in
        "private static void turn(Player your)
        {
        " ^ String.concat "" (List.map string_of_turn_block
block_list) ^
        "if (your.turnID < 0)
            win();
        System.exit(0);
        }
        "
    else ""


(* Writes the win() function of the core library *)
let string_of_win_block block =
    "private static void win()
    {
    " ^ String.concat "" (List.map string_of_stmt block.bbody) ^
    "
    System.exit(0);
    }
    "


(* The primary Java code generator function. *)
let make_main setup turn_list win =
    (* strips the vdecls in the main() block of their datatypes and
access
     * level modifiers, because vdecls are global variables *)
    let string_of_setup_stmt stmt = match stmt with
          Vdecl(vdecl) ->
            vdecl.vname ^ " = " ^ string_of_expr vdecl.value ^
";\n"
        | _ -> string_of_stmt stmt
    in
    (* makes the vdecls global variables *)
    let string_of_setup_vdecl stmt = match stmt with
          Vdecl(vdecl) ->
            "private static " ^ string_of_datatype vdecl.vdt ^ " "
^
            vdecl.vname ^ ";\n"
        | _ -> ""
    in


    (* converts the turn and win blocks into their corresponding
strings *)
    let turn_block = string_of_turn_block_list turn_list in
    let win_block = match win.bname with
          "WIN" -> string_of_win_block win
        | _ -> "public static void win() {}" in


    (* creates the string of the setup block by invoking Java's
main()
     * function; also accounts for other necessary parts of the
program *)
```

```
    "
    import java.util.*;


    public class Main {
    " ^ turn_block ^ "\n" ^ win_block ^ "\n" ^
    String.concat "\n" (List.map string_of_fdecl setup.funcs) ^
    Corelibrary.coreFunc ^ "\n" ^
    "
    public static void main(String[] args)
    {" ^
        Corelibrary.fillConst ^ "\n" ^
        String.concat "" (List.map string_of_setup_stmt setup.bbody)
^
    "}


    " ^
    Corelibrary.coreConst ^ "\n" ^
    String.concat "" (List.map string_of_setup_vdecl setup.bbody) ^
"\n" ^
    "private static Object[] list;
    }\n"


(* Figures out the PLAYER/SETUP/TURN/WIN structure of the CGL
program, and
 * then calls make_main with the appropriate arguments
 *)
let string_of_main_class bdecl_list =
    let empty = { bname = ""; bid = 0; funcs = []; bbody = []; } in
    let a = Array.of_list bdecl_list in
    let length = Array.length a in
    let first = a.(0) in
    let last = a.(length - 1) in
    if first.bname = "PLAYER" then
        let setup = a.(1) in
        if last.bname = "WIN" then
            let turn_array = Array.sub a 2 (length - 3) in
            let turn_list = Array.to_list turn_array in
            make_main setup turn_list last
        else
            let turn_array = Array.sub a 2 (length - 2) in
            let turn_list = Array.to_list turn_array in
            make_main setup turn_list empty
    else
        let setup = a.(0) in
        if last.bname = "WIN" then
            let turn_array = Array.sub a 1 (length - 2) in
            let turn_list = Array.to_list turn_array in
            make_main setup turn_list last
        else
            let turn_array = Array.sub a 1 (length - 1) in
            let turn_list = Array.to_list turn_array in
            make_main setup turn_list empty
```

corelibrary.ml

```
(* Author: Mark Micchelli *)

let teq =
    "
    private static boolean teq(Object o1, Object o2)
    {
        return o1.getClass().isAssignableFrom(o2.getClass());
    }
    "

let intToString =
    "
    private static String intToString(Object toCast)
    {
        Integer i = (Integer) toCast;
        return i.toString();
    }
    "

let doubleToString =
    "
    private static String doubleToString(Object toCast)
    {
        Double d = (Double) toCast;
        return d.toString();
    }
    "


let stringToInt =
    "
    private static Integer stringToInt(Object toCast)
    {
        String s = (String) toCast;
        return Integer.parseInt(s);
    }
    "

let stringToDouble =
    "
    private static Double stringToDouble(Object toCast)
    {
       String s = (String) toCast;
       return Double.parseDouble(s);
    }
    "


let scan =
    "
    private static String scan()
    {
        Scanner input = new Scanner(System.in);
        return input.nextLine();
    }
    "

let print =
    "
    private static void print(Object toCast)
    {
        String toPrint = (String) toCast;
        System.out.print(toPrint);
    }
    "

let value =
    "
    private static Integer value(Object toCast)
    {
        Card c = (Card) toCast;
        return c.getValue();
    }
    "

let suit =
    "
    private static String suit(Object toCast)
    {
        Card c = (Card) toCast;
        return c.getSuit();
    }
    "

let random =
    "
    private static Integer random(Object toCast1, Object toCast2)
    {
        Integer lower = (Integer) toCast1;
        Integer higher = (Integer) toCast2;
        Random r = new Random();
        return lower + r.nextInt(higher + 1);
    }
    "

(* The Fisher-Yates shuffle: en.wikipedia.org/wiki/Fisher-
Yates_shuffle *)
let shuffle =
    "
    private static CGLList shuffle(Object toCast)
    {
        CGLList l = (CGLList) toCast;
        Object[] a = l.toArray();
        int len = a.length;

        Random r = new Random();

        for (int i = len - 1; i >= 1; i--)

        {

            int j = r.nextInt(i + 1);

            Object temp = a[i];

            a[i] = a[j];

            a[j] = temp;

        }


        CGLList res = new CGLList();

        for (Object ele : a)

            res.addLast(ele);

        return res;

    }

    "


let nemo =

    "private static Player NEMO = Player.NEMO;\n"


let standard =

    "private static CGLList STANDARD;\n"


let fillConst =

    "STANDARD = new CGLList();

    STANDARD.addLast(new Card(2, 'C'));
    STANDARD.addLast(new Card(3, 'C'));

    STANDARD.addLast(new Card(4, 'C'));

    STANDARD.addLast(new Card(5, 'C'));

    STANDARD.addLast(new Card(6, 'C'));

    STANDARD.addLast(new Card(7, 'C'));

    STANDARD.addLast(new Card(8, 'C'));

    STANDARD.addLast(new Card(9, 'C'));

    STANDARD.addLast(new Card(10, 'C'));

    STANDARD.addLast(new Card(11, 'C'));

    STANDARD.addLast(new Card(12, 'C'));

    STANDARD.addLast(new Card(13, 'C'));

    STANDARD.addLast(new Card(14, 'C'));

    STANDARD.addLast(new Card(2, 'D'));

    STANDARD.addLast(new Card(3, 'D'));

    STANDARD.addLast(new Card(4, 'D'));

    STANDARD.addLast(new Card(5, 'D'));

    STANDARD.addLast(new Card(6, 'D'));

    STANDARD.addLast(new Card(7, 'D'));

    STANDARD.addLast(new Card(8, 'D'));

    STANDARD.addLast(new Card(9, 'D'));

    STANDARD.addLast(new Card(10, 'D'));

    STANDARD.addLast(new Card(11, 'D'));

    STANDARD.addLast(new Card(12, 'D'));

    STANDARD.addLast(new Card(13, 'D'));

    STANDARD.addLast(new Card(14, 'D'));

    STANDARD.addLast(new Card(2, 'H'));

    STANDARD.addLast(new Card(3, 'H'));
```

```
    STANDARD.addLast(new Card(4, 'H'));
    STANDARD.addLast(new Card(5, 'H'));
    STANDARD.addLast(new Card(6, 'H'));
    STANDARD.addLast(new Card(7, 'H'));
    STANDARD.addLast(new Card(8, 'H'));
    STANDARD.addLast(new Card(9, 'H'));
    STANDARD.addLast(new Card(10, 'H'));
    STANDARD.addLast(new Card(11, 'H'));
    STANDARD.addLast(new Card(12, 'H'));
    STANDARD.addLast(new Card(13, 'H'));
    STANDARD.addLast(new Card(14, 'H'));
    STANDARD.addLast(new Card(2, 'S'));
    STANDARD.addLast(new Card(3, 'S'));
    STANDARD.addLast(new Card(4, 'S'));
    STANDARD.addLast(new Card(5, 'S'));
    STANDARD.addLast(new Card(6, 'S'));
    STANDARD.addLast(new Card(7, 'S'));
    STANDARD.addLast(new Card(8, 'S'));
    STANDARD.addLast(new Card(9, 'S'));
    STANDARD.addLast(new Card(10, 'S'));
    STANDARD.addLast(new Card(11, 'S'));
    STANDARD.addLast(new Card(12, 'S'));
    STANDARD.addLast(new Card(13, 'S'));
    STANDARD.addLast(new Card(14, 'S'));"

let coreFunc =
    teq ^ intToString ^ stringToInt ^ doubleToString ^
stringToDouble ^ suit ^
    value ^ print ^ scan ^ random ^ shuffle

let coreConst = nemo ^ standard
```

javaclasses.ml

```
(* Author: Mark Micchelli *)

let string_of_card_class =
"
    public class Card {
    public Card(Integer value, String suit) {
        this.value = value;
        this.suit = suit;
    }

    public Card(int i, char s) {
        value = new Integer(i);
        suit = String.valueOf(s);
    }


    public Integer getValue() { return value; }
    public String getSuit() { return suit; }

    public boolean equals(Object other)

    {

        if (other instanceof Card)

        {

            Card that = (Card) other;

            boolean sameValue = false;

            boolean sameSuit = false;

            if (value.equals(that.getValue()) ||

                that.getValue().intValue() == 0)

                sameValue = true;

            if (suit.equals(that.getSuit()) ||
that.getSuit().equals(\"*\"))

                sameSuit = true;

            return sameValue && sameSuit;

        }

        return false;

    }


    private Integer value;

    private String suit;

    }

"


let string_of_list_class =

"

import java.util.*;
```

```
public class CGLList
{
  public CGLList()
  {
        list = new LinkedList<Object>();
  }

  public CGLList(Object... elements)
  {
        list = new LinkedList<Object>();
        for (Object ele : elements)
              list.addLast(ele);
  }

  public void addFirst(Object ele)
  {
        list.addFirst(ele);
  }

  public void addLast(Object ele)
  {
        list.addLast(ele);
  }

    // the remove methods are the source of the casting warnings
```

```
public <T> T removeFirst()

{

      return (T) list.removeFirst();

}


public <T> T removeLast()

{

      return (T) list.removeLast();

}


public int size()

{

      return list.size();

}


public Object[] toArray()

{

      return list.toArray();

}


  public ListIterator<Object> listIterator(int index)

  {

      return list.listIterator(index);

  }


public boolean equals(Object other)
```

```
  {
        if (other instanceof CGLList)
        {
            CGLList that = (CGLList) other;
            if (this.size() != that.size()) return false;
            ListIterator<Object> i1 = this.listIterator(0);
            ListIterator<Object> i2 = that.listIterator(0);
            while (i1.hasNext())
            {
                if (!i1.next().equals(i2.next())) return false;
            }
            return true;
        }
        return false;
    }

    private LinkedList<Object> list;
}
"
```

cgl.ml

```
(* Author: Ryan Jones and Mark Micchelli *)

open Parser
open Generator
open Javaclasses
open Semantic_analyzer

type action = Semantic | Java | Classes | Execute
exception InvalidOption of string
exception WrongNumOfArguments of string

let usage_string = "Usage: ./cgl [-s|-j|-c|-e] filename.cgl"
let _ =
    let num_args = Array.length Sys.argv in
    let get_action =
        if num_args == 3 then
            match Sys.argv.(1) with
                  "-s" -> Semantic
                | "-j" -> Java
                | "-c" -> Classes
                | "-e" -> Execute
                | _ -> raise (InvalidOption(usage_string))
        else raise (WrongNumOfArguments(usage_string)) in
    let in_channel = open_in Sys.argv.(2) in
    let lexbuf = Lexing.from_channel in_channel in
    let program = Parser.program Scanner.token lexbuf in
    let main_string = Generator.string_of_main_class program in
    let player_string = Generator.string_of_player_class program in
    let card_string = Javaclasses.string_of_card_class in
    let list_string = Javaclasses.string_of_list_class in
    let execute_action = match get_action with
      Semantic ->
    (
        try
        let _ = checkProgram program global_scope in
        print_string "passed semantic checking \n"
        with Failure(msg) ->
        print_string ("didn't pass semantic checking: \n" ^ msg ^
"\n")
    )
    | Java ->
        let create_files =
            let main_java = open_out "Main.java" in
            let player_java = open_out "Player.java" in
            let card_java = open_out "Card.java" in
            let list_java = open_out "CGLList.java" in
            output_string main_java main_string;
            output_string player_java player_string;
            output_string card_java card_string;
            output_string list_java list_string
        in
      create_files
    | Classes -> print_string "not yet implemented\n"
    | Execute -> print_string "not yet implemented\n"
    in execute_action
```

makefile

```
# Authors: Kevin Henrick, Ryan Jones, and Mark Micchelli

OBJS = scanner.cmo parser.cmo corelibrary.cmo javaclasses.cmo
generator.cmo semantic_analyzer.cmo cgl.cmo

cgl: $(OBJS)
  ocamlc -o cgl $(OBJS)

scanner.ml : scanner.mll
  ocamllex scanner.mll

parser.ml parser.mli: parser.mly
  ocamlyacc parser.mly

%.cmo : %.ml
  ocamlc -c $<

%.cmi : %.mli
  ocamlc -c $<

.PHONY : clean
clean :
  rm -rf *.cmo *.cmi *.java *.class cgl parser.mli parser.ml
scanner.ml
```

```
# generated by ocamldep
cgl.cmo: scanner.cmo parser.cmi javaclasses.cmo generator.cmo
cgl.cmx: scanner.cmx parser.cmx javaclasses.cmx generator.cmx
corelibrary.cmo:
corelibrary.cmx:
generator.cmo: corelibrary.cmo ast.cmi
generator.cmx: corelibrary.cmx ast.cmi
generatorbackup.cmo: corelibrary.cmo ast.cmi
generatorbackup.cmx: corelibrary.cmx ast.cmi
javaclasses.cmo:
javaclasses.cmx:
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
scanner.cmo: parser.cmi ast.cmi
scanner.cmx: parser.cmx ast.cmi
ast.cmi:
parser.cmi: ast.cmi
semantic_analyzer.cmo: ast.cmi sast.cmi
```

Appendix C. Example Games

C.1 Simplified Blackjack

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

SETUP
{
     /* card function */
     def string valueToString(int value)
     {
          if (value == 14) return "A";
          else if (value == 13) return "K";
          else if (value == 12) return "Q";
          else if (value == 11) return "J";
          else if (value == 0) return "*";
          else if (value < 2 || value > 14) return "INVALID";
          else return intToString(value);
     }

     /* list function */
     def int length (list l)
     {
          int length = 0;
          foreach(l) { length = length + 1; }
          return length;
     }
```

```
/* list function */
def anytype get(int index, list l)
{
     if (index > length(l))
     {
          print("index too high\n");
          return -1;
     }
     else
     {
          int i = 1;
          foreach(l)
          {
               if (i == index) return ele;
               i = i + 1;
          }
     }
     return "never reached";
}

/* list function */
def bool in(anytype e, list l)
{
     bool in = false;
     foreach(l)
     {
```

```
                if (ele === e && ele == e)
                      in = true;
           }
           return in;
     }


     def int bigBreak()
     {
           print("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
^

"\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
           return 0;
     }


     /* calculates blackjack score (ace is always 11 for convenience
*/
     def int score (list hand)
     {
           int handScore = 0;
           foreach (hand)
           {
                if (ele == $A*) handScore = handScore + 11;
                else if (ele == $J* || ele == $Q* || ele == $K*)
                      handScore = handScore + 10;
                else handScore = handScore + value(ele);
           }
           return handScore;
```

```
}

/* takes in player name and AI info */
def list addPlayers(list validIDs)
{
      list players = [];
      while (length(players) < 4)
      {
            print("please enter player name\n");
            string name = scan();
            print("please enter 1 if human, or 2 if AI\n");
            string id = scan();
            int turnID = stringToInt(id);
            if (in(turnID, validIDs))
                  players <+ <name, turnID>;
            else
                  print("invalid input\n");
      }
      return players;
}

/* code proper */
list players = addPlayers([1, 2]);
player p1 = <- players;
player p2 = <- players;
player p3 = <- players;
player p4 = <- players;
```

```
    p1.next = p2;

    p2.next = p3;

    p3.next = p4;

    p4.next = NEMO;


    int maxScore = 21;

    list deck = STANDARD;

    deck = shuffle(deck);


    p1.hand <+ <- deck;

    p2.hand <+ <- deck;

    p3.hand <+ <- deck;

    p4.hand <+ <- deck;

    p1.hand <+ <- deck;

    p2.hand <+ <- deck;

    p3.hand <+ <- deck;

    p4.hand <+ <- deck;


    turn(p1);

}


TURN 1

{

    your.score = score(your.hand);

    bool done = false;

    print(your.name ^ "'s turn; press enter to continue\n");
```

```
      scan();
      bigBreak();
      print("you have ");
      foreach (your.hand)
      {
            print(valueToString(value(ele)) ^ suit(ele) ^ " ");
      }
      print("\ntype \"h\" for hit; anything else for stay\n");
      string s = scan();
      if (s == "h")
      {
            card c = <- deck;
            your.hand <+ c;
            print("you got a " ^ valueToString(value(c)) ^ suit(c) ^
"\n");
            your.score = score(your.hand);
            if (your.score > maxScore)
            {
                  your.bust = true;
                  your.score = 0;
                  bigBreak();
                  print("bust!\n");
                  turn(your.next);
            }
            else
                  turn(your);
      }
```

```
    else
    {
        bigBreak();
        turn(your.next);
    }
}

/* AI turn */
TURN 2
{
    your.score = score(your.hand);
    if (your.score <= 14) /* AI only hits if score lower than 14 */
    {
        your.hand <+ <- deck;
        your.score = score(your.hand);
        if (your.score > maxScore)
        {
            your.bust = true;
            your.score = 0;
            turn(your.next);
        }
        else
            turn(your);
    }
    else
        turn(your.next);
}
```

```
WIN
{
      player best = NEMO;
      int bestScore = 0;
      list scores = [];
      scores <+ p1.score;
      scores <+ p2.score;
      scores <+ p3.score;
      scores <+ p4.score;
      print(p1.name ^ " scored " ^ p1.score ^ "\n");
      print(p2.name ^ " scored " ^ p2.score ^ "\n");
      print(p3.name ^ " scored " ^ p3.score ^ "\n");
      print(p4.name ^ " scored " ^ p4.score ^ "\n");

      int i = 0;
      while (length(scores) != 0)
      {
            int playerScore = <- scores;
            if (playerScore > bestScore)
                  bestScore = playerScore;
      }

      if (p1.score == bestScore) print(p1.name ^ " wins\n");
      if (p2.score == bestScore) print(p2.name ^ " wins\n");
      if (p3.score == bestScore) print(p3.name ^ " wins\n");
      if (p4.score == bestScore) print(p4.name ^ " wins\n");
}
```

C2. Finding_the_First_Ace

```
/* Author: Kevin Henrick, Hebo Yang */

/* This is a typical, but very short game that is used during a poker game to determine who deals
first.*/

/* In this example, there are four players. */

/* Finding_the_First_Ace */


PLAYER{

    player next = NEMO;

}


SETUP {

     print("please enter name for player 1\n");

     string name1 = scan();

     print("please enter name for player 2\n");

    string name2 = scan();

     print("please enter name for player 3\n");

    string name3 = scan();

     print("please enter name for player 4\n");

    string name4 = scan();

    player p1 = <name1, 1>;

    player p2 = <name2, 1>;

    player p3 = <name3, 1>;
    player p4 = <name4, 1>;

    list deck = STANDARD;

    deck = shuffle(deck);

    p1.next = p2;

    p2.next = p3;

    p3.next = p4;

    p4.next = p1;


     card c = $2H;

    turn(p1);

}


TURN 1

{

       bool properInput = false;


     while (!properInput){

                 print("please (d)raw a card from the deck\n");

                 string draw = scan();

                 properInput = true;

                 if (draw == "d"){

                     c = <- deck;

                             if (c == $A*){


                             print(your.name ^ " drew " ^ "A" ^ suit(c) ^ ", the first Ace, and gets to deal.
Shuffle 'em up!\n");
```

```
                    }
                        else if (c != $A*){
                        print(your.name ^ " drew " ^ intToString(value(c)) ^ suit(c) ^ ", " ^
your.next.name ^ "'s turn\n");
                      turn(your.next);
                  }
                        }
              else if (draw != "d"){
                        print("invalid input\n");
                        properInput = false;
              }
              }


}
```

C.3 highlow

```
/* Author: Mark Micchelli. Note: last year's DesCartes team implemented
high-low in 82 lines; CGL's implementation only takes 47 */


SETUP
{
    int score = 0;
    list deck = STANDARD;
    deck = shuffle(deck);
    player p = <"", 1>;
    card c = <- deck;
    int lastValue = value(c);
    print("the first card has value " ^ intToString(lastValue) ^ "\n");
    deck <+ c;
    turn(p);
}

TURN 1
{
    bool properInput = false;
    bool high = true;
    while (!properInput)
    {
            print("will the next card be (h)igher or (l)ower?\n");
            string guess = scan();
            properInput = true;
            if (guess == "l")
                    high = false;
            else if (guess != "h")
            {
                    print("invalid input\n");
                    properInput = false;
            }
    }
```

```
        c = <- deck;
        int thisValue = value(c);
        deck <+ c;
        print("new card's value is " ^ intToString(thisValue) ^ "\n");
        if ((thisValue > lastValue && high) || (thisValue < lastValue && !high))
        {
                print("correct prediction\n");
                score = score + 1;
                lastValue = thisValue;
                turn(p);
        }
        else
        {
                print("incorrect prediction; game over\n");
                print("total score = " ^ intToString(score) ^ "\n");
        }
    }
```

APPENDIX D. Changes to the LRM

1) Deleted a feature about $12H being a valid card declaration; you must use $QH. - Mark Micchelli

2) Made change about SETUP block function scoping: Functions can now only be defined at the beginning of the SETUP block. - Kevin Henrick

3) Furthermore, the user can no longer declare functions in TURN n and WIN, only SETUP.

4) We got rid of external libraries altogether. There is no #include statement in the submitted version of CGL. - Mark Micchelli

5) There are now two new core library functions: doubleToString and stringToDouble. This was an obvious oversight in our original LRM, and it didn’t take us long to realize we needed them. - Mark Micchelli