

PCGSL: Playing Card Game Simulation Language

Enrique Henestroza
eh2348@columbia.edu

Yuriy Kagan
yk2159@columbia.edu

Andrew Shu
ans2120@columbia.edu

Peter Tsonev
pvt2101@columbia.edu

COMS W4115
Programming Languages and Translators
December 19, 2008

Contents

1	Introduction	3
1.1	Language Overview	3
2	Language Tutorial	5
2.1	Blocks	5
2.2	Card Features	6
2.3	Variables	6
2.4	Control Flow	7
2.5	Lists	7
2.6	Conclusion	7
3	Language Manual	9
3.1	Lexical Conventions	9
3.1.1	Comments	9
3.1.2	Identifiers	9
3.1.3	Keywords	10
3.1.4	Constants	10
3.1.5	Operators	10
3.1.6	Meaning of Identifiers	10
3.1.7	Scope, Namespace, and Storage Duration	11
3.2	Declarations	12
3.2.1	Variables	12
3.2.2	Functions	13
3.2.3	Special Blocks	13
3.3	Expressions and Operators	13

3.3.1	Precedence and Association Rules in PCGSL	13
3.3.2	Expressions	15
3.3.3	Function Calls	15
3.3.4	Assignment	15
3.4	Statements	16
3.4.1	Expression Statements	16
3.4.2	Selection Statements	16
3.4.3	Iteration Statements	16
3.4.4	Jump Statements	17
4	Project Plan	18
4.1	Planning & Specification	18
4.2	Development & Testing	19
4.3	Programming Style Guide	19
4.4	Project timeline	21
4.5	Roles and Responsibilities	21
4.6	Software Development Environment	22
4.7	Project Log	22
5	Architectural Design	23
6	Test Plan	25
7	Lessons Learned	28
7.1	Enrique Henestroza	28
7.2	Yuriy Kagan	28
7.3	Andrew Shu	28
7.4	Peter Tsonev	29
8	Appendix	30

Chapter 1

Introduction

The Playing Card Game Simulation Language (PCGSL) is designed to be a simple programming language for simulating card games. Our language allows a programmer to work within a standard set of conventions and procedures for playing card games, without having to write a large amount of code as would be required with a general-purpose language. Using PCGSL, the programmer can focus on creating randomized simulations of popular games, or developing quick mock-ups of new games based around standard 52-card decks.

1.1 Language Overview

PCGSL is an interpreted language, implemented in O’Caml. It reads source files by argument from the command line, so it is not interactive.

PCGSL syntax is simple and recognizable, following well-known C-style imperative programming syntax conventions. Certain aspects of the PCGSL language are simplified for convenience: there are no typed declarations or typed return values for functions, which allows for quick and efficient programming reminiscent of scripting languages like Perl and Python.

Additionally, PCGSL introduces a powerful yet unobtrusive set of language features to facilitate programming in the domain of card games. PCGSL comes with built-in `Card` and `CardEntity` types, which allow a useful level of abstraction for working with cards from standard 52-card decks and the players, dealers, and other meaningful entities that possess varying subsets of cards throughout the course of a card game. Specialized operators allow for the easy transferral of cards from one entity to another, and the language places useful

restrictions on the behavior of cards - for instance, at any given point during the course of a program's execution, each of the 52 cards is guaranteed to be in the possession of exactly one entity. These built-in features make card game programming hassle-free.

Chapter 2

Language Tutorial

Playing Card Simulation Game Language, as you can tell by its name, is a language specifically designed for writing card games. Hence it has a few features you will not find in most other languages. Aside from these special features, the syntax shares many features with C and Java. Users of these languages will be able to pick up PCGSL easily.

2.1 Blocks

The first special feature of PCGSL is that the program consists of a series of blocks:

- Includes: Here you specify any include files you want, such as the standard library.
- Globals: Here you specify any global variables. They will be initialized to null.
- CardEntities: Here you specify CardEntity names. The first contains all the cards by default.
- Start: The program begins here. This block is meant for initialization routines.
- Play: The main logic of the program goes here. It is automatically executed after Start.
- WinCondition: Logic to check for a winner goes here. If null is returned here, there is no winner yet. However, if a variable containing a list is returned, then that signals the end of the game. An empty list returned means there were no winners. The game programmer can also put in code to print out the winners here.

- (Other functions): You can declare other functions too, which can be used as expressions in your main program.

2.2 Card Features

Among the special language features of PCGSL are the CardEntities. These can be thought of as card owners (the deck, players, the cards on the board, ...). Each of the standard 52 cards belongs to exactly one CardEntity. The transfer operator is used to transfer cards between entities, without having to know the prior owner of the card. You just specify the CardEntity destination and the card by its built-in name, which is one of:

- *Clubs*: CA, C2, C3, C4, C5, C6, C7, C8, C9, C10, CJ, CQ, CK
- *Diamonds*: DA, D2, D3, D4, D5, D6, D7, D8, D9, D10, DJ, DQ, DK
- *Hearts*: HA, H2, H3, H4, H5, H6, H7, H8, H9, H10, HJ, HQ, HK
- *Spades*: SA, S2, S3, S4, S5, S6, S7, S8, S9, S10, SJ, SQ, SK

2.3 Variables

The types in PCGSL are integers, booleans, strings, cards, lists, and CardEntities. To declare a variable, you must put the following statement at the beginning of a function (including Start):

```
var varname;
```

As you can see, there are no typed declarations. Therefore, once you declare a variable, you can store any variable type inside it, and later change to a different type.

To refer to local variables, that is, those declared with var, you simply use the variable anywhere you would normally use a literal. To refer to a global variable, prefix the # symbol in front of the variable name, like: #globalvar. To refer to a CardEntity, you use the \$symbol like: \$entityvar. CardEntity variables are special in that you can store references to them in regular vars.

2.4 Control Flow

There are If and While statements too. The syntax is like C; however if you want to chain together a series of conditionals (if... else if...) you must enclose each following statement in another pair of braces. Like this:

```
if (some_condition) {  
    do_something;  
} else {  
    if (some_other_condition) {  
        do_something_else;  
    } else {  
        do_yet_another_thing;  
    }  
}
```

The << operator is used for printing. Each print statement goes on its own line:

```
<< Hello, world!;
```

2.5 Lists

List variables are written as literals using [element_a, element_b, element_c]. To dereference a list, use square brackets containing some expression that evaluates to an int. listvar[123]. You can also dereference a CardEntity treating it as a list, and it will return the card stored at that position in its list.

2.6 Conclusion

That's it for this tutorial. It assumes you're familiar with C or Java, and will be able to read a program to figure out how it works. There are other language features that are covered in the Language Reference Manual that are not covered here.

You should check out the standard library, called `stdlib.cgl`, which contains functions such as shuffling, list operations, card operations, and conversions from string to other types (useful when reading input from the user, which is returned as a string).

Also check out the sample programs provided, one of which is a simple poker implementation, another is a basic sample that returns a high card from a deck. They cover the whole spectrum of useful language features for writing your own card game.

Chapter 3

Language Manual

3.1 Lexical Conventions

This section covers the lexical conventions including comments and tokens. A token is a series of contiguous characters that the compiler treats as a unit. Blanks, tabs, newlines, and comments are collectively known as white space. White space is ignored except as it serves to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

3.1.1 Comments

The `//` characters introduce a comment; a newline terminates a comment. The `//` characters do not indicate a comment when occurring within a string literal. Comments do not nest. Once the `//` introducing a comment are seen, all other characters are ignored until the ending newline is encountered.

3.1.2 Identifiers

An identifier is a sequence of letters, digits, and underscores (`_`). The first character cannot be a digit. Uppercase and lowercase letters are distinct. Identifier length is unlimited.

3.1.3 Keywords

The identifiers listed below are reserved for use as keywords and cannot be used for any other purpose. Among these are a group of reserved identifiers corresponds to card names for a standard 52-card deck.

break Play CardEntities return else Start false true Globals while If WinCondition Include
var null H2 H3 H4 H5 H6 H7 H8 H9 H10 HJ HQ HK HA D2 D3 D4 D5 D6 D7 D8 D9 D10
DJ DQ DK DA C2 C3 C4 C5 C6 C7 C8 C9 C10 CJ CQ CK CA S2 S3 S4 S5 S6 S7 S8 S9
S10 SJ SQ SK SA

3.1.4 Constants

The two types of constants are integer and character string.

Integer Constants

An integer constant consists of a sequence of one or more digits, and is always considered decimal.

String Constants

A string constant is a sequence of characters surrounded by double quotation marks, such as Hello World!. We consider characters to be ASCII printable characters.

3.1.5 Operators

An operator specifies an operation to be performed. The operators are shown below:

<-	++	+	--	-	*	/	+=	-=	*=	/=	==	=	!=	
<	<=	>	>=	&&		::		~	^	@	#	\$	>>	<<

3.1.6 Meaning of Identifiers

Identifiers are disambiguated by their type, scope, and namespace. No identifier will have linkage, and storage duration will be determined by the scope, e.g. identifiers within the

same scope will have the same storage duration.

Type

Our language has four fundamental object types `int`, `string`, `boolean`, and `Card`. In addition, there are two derived types: `list` and `CardEntity`. There is no notion of a floating point number since it is not really needed in card games. There is also no need for a `char` type, since it can be simulated by a string consisting of a single ASCII symbol.

- `int`: the `int` type can represent an arbitrary integer since it will be mapped to the OCaml integer internally.
- `bool`: the `bool` type represents a boolean, either of the value `'true'` or `'false'`. It is mapped to OCaml boolean internally.
- `string`: the `string` type will be able to hold arbitrary strings since it will also be mapped to OCaml strings internally.
- `Card`: the `Card` type is a basic type that represents one of the 52 cards in a standard playing card deck.
- `list`: the `list` type is a derived type since it is a collection of fundamental objects. They have an attribute called `length` that stores the length of the list.
- `CardEntity`: The `CardEntity` type represents a certain participant in the card game who can be active (e.g. a player) or passive (e.g. a deck or a flop). Each `CardEntity` has a list of `Card` objects that belongs to it, and special operators for transferring `Card` objects among `CardEntity` objects.

3.1.7 Scope, Namespace, and Storage Duration

Unlike C identifiers, PCGSL identifiers have no linkage, e.g. the scopes are disjoint.

Scope

The scope specifies the region where certain identifiers are visible. PCGSL employs static scope. There are two kinds of scope, and they do not intersect:

- Global scope variables defined within the Globals block have global scope. Global variables cannot be defined in functions or any other block. Global variables are accessed via the '#' symbol. Therefore there are no intersections with the local scope.
- Function/Block scope variables declared within a function or block will be visible within that function or block. Nested functions or blocks are disallowed in the language.

Namespace

Functions and blocks share a namespace. Variables have their own namespace, as do CardEntities. None of these three namespaces overlap.

Storage Duration

Local variables have automatic storage duration. Their lifetime expires after the function in which they are defined returns. Global variables have static storage duration and live from their declaration to the end of program execution.

3.2 Declarations

A declaration specifies the interpretation given to a set of identifiers. Declarations in PCGSL define variables (including lists), CardEntity objects, and functions. Variable declarations are untyped. Declarations have the following form:

1. Variable Declaration: *var identifier*;
2. Function Declaration: *identifier (parameter-list) {body}*;

3.2.1 Variables

Declared variables consist of the keyword *var* followed by an identifier. They are uninitialized, and are given a Null value when declared. Null is a special data type that can be compared to any other data type.

3.2.2 Functions

Functions in PCGSL have no return type (returning the wrong type generates a runtime error). Functions may only be declared in the global scope. The *parameter-type-list* is the list of parameter identifiers, separated by commas with each preceded by the keyword *var*. The *body* is optional, and contains variable declarations as well as statements to be executed.

3.2.3 Special Blocks

There are several special required blocks that are declared in global scope. All blocks must exist in every PCGSL program, and appear at the beginning of the source file in the order below (followed by function declarations):

1. *Include* {*file-list*} ; Block containing a comma delimited list of files to import (e.g. "stdlib/stdlib.cgl")
2. *CardEntities* {*entity-list*} ; Block containing a comma delimited list of card entities (e.g. player1)
3. *Globals* {*declaration-list*} ; Block containing variable declarations. These variables, and only these variables, have global scope.
4. *Start* {*statement-list*} ; Block containing the code executed at initialization of the program.
5. *Play* {*statement-list*}; Block that is called after the Start block. This block is executed repeatedly until the WinCondition block returns a non-Null value.
6. *WinCondition* {*statement-list*} ; Block that is called automatically after every play() function. The game stops when this returns a non-Null value.

All code in PCGSL must be contained in one of the above blocks or inside function bodies.

3.3 Expressions and Operators

3.3.1 Precedence and Association Rules in PCGSL

Precedence of operators is list in order from lowest to highest:

- IDs and literals Primary; L-R; Token.
- ^ Binary; L-R; String concatenation.
- || Binary; L-R; Logical OR.
- && Binary; L-R; Logical AND.
- = Binary; R-L; Assignment.
- += Binary; R-L; Assignment with addition.
- -= Binary; R-L; Assignment with subtraction.
- *= Binary; R-L; Assignment with multiplication.
- /= Binary; R-L; Assignment with division.
- :: Binary; R-L; Appending to a list.
- >> Unary; R-L; Reading in from standard input.
- << Unary; R-L; Printing to standard output.
- <- Binary; L-R; Card transfer.
- == Binary; L-R; Equality test.
- != Binary; L-R; Inequality test.
- < Binary; L-R; Less-than test.
- <= Binary; L-R; Less-than-or-equal-to test.
- > Binary; L-R; Greater-than test.
- >= Binary; L-R; Greater-than-or-equal-to test.
- + Binary; L-R; Addition.
- - Binary; L-R; Subtraction.
- * Binary; L-R; Multiplication.

- `/` Binary; L-R; Division.
- `++` Unary; L-R; Assignment with increment.
- `--` Unary; L-R; Assignment with decrement.
- `~` Unary; L-R; Random integer generation.
- `@` Unary; L-R; Type checker.
- `#` Unary; L-R; Global variable indicator.
- `$` Unary; L-R; CardEntity indicator.

3.3.2 Expressions

Primary expressions may consist of identifiers, integer/boolean/card constants, string literals (e.g. "hello"), and list literals (e.g. [1, 2, 3]). Expressions may also be derived from operations, using the operators listed above, on one or two sub-expressions. Finally, expressions may also be derived from function calls.

3.3.3 Function Calls

Function call syntax is as follows:

- *postfix-expression* (*argument-expression-list*)

An *argument-expression-list* is a comma-separated list of expressions passed to the function (which undergoes applicative order evaluation). The function may return any value, which can then be evaluated as an expression.

3.3.4 Assignment

Assignment is handled in a standard fashion. The left-hand argument to assignment operators must be variable locations to which the evaluated right-hand argument of the operator is assigned.

3.4 Statements

A statement is a complete instruction to the computer. Except as indicated, statements are executed in sequence. A statement can be an *expression-statement*, a *selection-statement*, an *iteration-statement*, or a *jump-statement*.

3.4.1 Expression Statements

Expression statements consist of an expression terminated by a semicolon: The expression may have side effects or return a value. If it returns the value, it is discarded.

3.4.2 Selection Statements

Selection statements define branching in PCGSL. These statements select a set of statements to execute based on the evaluation of an expression. The only selection statement PCGSL supports is the if/else statement:

- *if(expression) {statement-list}*
- *if (expression) {statement-list} else {statement-list}*

The controlling expression must have a boolean type. Returning the wrong type will cause a runtime error.

3.4.3 Iteration Statements

An iteration statement repeatedly executes a list of statements, so long as its controlling expression returns true after each pass. The only iteration statement PCGSL supports is the while statement:

- *while(expression) {statement-list}*

The controller expression must be boolean type. For the while loop, the controller is executed before each execution of the body's statement-list.

3.4.4 Jump Statements

Jump statements cause unconditional transfer of control. We currently support both break and return statements, which appear followed by a semicolon. Break only has meaning inside iteration statements, break passing control to the statement immediately following the iteration statement. Return ends the currently executing function and returns the value of the expression. Since PCGSL functions have no return type, no type checking is necessary.

Chapter 4

Project Plan

We describe here the process used for the planning, specification, development and testing of PCGSL.

4.1 Planning & Specification

After deciding on the language, we spent a good deal of time coming up with the specifics, like how we wanted to refer to cards, players, how to control the flow of a program, how to know when the game is over, whether to allow various language features, etc. This is all reflected in our LRM.

After creating our LRM, we conferred with the TA to get some feedback. In accordance with some suggestions by the TA, we left out some features we wanted but felt would be too painful to implement, such as floating point numbers and a few other things. As we made revisions, we went back to discuss things with the TA a couple more times to straighten out some specific issues with our language. Namely, we wanted to make sure the language was possible to implement yet at the same time not too simplistic.

During this time our language went through several changes, such as syntax changes, what a CardEntity object encompasses, how to handle types, and so on. Of course, during development as well, we had to keep making slight modifications to the language if something was not as easy as we thought, or if another option presented itself that ended up being more intuitive.

4.2 Development & Testing

Development began with setting up the version control system on Google code, followed by creation of code based on the examples from class as templates. We referred to examples from class, especially MicroC, for help, but of course our language goes far beyond what MicroC offers, so we had to make extensive modifications. We knew that an interpreted language would be completely doable in OCAML and we wanted to stick with the one development language, so we decided to create an interpreter.

In order to create the interpreter, we decided to stick with the suggestion of using a Parser to create an AST even though that is not strictly necessary. Thus we laid down the code for Scanner, Parser, and AST, as well as a regression test in the form of a printer, which takes the AST and prints out each piece of it, to verify that the input is parsed in a predictably deterministic fashion. Laying down this code of course revealed a few more changes to be done in our language to remove ambiguities and such.

Next came the interpreter. The interpreter was also based on the MicroC interpreter at first, but we made many changes to it to fit our needs and to expand the number of expressions and statements to fit what we wanted. Also since MicroC only has 1 type (integers) while we have several (integers, strings, Cards, CardEntities, Lists), a lot of type checking had to be added. The coding of the interpreter brought about a great deal more discussion about changes to be made in the language. Some expressions and types we originally used turned out to be unnecessary, we discovered that we were missing some useful operators, and so on. So we were continually having discussions and making edits to our language.

For testing, we wrote up a suite of test programs, basic games, that as a whole use the full spectrum of language features. In addition, we kept the printer up to date to be able to check that the Parser and AST were updated correctly each time we made changes to our language during development. During testing of course we discovered a few unanticipated or forgotten issues from the development phase, so the last push for development was actually a tight cycle of testing and development.

4.3 Programming Style Guide

The team tried its best to stick to a good set of style points in writing the language. This being OCAML, we had to adopt slightly different practices than we were used to for C-like

languages. However we maintained a mostly uniform body of code style-wise, with breaches allowed when said breaches allowed for more clarity.

Let in

Since the `let in` construct is akin to a function declaration at times, and a variable assignment at other times, it was important to put each `let in` on a new line. For longer blocks, the `in` was placed on a line by itself at the end of the block, while shorter `let in` statements were allowed to be on a single line.

Indentation

We treated certain expressions in OCAML as nested blocks, and indented them. These included `let...in`, `match` statements, `if/else` statements. Indentation was of utmost importance for us, because with OCAML it is very easy to confuse different blocks, such as nested `let` or `match` expressions. So we made sure that any lines contained within an expression were indented the same amount or more than the beginning of that expression. Expressions with an ending operator, such as `let in` and `begin end` were written such that the ending operator lined up with the beginning operator, so you can visually spot the span of each logical block with ease.

Line Length

Line length was not as important to us as if we had written in another language. We tried to keep lines under 100 characters in length, but more important to us was indentation. Since it is very easy to lose track of, e.g., which `match` cases belong to which `match` operator with bad indentation, we made indentation a higher priority than keeping lines under a certain length.

Match

The `match` operator, which we used extensively in our code, maintains the guideline of keeping each case on a separate line so that they won't get confused. Similarly, in the AST and other places that use multiple cases, we stuck to using a single line per case, except for very simplistic cases like defining the different types of binary operators in the AST. There,

we allowed multiple cases per line, since each case is a single word and self-explanatory (e.g., Add, Sub, Equal, Concat, etc.)

Raising Exceptions for Default Cases

In many of the default _ match cases, we raise exceptions detailing what was wrong and what was expected. Of course, where the default case is desired, we do not raise any exception, unless it is a special non-error exception.

4.4 Project timeline

- September 17: Began working on different proposals for our own individual languages.
- September 23: Decided on a card game simulation language. Spent some time thinking about the language.
- October 20: Talked to TA and got feedback about proposal, beginning some detailed thought about the language features.
- October 21: First draft of Language Reference Manual. Spent more time thinking about the details and features of the language.
- November 14: Checked in initial versions of Scanner, Parser, and AST.
- November 25: Met with TA again and getting more feedback about some of the decisions we made about how we're implementing the language. That we wanted to make it interpreted, that we want to use such data types, and so on. After we got feedback, revised several features. Revisions of Scanner, Parser, and AST.
- December 13: Initial work on interpreter.
- December 19: Project due.

4.5 Roles and Responsibilities

Enrique Henestroza: Scanner, Parser, AST, stdlib, test programs

Yuriy Kagan: Interpreter, test programs

Andrew Shu: Interpreter, test programs

Peter Tsonev: Interpreter, presentation

4.6 Software Development Environment

Tools: Ocaml yacc, Ocaml lex, Subversion (Google code), GNU Make, bash

Languages: OCAML

4.7 Project Log

The project log generated by Subversion is located in the appendix as Changelog. Our usernames are:

Enrique Henestroza ehenestroza

Yuriy Kagan yuriy.kagan

Andrew Shu ans2120@columbia.edu and talklittle

Peter Tsonev pvt2101

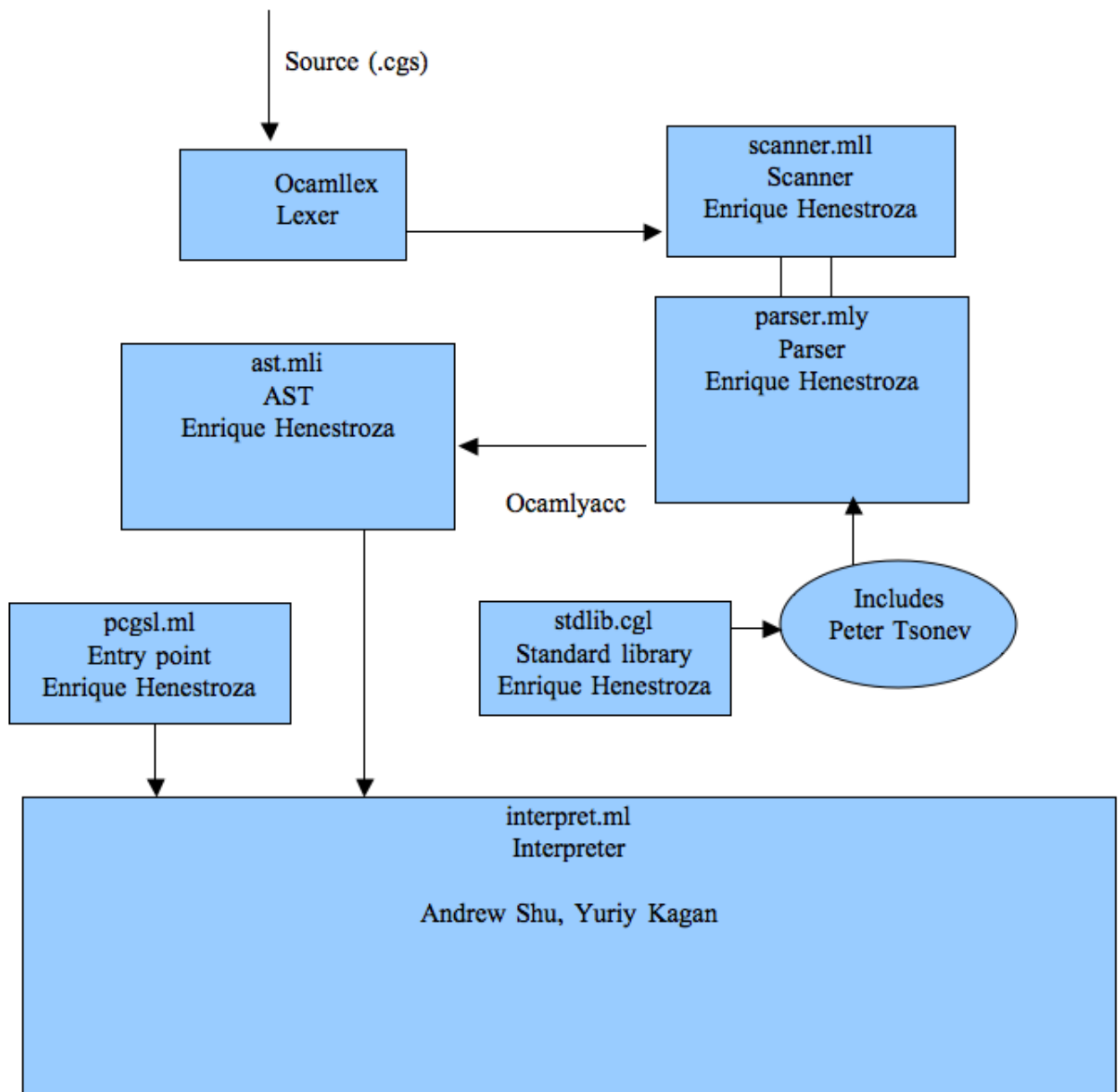
Chapter 5

Architectural Design

PCGSL is an interpreted language. As with most languages, it features a Scanner and Parser. Although it is said that an AST is not strictly required, we chose to use the Parser to generate an AST so that the interpreter could be worked on with little knowledge of the syntax and without having to keep up with every change in the underlying syntax.

Things like type checking were initially incorporated into the Parser and AST, and things such as declaring variables and functions required types to be associated with them. Later on we modified the language to use more flexible typing, where nearly all type checking is done in the interpreter. The final result of PCGSL comes after ongoing relocations of semantic logic between the interpreter and the parser and AST.

The next page contains a block diagram showing the major components of our language.



Chapter 6

Test Plan

The test plan we designed for PCSGL was specifically developed to test every aspect of the language in as much isolation from other features as possible. This was done so that we could discover any problem as quickly as possible, isolate it, and fix it. Every major language feature was included in our test suite, paying special attention to edge cases (such as divide by 0, null-value variables, etc). The specific features tested included, but were not limited to:

- Boolean Operators (with all data types)
- Arithmetic Operators
- Main-block level control flow (Start, Play, WinningCondition blocks)
- Function-level control flow
- Local vs Global scope namespaces
- General control flow (If, Else, While)
- Input/Output operators
- Variable declaration
- Variable assignment, including dynamic typing
- List creation

- List assignment, including all data types
- List operators append, list length
- GetType operator for all datatypes
- Random number generation
- Several large, sophisticated tests for Card and CardEntity functionality including
- CardEntity initialization
- Card transfer
- Card literals
- Manipulation of CardEntity and Card objects, ensuring correct functionality

We had a total of 20 tests, each of which were run on every build of PCGSL to ensure no bugs crept into the code base. We used a bash script to compare the results of the tests with expected results and corrected any errors before continuing to work on new features. This method of system turned out to be very efficient at spotting errors early before they became more difficult to debug and fix.

Here are a few small representative programs from our test suite:

I/O test:

```

Include
{
}
CardEntities
{
    deck;
}
Globals
{
}
Start
{
    var str;

    >>str;
    <<str;
}
Play
{
}
WinCondition
{
    .....

```

Array declaration and assignment test:

```

...
Start
{
    var stringList;
    var intList;
    var cardList;
    var cardEntityList;
    var boolList;

    stringList = ["a", "b", "c"];
    intList = [1,2,3];
    cardList = [H2, H3, H4];
    cardEntityList = [$deck, $player1,
$player2, $player3];
    boolList = [false, true, false];
    <<stringList[0]^stringList[1]^stringL
ist[2];
    <<"^intList[0]^intList[1]^intList[2]
";
    <<"^cardList[0]^cardList[1]^cardList
[2];
    <<"^cardEntityList[0]^cardEntityList
[1]^cardEntityList[2];
    <<"^boolList[0]^boolList[1]^boolList
[2];
}

```

Print Deck test:

```

Start
{
    var i;

    i = 0;
    while (i < |$deck|)
    {
        << $deck[i];

        i++;
    }
}

```

To divide the work for the test suite, each of us implemented the tests that corresponded to the features that person developed. For example, I implemented main-block control flow, so I added that specific test to the suite. Also many of the tests were written before we wrote any code, during our meetings about language design.

Chapter 7

Lessons Learned

7.1 Enrique Henestroza

This project was a great learning experience. Although the implementation of our interpreter proved to be an interesting and challenging task, I founded that the more important lessons arose in our semester-long, ongoing debate about the design of our language. Reaching a consensus between team members proved to be a beneficial approach instead of the selection of a team leader, since it forced us to provide justifications and persuasive arguments for our design ideas. Ultimately, we arrived at a design that was focused and geared toward a particular audience, without being bloated with extraneous features that could potentially increase the learning curve for programming in our language. My advice to future students in this class would be to focus on keeping the design, simple, flexible, and elegant. Additionally, closely scrutinize your early design decisions in order to avoid making mistakes that will hurt you when you go to implement your language down the road.

7.2 Yuriy Kagan

The PCGSL project was a fascinating experiance for me, since writing a compiler from scratch is a very different project from almost any other CS assignments. For one, the complexity of a compiler is much greater than the typical undergrad project. More importantly, this project let me undestand the structure of programming languages in a very different perspective. I think in the long run this will make me a better programmer, and appreciate the differences between different programming languages in a much deeper way.

7.3 Andrew Shu

- How to use OCAML. It's fun learning new languages.
- Try to anticipate changes that you'll make further down and don't code things too deeply before having to delete large pieces of code. Better yet, just start coding early on so that you can encounter as many problems as you can early on.
- Getting the ball rolling is hard with long deadlines. It's good to set milestones.
- It's good to try to get any piece of code down even if it's bad. That way it's easier for your brain to think up alternatives. Also write placeholders so you see the holes in the program.

7.4 Peter Tsonev

- The hardest part is finalizing the set of language features that you want in a program.
- When working in a group designing something complex like a language, a huge bulk of the time will be spent discussing and changing things that you had decided on earlier.
- It is important to try to implement the fundamental features of the language early on so you know what to keep and what to tweak.
- Expect to have to deal with many unexpected situations that come up only when doing the development and implementation phase.

Chapter 8

Appendix

The various source files, test files, etc. are attached to the end of this document.

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var a;
    var b;

    a=2;
    b=5;

    <<a+b;
    <<a*b;
    <<a-b;
    <<b-a;
    <<a/b;
    <<b/a;
    <<a/0;

}

Play
{
}

WinCondition
{
    return;
}
```



```

(* Simple binary operators *)
type op =
  Add | Sub | Mult | Div | Equal | Neq | Less |
  Leq | Greater | Geq | And | Or | Concat

(* Simple indicator of scope for variables found in expressions *)
type scope =
  Global (* "Global" scope, which means that this var is a global variable *)
  | Local (* "Local" scope, which means that this var is a local variable *)
  | Entity (* "Entity" scope, which means that this var is a CardEntity *)

(* Simple indicator of type for variable declarations *)
type t =
  Int (* "int" type *)
  | StringType (* "string" type *)
  | Bool (* "boolean" type *)
  | Card (* "Card" reference type *)
  | CardEntity (* "CardEntity" reference type *)
  | ListType (* "list" type. elements of a list can be expressions. *)

(* Variable used in an expression, contains the id and scope of the variable *)
(* Also can be a "GetIndex", which is a list dereference with string being *)
(* the list variable, and expr being the expression within the brackets. *)
type varexp =
  VarExp of string * scope (* Used by interpreter to store CardEntity refs *)
  | GetIndex of string * scope * expr

(* The expression type *)
and expr =
  Null (* The null type, comes from the "null" keyword *)
  | Variable of varexp
  | IntLiteral of int (* An "int" literal. Needs to be coerced to int in interpreter *)
  | StringLiteral of string (* A "string" literal *)
  | BoolLiteral of bool (* A "bool" literal. Needs to be coerced to bool in interpreter *)
  | CardLiteral of string (* A "card" reference literal, e.g. H2, DQ, S10 *)
  | ListLiteral of expr list (* The list literal, whose items can each be *)
    (* expressions, so type checking needs to occur *)
    (* in the interpreter. *)
  | Binop of expr * op * expr
  | Rand of expr (* The random operator, e.g. ~1, ~(a + b). The interpreter *)
    (* needs to check that its expression evaluates to "int" *)
  | Assign of varexp * expr (* Assignment of an expression to a variable *)
  | Append of expr * expr (* Appending to a list variable *)
  | GetType of expr (* Returns the string description of the type of expr *)
  | ListLength of expr (* Should evaluate to the length of a list *)
  | Transfer of varexp * expr (* The transfer operator, e.g. $player1 <- H1. *)
    (* The interpreter needs to check that the lhs *)
    (* evaluates to CardEntity and rhs evaluates *)
    (* to Card *)
  | Call of string * expr list (* The function call where string is its id *)
    (* and the list of expressions is the list of *)
    (* actual arguments *)
  | Noexpr (* Could appear in the "for(;;)" or the "return" construction *)

type stmt =
  Break (* Should break out of the current for/while loop *)
  | Print of expr (* Prints out the contents of expr. The interpreter should *)
    (* checks that the expr evaluates to a string *)
  | Read of varexp (* Reads in standard input into the variable *)
  | Expr of expr
  | Return of expr (* Returns from the current function. Interpreter should *)
    (* check that expr evaluates to the return type of the *)
    (* function. *)
  | If of expr * stmt list * stmt list (* If statement. the expr should *)
    (* evaluate to bool type. The first *)
    (* stmt list is executed when the *)
    (* expr is true, otherwise execute *)

```

```

(* the second stmt list *)
(* | For of expr * expr * expr * stmt list Expressions are, in order, the *)
(*   (* initialization, truth condition *)
(*   (* and finally update step *)
| While of expr * stmt list (* As long as expr is true, execute stmt list *)
(* indefinitely. *)
| Nostmt

(* Standard function declaration *)
type func_decl = {
  fname : string;
  formals : string list;
  locals : string list;
  body : stmt list;
}

(* Include declaration, behavior is undefined for now... *)
type incl_decl = {
  includes : string list;
}

(* Card Entity declaration, contains a list of names for the entities *)
type cent_decl = {
  entities : string list;
}

(* Global variable declaration, contains a list of variable declarations *)
type glob_decl = {
  globals : string list;
}

(* Start declaration. Executed once at the beginning of the interpretation. *)
(* Should be able to break out with "return" *)
type strt_decl = {
  slocals : string list;
  sbody : stmt list;
}

(* Play declaration. Executed indefinitely as long as WinCondition returns *)
(* null. Should be able to break out with "return" *)
type play_decl = {
  plocals : string list;
  pbody : stmt list;
}

(* WinCondition declaration. Executed before each Play execution. Has a *)
(* return type of List (containing CardEntities) *)
type wcon_decl = {
  wlocals : string list;
  wbody : stmt list;
}

(* Special declarations. Contains each of the above special declarations. *)
type spec_decl = {
  incl : incl_decl;
  cent : cent_decl;
  glob : glob_decl;
  strt : strt_decl;
  play : play_decl;
  wcon : wcon_decl;
}

(* The program. Contains the special declarations and function declarations *)
type program = spec_decl * func_decl list

```

```
Include
{
}

CardEntities
{
    deck;
    player;
}

Globals
{
}

Start
{
    var a;
    var b;

    a = $deck;
    b = $player;

    <<a==b;
    <<a!=b;
    <<a==a;
    <<a!=a;
    <<a==Null;
    <<a!=Null
    <<Null==a;
    <<Null!=a;

}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var a;
    var b;

    a = $deck[0];
    b = $deck[1];

    <<a == H2;
    <<a != H2;
    <<a == H3;
    <<a != H3;

    <<a == b;
    <<a != b;

    <<a == a;
    <<a != a;

    <<a == Null;
    <<Null == a;
}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    <<1>1;
    <<1<1;
    <<1==1;
    <<1<=1;
    <<1>=1;
    <<1!=1;

    <<1>2;
    <<1<2;
    <<1==2;
    <<1<=2;
    <<1>=2;
    <<1!=2;

    <<2>1;
    <<2<1;
    <<2==1;
    <<2<=1;
    <<2>=1;
    <<2!=1;

    <<1==Null;
    <<Null==1;

    <<1 != Null;
    <<Null != 1;

}

Play
{
}

WinCondition
{
    return [];
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    <<"==">>;

    <<"=="adsadsa">>;
    <<"asdsa==">>;

    <<"adsad" == Null>>;
    <<Null == "adsa">>;

    <<"!=">>;

    <<"!="adsadsa">>;
    <<"asdsa!=">>;

    <<"adsad" != Null>>;
    <<Null != "adsa">>;

}

Play
{
}

WinCondition
{
    return [];
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var i;

    i=true;

    if(i)
    {
        <<i;
    }

    if(false)
    {
    }
    else
    {
        <<"else";
    }
}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    <<"Start Executing";
}

Play
{
    <<"Play Executing";
}

WinCondition
{
    <<"WinCondition Executing";
    return false;
}
```



```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var i;

    i=0;
    while(i<5)
    {
        <<i;
        i=i+1;
    }
}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var a;
    var b;
    var c;

    a = 1;
    b = "hi";

    c = test(a,b);

    <<c;
}

Play
{
}

WinCondition
{
    return ;
}

test(var a, var b)
{
    <<a;
    <<b;

    return 3;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var array;

    array = ["a", "b", "c"];

    <<@"string";
    <<@1;
    <<@false;
    <<@H2;
    <<@$deck;
    <<@array;
    <<@array[0];
}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
    var int;
    var string;
    var card;
    var cardEntity;
    var list;
}

Start
{

    #int = 1;
    #string = "string";
    #card = H2;
    #cardEntity = $deck;
    #list = ["a", "b"];

    <<#int;
    <<#string;
    <<#card;
    <<#cardEntity;
    <<#list[0];

    test();
}

Play
{
}

WinCondition
{
    return ;
}

test()
{
    <<#int;
    <<#string;
    <<#card;
    <<#cardEntity;
    <<#list[0];
}
```

```
// A program that implements a simple simulation of High-Card for 4 players.

Include
{
    "stdlib/stdlib.cgl";
}

CardEntities
{
    dealer;
    player0;
    player1;
    player2;
    player3;
}

Globals
{
    var players;
}

Start // Deal cards, set chips
{
    var i;
    var e;

    << "Hello and Welcome to PCGSL Highcard!";

    #players = [$player0, $player1, $player2, $player3];

    << "Shuffling deck";
    shuffle($dealer);
    << "Dealing Cards";
    i = 0;
    while (i < |#players|) {
        e = #players[i];

        // Deal five cards to the player.
        e <- $dealer[0];
        e <- $dealer[0];
        e <- $dealer[0];
        e <- $dealer[0];
        e <- $dealer[0];

        // Print out the player's hand.
        << "Player " ^ i ^ " hand: " ^ e[0] ^ " " ^ e[1] ^ " " ^ e[2]
            ^ " " ^ e[3] ^ " " ^ e[4];

        i++;
    }
}

Play
{
}

WinCondition
{
    var comp;
    var highplayer;
    var highcard;
    var card1;
    var card2;
    var card3;
    var card4;

    card1 = high_card($player0);
    << "Player 0 high card : " ^ card1;
    card2 = high_card($player1);

```

```

    << "Player 1 high card : " ^ card2;
    card3 = high_card($player2);
    << "Player 2 high card : " ^ card3;
    card4 = high_card($player3);
    << "Player 3 high card : " ^ card4;

    comp = card_compare(card1, card2);
    if (comp > 0) {
        highplayer = $player0;
        highcard = card1;
    } else {
        highplayer = $player1;
        highcard = card2;
    }
    comp = card_compare(highcard, card3);
    if (comp < 0) {
        highplayer = $player2;
        highcard = card3;
    }
    comp = card_compare(highcard, card4);
    if (comp < 0) {
        highplayer = $player3;
        highcard = card4;
    }

    << "The winner is: " ^ highplayer;
    return [highplayer];
}

// Simply returns the highest value card (by value, and ties broken by suit)
// of a player. Assumes exactly 5 cards in the player's card pile.
high_card(var e)
{
    var comp;
    var card;

    comp = card_compare(e[0], e[1]);
    if (comp > 0) {
        card = e[0];
    } else {
        card = e[1];
    }
    comp = card_compare(card, e[2]);
    if (comp < 0) {
        card = e[2];
    }
    comp = card_compare(card, e[3]);
    if (comp < 0) {
        card = e[3];
    }
    comp = card_compare(card, e[4]);
    if (comp < 0) {
        card = e[4];
    }

    return card;
}

// Returns 1 if c1 is higher, -1 if c1 is lower, 0 if equal.
card_compare(var c1, var c2) {
    var s1;
    var s2;
    var f1;
    var f2;

    s1 = cardsuit(c1);
    s2 = cardsuit(c2);
    f1 = cardface(c1);
    f2 = cardface(c2);

```

```
if (f1 > f2) {  
    return 1;  
}  
if (f1 < f2) {  
    return 0 - 1;  
}  
if (s1 > s2) {  
    return 1;  
}  
if (s1 < s2) {  
    return 0 - 1;  
}  
return 0;  
}
```

```

open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

(* return: value, globals, entities, cards *)
exception ReturnException of Ast.expr * Ast.expr NameMap.t * Ast.expr NameMap.t * Ast.expr NameMap.t
(* return: string or string list of winners, or [] for no winners *)
exception GameOverException of Ast.expr

(* seed random number generator with current time *)
let _ = Random.init (truncate (Unix.time()))
let entityData = []

(* Main entry point: run a program *)

let run (program) =
  let spec = fst(program)
  in
  let funcs = snd(program)
  in
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
    (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
    NameMap.empty funcs
  in

  (* Invoke a function and return an updated global symbol table *)
  let rec call fdecl actuals globals entities cards =

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
      | Null -> Null, env
      | Noexpr -> Noexpr, env

      | IntLiteral(i) -> IntLiteral(i), env
      | StringLiteral(i) -> StringLiteral(i), env
      | BoolLiteral(i) -> BoolLiteral(i), env
      | CardLiteral(i) -> CardLiteral(i), env

      (* Return (list of evaluated expressions), env *)
      (* Applicative order: evaluate each argument, updating env each time *)
      | ListLiteral(lis) ->
        (match lis with
         | [] -> ListLiteral([], env)
         | hd :: tl ->
          let evalhd, env = eval env hd in
          let evaltl, env = eval env (ListLiteral(tl)) in
          (match evaltl with
           | ListLiteral(lstl) -> ListLiteral(evalhd :: lstl), env
           | _ -> raise (Failure ("invalid ListLiteral construction"))))
        in

      | Binop(e1, op, e2) ->
        let v1, env = eval env e1 in
        let v2, env = eval env e2 in
        let boolean i = if i then BoolLiteral(true) else BoolLiteral(false) in
        (match v1, op, v2 with
         | IntLiteral(i1), Add, IntLiteral(i2) -> IntLiteral(i1 + i2)
         | IntLiteral(i1), Sub, IntLiteral(i2) -> IntLiteral(i1 - i2)
         | IntLiteral(i1), Mult, IntLiteral(i2) -> IntLiteral(i1 * i2)
         | IntLiteral(i1), Div, IntLiteral(i2) -> IntLiteral(i1 / i2)
         | IntLiteral(i1), Equal, IntLiteral(i2) -> boolean (i1 = i2)
         | StringLiteral(i1), Equal, StringLiteral(i2) -> boolean (i1 = i2)
         | CardLiteral(i1), Equal, CardLiteral(i2) -> boolean (i1 = i2)
         | BoolLiteral(i1), Equal, BoolLiteral(i2) -> boolean (string_of_bool i1 =
string_of_bool i2)

```



```

| IntLiteral(i1),    Neq, IntLiteral(i2)    -> boolean (i1 <> i2)
| StringLiteral(i1), Neq, StringLiteral(i2) -> boolean (i1 <> i2)
| CardLiteral(i1),   Neq, CardLiteral(i2)   -> boolean (i1 <> i2)
| BoolLiteral(i1),   Neq, BoolLiteral(i2)   -> boolean (string_of_bool i1 <> s
tring_of_bool i2)
| Null,    Equal, Null    -> boolean(true)
| Null,    Neq, Null    -> boolean(false)
| IntLiteral(i1),    Equal, Null    -> boolean(false)
| StringLiteral(i1), Equal, Null -> boolean(false)
| CardLiteral(i1),   Equal, Null    -> boolean(false)
| Variable(VarExp(id, Entity)),    Equal, Null    -> boolean(false)
| BoolLiteral(i1),   Equal, Null    -> boolean(false)
| IntLiteral(i1),    Neq, Null    -> boolean(true)
| StringLiteral(i1), Neq, Null    -> boolean(true)
| CardLiteral(i1),   Neq, Null    -> boolean(true)
| Variable(VarExp(id, Entity)),    Neq, Null    -> boolean(true)
| BoolLiteral(i1),   Neq, Null    -> boolean(true)
| Null,    Equal, IntLiteral(i2)    -> boolean(false)
| Null,    Equal, StringLiteral(i2) -> boolean(false)
| Null,    Equal, CardLiteral(i2)   -> boolean(false)
| Null,    Equal, Variable(VarExp(id, Entity)) -> boolean(false)
| Null,    Equal, BoolLiteral(i2)   -> boolean(false)
| Null,    Neq, IntLiteral(i2)    -> boolean(true)
| Null,    Neq, StringLiteral(i2) -> boolean(true)
| Null,    Neq, CardLiteral(i2)   -> boolean(true)
| Null,    Neq, Variable(VarExp(id, Entity)) -> boolean(true)
| Null,    Neq, BoolLiteral(i2)   -> boolean(true)
| IntLiteral(i1),    Less, IntLiteral(i2)    -> boolean (i1 < i2)
| StringLiteral(i1), Less, StringLiteral(i2) -> boolean (i1 < i2)
| CardLiteral(i1),   Less, CardLiteral(i2)   -> boolean (i1 < i2) (* cmp cards
as string? *)
| IntLiteral(i1),    Leq, IntLiteral(i2)    -> boolean (i1 <= i2)
| StringLiteral(i1), Leq, StringLiteral(i2) -> boolean (i1 <= i2)
| CardLiteral(i1),   Leq, CardLiteral(i2)   -> boolean (i1 <= i2) (* cmp cards
as string? *)
| IntLiteral(i1),    Greater, IntLiteral(i2)    -> boolean (i1 > i2)
| StringLiteral(i1), Greater, StringLiteral(i2) -> boolean (i1 > i2)
| CardLiteral(i1),   Greater, CardLiteral(i2)   -> boolean (i1 > i2) (* cmp ca
rds as string? *)
| IntLiteral(i1),    Geq, IntLiteral(i2)    -> boolean (i1 >= i2)
| StringLiteral(i1), Geq, StringLiteral(i2) -> boolean (i1 >= i2)
| CardLiteral(i1),   Geq, CardLiteral(i2)   -> boolean (i1 >= i2) (* cmp cards
as string? *)
| BoolLiteral(i1), And, BoolLiteral(i2) -> boolean (i1 && i2)
| BoolLiteral(i1), Or, BoolLiteral(i2)  -> boolean (i1 || i2)
| StringLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(i1 ^ i2) (* w
e want String concat, right? *)
| StringLiteral(i1), Concat, CardLiteral(i2) -> StringLiteral(i1 ^ i2) (* we
want String concat, right? *)
| StringLiteral(i1), Concat, Variable(VarExp(id, Entity)) -> StringLiteral(i1
^ id) (* we want String concat, right? *)
| StringLiteral(i1), Concat, IntLiteral(i2) -> StringLiteral(i1 ^ string_of_in
t i2) (* we want String concat, right? *)
| StringLiteral(i1), Concat, BoolLiteral(i2) -> StringLiteral(i1 ^ string_of_b
ool i2) (* we want String concat, right? *)
| CardLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(i1 ^ i2) (* we
want String concat, right? *)
| Variable(VarExp(id, Entity)), Concat, StringLiteral(i2) -> StringLiteral(id
^ i2) (* we want String concat, right? *)
| IntLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(string_of_int i1
^ i2) (* we want String concat, right? *)
| BoolLiteral(i1), Concat, StringLiteral(i2) -> StringLiteral(string_of_bool i
1 ^ i2) (* we want String concat, right? *)

| _, _, _ ->
    raise (Failure ("invalid binary operation - likely comparing two i
ncompatible types"))
), env

```

```

| Rand(e) ->
  let v, env = eval env e in
  (match v with
  | IntLiteral(i) -> IntLiteral(Random.int i), env
  | _ -> raise (Failure ("invalid argument for random operator ~. Must supply an
int.")))
)

| GetType(e) ->
  let v, env = eval env e in
  (match v with
  | Null -> StringLiteral("null")
  | IntLiteral(_) -> StringLiteral("int")
  | StringLiteral(_) -> StringLiteral("string")
  | BoolLiteral(_) -> StringLiteral("bool")
  | CardLiteral(_) -> StringLiteral("Card")
  | ListLiteral(_) -> StringLiteral("list")
  | Variable(VarExp(_, Entity)) -> StringLiteral("CardEntity")
  | _ -> raise (Failure ("internal error: unrecognized type in GetType")))
  ), env

| Variable(var) ->
  let locals, globals, entities, cards = env in
  (match var with
  | VarExp(id, scope) ->
    (match scope with
    | Local ->
      (* NameMap maps var name to (literalvalue) *)
      if NameMap.mem id locals then
        NameMap.find id locals, env
      else raise (Failure ("undeclared local variable " ^ id))
    | Global ->
      if NameMap.mem id globals then
        NameMap.find id globals, env
      else raise (Failure ("undeclared global variable " ^ id))
    | Entity ->
      if NameMap.mem id entities then
        (* return the entity variable *)
        Variable(var), env
      else raise (Failure ("undeclared CardEntity " ^ id))
    )
  | GetIndex(id, scope, index) ->
    let evalidx, env = eval env index in
    (match scope, evalidx with
    | Local, IntLiteral(i) ->
      if NameMap.mem id locals then
        (match NameMap.find id locals with
        | ListLiteral(ls) -> List.nth ls i
        | Variable(VarExp(origid, Entity)) ->
          if NameMap.mem origid entities then
            (match NameMap.find origid entities with
            | ListLiteral(ls) -> List.nth ls i
            | _ -> raise (Failure ("internal error: CardEntity "^origid^"
not storing ListLiteral"))))
          else raise (Failure ("internal error: "^id^" holding invalid ref
erence to CardEntity "^origid))
        | _ -> raise (Failure ("You can only dereference a list or CardEntit
y")))
      ), env
    else raise (Failure ("undeclared local variable " ^ id))
    | Global, IntLiteral(i) ->
      if NameMap.mem id globals then
        (match NameMap.find id globals with
        | ListLiteral(ls) -> List.nth ls i
        | Variable(VarExp(origid, Entity)) ->
          if NameMap.mem origid entities then
            (match NameMap.find origid entities with
            | ListLiteral(ls) -> List.nth ls i
            | _ -> raise (Failure ("internal error: CardEntity "^origid^"

```

```

not storing ListLiteral"))
    else raise (Failure ("internal error: "^id^" holding invalid ref
reference to CardEntity "^origid))
    | _ -> raise (Failure ("You can only dereference a list or CardEntit
y"))
    ), env
    else raise (Failure ("undeclared global variable " ^ id))
| Entity, IntLiteral(i) ->
    if NameMap.mem id entities then
        (match NameMap.find id entities with
        | ListLiteral(ls) -> List.nth ls i
        | _ -> raise (Failure ("internal error: CardEntity "^id^" not storin
g ListLiteral")))
    ), env
    else raise (Failure ("undeclared CardEntity " ^ id))
| _, _ ->
    raise (Failure ("invalid list dereference, probably using non-integer
index"))
))

| Assign(var, e) ->
    let v, (locals, globals, entities, cards) = eval env e in
    (match var with
    | VarExp(id, scope) ->
        (match scope with
        | Local ->
            if NameMap.mem id locals then
                v, (NameMap.add id v locals, globals, entities, cards)
            else raise (Failure ("undeclared local variable " ^ id))
        | Global ->
            if NameMap.mem id globals then
                v, (locals, NameMap.add id v globals, entities, cards)
            else raise (Failure ("undeclared global variable " ^ id))
        | Entity ->
            raise (Failure ("You cannot assign to a cardentity"))
        )
    | GetIndex(id, scope, index) ->
        let evalidx, env = eval env index in
        (match scope, evalidx with
        | Local, IntLiteral(i) ->
            if NameMap.mem id locals then
                let rec inserthelper ls targetindex value curr =
                    if curr = targetindex then
                        (match ls with
                        | [] -> [value]
                        | _ :: tl -> value :: tl)
                    else
                        (match ls with
                        | [] -> raise (Failure ("index out of bounds"))
                        | hd :: tl -> hd :: (inserthelper tl targetindex value (curr+1)))
                in
                (match NameMap.find id locals with
                | ListLiteral(ls) ->
                    v, (NameMap.add id (ListLiteral(inserthelper ls i v 0)) locals,
globals, entities, cards)
                | Variable(vexp) ->
                    let ret, env = eval env (Assign(vexp, v)) in ret, env
                | _ -> raise (Failure ("You can only dereference a list or CardEntit
y"))))
            else raise (Failure ("undeclared local variable " ^ id))
        | Global, IntLiteral(i) ->
            if NameMap.mem id globals then
                let rec inserthelper ls targetindex value curr =
                    if curr = targetindex then
                        (match ls with
                        | [] -> [value]
                        | _ :: tl -> value :: tl)
                    else

```

```

        (match ls with
        | [] -> raise (Failure ("index out of bounds"))
        | hd :: tl -> hd :: (inserthelper tl targetindex value (curr+1))
        )

    in
    (match NameMap.find id globals with
    ListLiteral(ls) ->
        v, (locals, NameMap.add id (ListLiteral(inserthelper ls i v 0))
globals, entities, cards)
    | Variable(vexp) ->
        let ret, env = eval env (Assign(vexp, v)) in ret, env
    | _ -> raise (Failure ("You can only dereference a list or CardEntity"))
    else raise (Failure ("undeclared global variable " ^ id))
    | Entity, IntLiteral(i) ->
        raise (Failure ("You must use the transfer operator (<-) to modify CardEntity"))
    | _, _ ->
        raise (Failure ("invalid list dereference, probably using non-integer index"))
    ))

    | ListLength(vlist) ->
        let evlist, (locals, globals, entities, cards) = eval env vlist in
        (match evlist with
        ListLiteral(ls) -> IntLiteral(List.length ls), env
        | Variable(VarExp(id, Entity)) ->
            if NameMap.mem id entities then
                (match NameMap.find id entities with
                ListLiteral(ls) -> IntLiteral(List.length ls)
                | _ -> raise (Failure ("internal error: CardEntity "^id^" not storing ListLiteral"))), env
            else raise (Failure ("undeclared CardEntity " ^ id))
        | _ -> raise (Failure ("argument to list length operator must be a list or CardEntity"))

    | Append(vlist, e) ->
        let v, env = eval env e in
        let evlist, env = eval env vlist in
        (match evlist with
        ListLiteral(ls) -> ListLiteral(ls @ [v]), env
        | _ -> raise (Failure ("trying to append an element to a non-list")))

    | Transfer(cevar, card) ->
        let evalc, env = eval env card in
        (match cevar, evalc with
        VarExp(id, Entity), CardLiteral(c) ->
            if NameMap.mem c cards then
                let locals, globals, entities, cards = env in
                (* delete Card from original CardEntity's list *)
                let rec deletehelper ls value =
                    (match ls with
                    [] -> []
                    | hd :: tl -> if hd = value then tl else hd :: (deletehelper tl value))
                in
                let oldownerlit = NameMap.find c cards in
                (match oldownerlit with
                StringLiteral(oldowner) ->
                    let entities =
                        (if NameMap.mem oldowner entities then
                        let oldownercards = NameMap.find oldowner entities in
                        (match oldownercards with
                        ListLiteral(c1) -> NameMap.add oldowner (ListLiteral(deletehelper c1 evalc)) entities
                        | _ -> raise (Failure ("internal error: CardEntity "^id^" not storing ListLiteral"))
                        else raise (Failure ("internal error: Card "^c^" invalid owner "^oldowner)))
                    )

```

```

    ) in
    (* add mapping from Card name to StringLiteral containing CardEntity
's name *)
    let cards = NameMap.add c (StringLiteral(id)) cards in
    let rec insertunique ls value =
      (match ls with
       [] -> [value]
       | hd :: tl -> if hd = value then ls else hd :: (insertunique t
l value))
    in
    (* add updated ListLiteral to new entity's list *)
    if NameMap.mem id entities then
      let entitycards = NameMap.find id entities in
      (match entitycards with
       ListLiteral(c2) ->
         StringLiteral(id), (locals, globals, NameMap.add id (ListLiter
al(insertunique c2 evalc)) entities, cards)
       | _ -> raise (Failure ("internal error: CardEntity "^id^" not stor
ing ListLiteral")))
      else raise (Failure ("Invalid CardEntity: " ^ id))
      | _ -> raise (Failure ("internal error: Card "^id^" not mapped to a Stri
ngLiteral")))
      else raise (Failure ("Invalid card name: " ^ c))
    | VarExp(id, _), CardLiteral(c) ->
      let cref, env = eval env (Variable(cevar)) in
      (match cref with
       Variable(VarExp(id2, Entity)) -> eval env (Transfer(VarExp(id2, Entity),
evalc))
       | _ -> raise (Failure ("Transfer: arguments must be cardentity <- card")))
    | GetIndex(id, _, _), CardLiteral(c) ->
      let cref, env = eval env (Variable(cevar)) in
      (match cref with
       Variable(VarExp(id2, Entity)) -> eval env (Transfer(VarExp(id2, Entity),
evalc))
       | _ -> raise (Failure ("Transfer: arguments must be cardentity <- card")))
      | _, _ -> raise (Failure ("Transfer: arguments must be cardentity <- card")))
    | Call(f, actuals) ->
      let fdecl =
        try NameMap.find f func_decls
        with Not_found ->
          raise (Failure ("undefined function " ^ f))
      in
      let actuals, env = List.fold_left
        (fun (actuals, values) actual ->
         let v, env = eval env actual in
         List.append actuals [v], values) ([], env) actuals
      in
      let (locals, globals, entities, cards) = env in
      try
        let globals, entities, cards = call fdecl actuals globals entities cards
        in BoolLiteral(false), (locals, globals, entities, cards)
      with ReturnException(v, globals, entities, cards) -> v, (locals, globals, enti
ties, cards)
    in
    (* Execute a statement and return an updated environment *)
    let rec exec env = function
      Nostmt -> env
    | Expr(e) -> let _, env = eval env e in env
    | If(e, s1, s2) ->
      let v, env = eval env e in
      let b = (match v with
               BoolLiteral(b) -> b
               | _ -> raise (Failure ("Invalid conditional expression.")))
      in
      if b then
        List.fold_left exec env (List.rev s1)
      else

```

```

    List.fold_left exec env (List.rev s2)
  | While (e, s) ->
    let rec loop env =
      let v, env = eval env e in
      let b = (match v with
        BoolLiteral(b) -> b
        | _ -> raise (Failure ("Invalid conditional expression.")))
      in
      if b then
        loop (List.fold_left exec env (List.rev s))
      else env
    in loop env
  | Break ->
    env
  | Read(var) ->
    let input = read_line() in
    let v = (match input with
      a -> StringLiteral(a)
      | _ -> raise (Failure ("Invalid input")))
    in
    let ret, env = eval env (Assign(var, v)) in env
  | Print(e) ->
    let v, env = eval env e in
    begin
      let str = (match v with
        BoolLiteral(b) -> string_of_bool b
        | IntLiteral(i) -> string_of_int i
        | CardLiteral(c) -> "[Card: " ^ c ^ "]"
        | StringLiteral(s) -> s
        | Variable(VarExp(id, Entity)) -> "[Card Entity: " ^ id ^ "]"
        | _ -> raise (Failure ("Invalid print expression.")))
      in
      print_endline str;
      env
    end
  | Return(e) ->
    let v, (locals, globals, entities, cards) = eval env e in
    raise (ReturnException(v, globals, entities, cards))
in
(* end of statement execution *)

(* call: enter the function: bind actual values to formal args *)
let locals =
  try List.fold_left2
    (fun locals formal actual -> NameMap.add formal actual locals)
    NameMap.empty fdecl.formals actuals
  with Invalid_argument(_) ->
    raise (Failure ("wrong number of arguments to " ^ fdecl.fname))
in
let locals = List.fold_left (* Set local variables to Null (undefined) *)
  (fun locals local -> NameMap.add local Null locals)
  locals fdecl.locals
in (* Execute each statement; return updated global symbol table *)
(match (List.fold_left exec (locals, globals, entities, cards) fdecl.body) with
_, globals, entities, cards -> globals, entities, cards)

(* run: set global variables to Null; find and run "start" *)
in
(* initialize globals by reading from the globals block *)
let globals = List.fold_left
  (fun globals vdecl -> NameMap.add vdecl Null globals)
  NameMap.empty spec.glob.globals
in
(* initialize entities by reading from CardEntities block *)
let entities = List.fold_left
  (fun entities vdecl -> NameMap.add vdecl (ListLiteral([])) entities)
  NameMap.empty spec.cent.entities
in
(* initialize the cards symbol table to point to the first CardEntity *)

```

```

let firstentity =
  (match spec.cent.entities with
  | hd :: _ -> hd
  | [] -> raise (Failure ("You must declare at least one CardEntity.")))
in
let deckstrings = ["C2";"C3";"C4";"C5";"C6";"C7";"C8";"C9";"C10";"CJ";"CQ";"CK";"CA";
                  "D2";"D3";"D4";"D5";"D6";"D7";"D8";"D9";"D10";"DJ";"DQ";"DK";"DA";
                  "H2";"H3";"H4";"H5";"H6";"H7";"H8";"H9";"H10";"HJ";"HQ";"HK";"HA";
                  "S2";"S3";"S4";"S5";"S6";"S7";"S8";"S9";"S10";"SJ";"SQ";"SK";"SA"]
in
let cards = List.fold_left
  (fun cards vdecl -> NameMap.add vdecl (StringLiteral(firstentity)) cards)
  NameMap.empty deckstrings
in
(* Add the cards to the first CardEntity too. they map to each other. *)
let deckcards =
  ListLiteral(List.fold_left
    (fun acc cardstring -> CardLiteral(cardstring) :: acc) [] (List.rev deckstrings))
in
let entities = NameMap.add firstentity deckcards entities in
try
  let startDecl = { fname = "Start";
                    formals = [];
                    locals = spec.strt.slocals;
                    body=spec.strt.sbody }
  in
  let func_decls = NameMap.add "Start" startDecl func_decls
  in
  let startDecl = { fname = "Play";
                    formals = [];
                    locals = spec.play.plocals;
                    body=spec.play.pbody }
  in
  let func_decls = NameMap.add "Play" startDecl func_decls
  in
  let startDecl = { fname = "WinningCondition";
                    formals = [];
                    locals = spec.wcon.wlocals;
                    body=spec.wcon.wbody }
  in
  let func_decls = NameMap.add "WinningCondition" startDecl func_decls
  in
  let (globals, entities, cards) =
    call (NameMap.find "Start" func_decls) [] globals entities cards
  in
  let rec loop a (globals, entities, cards) =
    let (globals, entities, cards) =
      call (NameMap.find "Play" func_decls) [] globals entities cards
    in
    try
      let (globals, entities, cards) =
        call (NameMap.find "WinningCondition" func_decls) [] globals entities
        cards
      in
      in (globals, entities, cards)
      with ReturnException(v, globals, entities, cards) ->
        (match v with
        | Null -> loop a (globals, entities, cards)
        | _ -> raise (GameOverException (v)))
    in
    loop "blah" (globals, entities, cards)
  with
  Not_found -> raise (Failure ("did not find the start() function"))
  | GameOverException(winners) ->
    print_endline "Game over!"; exit 0

```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var str;

    >>str;
    <<str;
}

Play
{
}

WinCondition
{
    return ;
}
```



```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var list;

    list = [1,2,3];

    <<"^list[0]^list[1]^list[2];
    list = list::4;

    <<"^list[0]^list[1]^list[2]^list[3];
}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
    player1;
    player2;
    player3;
}

Globals
{
}

Start
{
    var stringList;
    var intList;
    var cardList;
    var cardEntityList;
    var boolList;

    stringList = ["a", "b", "c"];
    intList = [1,2,3];
    cardList = [H2, H3, H4];
    cardEntityList = [$deck, $player1, $player2, $player3];
    boolList = [false, true, false];

    <<stringList[0]^stringList[1]^stringList[2];

    <<"^intList[0]^intList[1]^intList[2];

    <<"^cardList[0]^cardList[1]^cardList[2];

    <<"^cardEntityList[0]^cardEntityList[1]^cardEntityList[2];

    <<"^boolList[0]^boolList[1]^boolList[2];
}

Play
{
}

WinCondition
{
    return ;
}
```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var listLong;
    var listShort;
    var listNull;

    listLong = [1, 2, 3, 4, 5, 6, 7 ,8, 9, 10, 11, 12, 1, 2];
    listShort = [1,2];
    listNull = [];

    <<|listLong|;
    <<|listShort|;
    <<|listNull|;
}

Play
{
}

WinCondition
{
    return ;
}
```

```

OBSJS = parser.cmo scanner.cmo printer.cmo interpret.cmo pcgsl.cmo
LIBS = unix.cma

TESTS = \
arith1 \
arith2 \
fib \
for1 \
func1 \
gcd \
global1 \
hello \
if1 \
if2 \
if3 \
if4 \
ops1 \
var1 \
while1

TARFILES = Makefile testall.sh scanner.mll parser.mly \
          ast.mli interpret.ml printer.ml pcgsl.ml \
          $(TESTS:%=tests/test-%.mc) \
          $(TESTS:%=tests/test-%.out)

pcgsl : $(OBSJS)
        ocamlc -o pcgsl $(LIBS) $(OBSJS)

.PHONY : test
test : pcgsl testall.sh
        ./testall.sh

scanner.ml : scanner.mll
        ocamllex scanner.mll

parser.ml parser.mli : parser.mly
        ocamlyacc -v parser.mly

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

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

pcgsl.tar.gz : $(TARFILES)
        tar czf ./pcgsl.tar.gz $(TARFILES:%=./%)

.PHONY : clean
clean :
        rm -f pcgsl parser.ml parser.mli scanner.ml testall.log *.cmo *.cmi *.out pars
er.output

# Generated by ocamldep *.ml *.mli
interpret.cmo: ast.cmi
interpret.cmx: ast.cmi
pcgsl.cmo: scanner.cmo parser.cmi interpret.cmo
pcgsl.cmx: scanner.cmx parser.cmx interpret.cmx
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
printer.cmo: ast.cmi
printer.cmx: ast.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmi

```

```

%{ open Ast %}

%token SEMI LBRACK RBRACK LPAREN RPAREN LBRACE RBRACE BAR COMMA
%token PLUS PLUSEQ MINUS MINUSEQ TIMES TIMESEQ DIVIDE DIVIDEEQ PLUSTWO MINUSTWO
%token TILDE TRANSFER ASSIGN EQ NEQ LT GT LEQ GEQ AND OR CONCAT APPEND
%token GLOBALVAR ENTITYVAR
%token PRINT READ
%token RETURN IF ELSE FOR WHILE BREAK CONTINUE
%token BOOL INT STRING CARD CARDENTITY LIST VAR
%token CARDENTITIES GLOBALS INCLUDE PLAY START WINCONDITION
%token NULL GETTYPE
%token <bool> TRUE FALSE
%token <int> INTLITERAL
%token <string> STRINGLITERAL
%token <string> CARDLITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%left CONCAT
%left AND OR
%right ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ APPEND
%right READ
%right PRINT
%left TRANSFER
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left PLUSTWO MINUSTWO
%left BOOL INT STRING CARD CARDENTITY LIST VAR
%left TILDE GETTYPE
%left GLOBALVAR ENTITYVAR

%start program
%type <Ast.program> program

%%

program:
    sdecl funcs_list { $1, List.rev $2 }

sdecl:
    INCLUDE LBRACE idecl_list RBRACE
    CARDENTITIES LBRACE cdecl_list RBRACE
    GLOBALS LBRACE vdecl_list RBRACE
    START LBRACE vdecl_list stmt_list RBRACE
    PLAY LBRACE vdecl_list stmt_list RBRACE
    WINCONDITION LBRACE vdecl_list stmt_list RBRACE
    { { incl = { includes = List.rev $3 };
      cent = { entities = List.rev $7 };
      glob = { globals = List.rev $11 };
      strt = { slocals = List.rev $15;
              sbody = List.rev $16 };
      play = { plocals = List.rev $20;
              pbody = List.rev $21 };
      wcon = { wlocals = List.rev $25;
              wbody = List.rev $26 } } } }

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

fdecl:
    ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
    { { fname = $1;
      formals = $3;
      locals = List.rev $6;

```

```

    body = List.rev $7 } }

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

formal_list:
    vdecl { [$1] }
    | formal_list COMMA vdecl { $3 :: $1 }

idecl_list:
    /* nothing */ { [] }
    | idecl_list idecl { $2 :: $1 }

idecl:
    STRINGLITERAL SEMI { $1 }

cdecl_list:
    /* nothing */ { [] }
    | cdecl_list cdecl { $2 :: $1 }

cdecl:
    ID SEMI { $1 }

vdecl_list:
    /* nothing */ { [] }
    | vdecl_list vdecl SEMI { $2 :: $1 }

vdecl:
    VAR ID { $2 }

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

stmt:
    expr SEMI { Expr($1) }
    | PRINT expr SEMI { Print($2) }
    | READ var SEMI { Read($2) }
    | BREAK SEMI { Break }
    | RETURN expr_opt SEMI { Return($2) }
    | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE /* %prec NOELSE */
      { If($3, $6, []) }
    | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list RBRACE
      { If($3, $6, $10) }
    /* | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN */
    /* | LBRACE stmt_list RBRACE */
    /* { For($3, $5, $7, $10) } */
    | WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE { While($3, $6) }

expr_opt:
    /* nothing */ { Noexpr }
    | expr { $1 }

expr:
    NULL { Null }
    | TRUE { BoolLiteral(true) }
    | FALSE { BoolLiteral(false) }
    | INTLITERAL { IntLiteral($1) }
    | CARDLITERAL { CardLiteral($1) }
    | STRINGLITERAL { StringLiteral($1) }
    | var { Variable($1) }
    | TILDE expr { Rand($2) }
    | GETTYPE expr { GetType($2) }
    | expr PLUS expr { Binop($1, Add, $3) }
    | expr MINUS expr { Binop($1, Sub, $3) }
    | expr TIMES expr { Binop($1, Mult, $3) }
    | expr DIVIDE expr { Binop($1, Div, $3) }
    | expr EQ expr { Binop($1, Equal, $3) }

```

```

| expr NEQ      expr { Binop($1, Neq, $3) }
| expr LT       expr { Binop($1, Less, $3) }
| expr LEQ      expr { Binop($1, Leq, $3) }
| expr GT       expr { Binop($1, Greater, $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) }
| var PLUSEQ    expr { Assign($1, Binop(Variable($1), Add, $3)) }
| var MINUSEQ   expr { Assign($1, Binop(Variable($1), Sub, $3)) }
| var TIMESEQ   expr { Assign($1, Binop(Variable($1), Mult, $3)) }
| var DIVIDEEQ  expr { Assign($1, Binop(Variable($1), Div, $3)) }
| var PLUSTWO   { Assign($1, Binop(Variable($1), Add, IntLiteral(1))) }
| var MINUSTWO  { Assign($1, Binop(Variable($1), Sub, IntLiteral(1))) }
| var ASSIGN    expr { Assign($1, $3) }
| expr APPEND   expr { Append($1, $3) }
| BAR expr BAR  { ListLength($2) }
| var TRANSFER  expr { Transfer($1, $3) }
| LBRACK list_opt RBRACK { ListLiteral($2) }
| ID LPAREN    actuals_opt RPAREN { Call($1, $3) }
| LPAREN expr  RPAREN { $2 }

var:
  ID { VarExp($1, Local) }
| GLOBALVAR ID { VarExp($2, Global) }
| ENTITYVAR ID { VarExp($2, Entity) }
| ID LBRACK expr RBRACK { GetIndex($1, Local, $3) }
| GLOBALVAR ID LBRACK expr RBRACK { GetIndex($2, Global, $4) }
| ENTITYVAR ID LBRACK expr RBRACK { GetIndex($2, Entity, $4) }

list_opt:
  /* nothing */ { [] }
| list_ { List.rev $1 }

list_:
  expr { [$1] }
| list_ COMMA expr { $3 :: $1 }

actuals_opt:
  /* nothing */ { [] }
| actuals_list { List.rev $1 }

actuals_list:
  expr { [$1] }
| actuals_list COMMA expr { $3 :: $1 }

```

pcgs1.sh

```
#!/bin/sh
```

```
OCAMLRUNPARAM="p,b"
```

```
export OCAMLRUNPARAM
```

```
./pcgs1 $1
```



```
// A program that implements a simple simulation of Poker for 4 players.
// Only recognizes single pairs, three-of-a-kinds, and four-of-a-kinds.

Include
{
    "stdlib/stdlib.cgl";
}

CardEntities
{
    dealer;
    player0;
    player1;
    player2;
    player3;
}

Globals
{
    var players;
}

Start // Deal cards, set chips
{
    var i;
    var e;

    << "Hello and Welcome to PCGSL Poker!";

    #players = [$player0, $player1, $player2, $player3];

    << "Shuffling deck.";
    shuffle($dealer);
    << "Dealing Cards.";
    i = 0;
    while (i < |#players|) {
        e = #players[i];

        // Deal five cards to the player.
        e <- $dealer[0];
        e <- $dealer[0];
        e <- $dealer[0];
        e <- $dealer[0];
        e <- $dealer[0];

        // Print out the player's hand.
        << "Player " ^ i ^ "'s hand: " ^ e[0] ^ " " ^ e[1] ^ " " ^ e[2]
            ^ " " ^ e[3] ^ " " ^ e[4];

        i++;
    }
}

Play
{
}

WinCondition
{
    var i;
    var comp;
    var highplayers;
    var highhand;
    var best0;
    var best1;
    var best2;
    var best3;

    best0 = best_hand($player0);

```

```

    << "Player 0 best hand : " ^ best0[0] ^ " of face " ^ best0[1];
    best1 = best_hand($player1);
    << "Player 1 best hand : " ^ best1[0] ^ " of face " ^ best1[1];
    best2 = best_hand($player2);
    << "Player 2 best hand : " ^ best2[0] ^ " of face " ^ best2[1];
    best3 = best_hand($player3);
    << "Player 3 best hand : " ^ best3[0] ^ " of face " ^ best3[1];

    comp = hand_compare(best0, best1);
    if (comp > 0) {
        highplayers = [$player0];
        highhand = best0;
    } else {
        if (comp < 0) {
            highplayers = [$player1];
            highhand = best1;
        } else {
            highplayers = [$player1, $player2];
            highhand = best0;
        }
    }

    comp = hand_compare(highhand, best2);
    if (comp < 0) {
        highplayers = [$player2];
        highhand = best2;
    } else {
        if (comp == 0) {
            highplayers = highplayers :: $player2;
        }
    }

    comp = hand_compare(highhand, best3);
    if (comp < 0) {
        highplayers = [$player3];
        highhand = best3;
    } else {
        if (comp == 0) {
            highplayers = highplayers :: $player3;
        }
    }

    << "The winners are:";
    i = 0;
    while (i < |highplayers|) {
        << highplayers[i];
        i++;
    }

    return highplayers;
}

// Returns [type, val] where type is an int for the type of hand (4 = four
// of a kind, 3 = 3 of a kind, 2 = best pair, 1 = high card), and val is the
// value of that type of hand (14 = Aces, 13 = Kings, ..., 2 = 2's). Assumes
// a 5-card CardEntity is given.
best_hand(var e) {
    var list;
    var i;
    var highpairval;
    var highsingval;

    list = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

    list[cardface(e[0])] += 1;
    list[cardface(e[1])] += 1;
    list[cardface(e[2])] += 1;
    list[cardface(e[3])] += 1;

```

```
list[cardface(e[4])] += 1;

i = 2;
while (i <= 14) {
    if (list[i] == 4) {
        return [4, i];
    }

    if (list[i] == 3) {
        return [3, i];
    }

    if (list[i] == 2) {
        highpairval = i;
    }

    if (list[i] == 1) {
        highsingval = i;
    }

    i++;
}

if (highpairval != null) {
    return [2, highpairval];
}

return [1, highsingval];
}

// Takes two hands of the form [type, val] as described in the high_hand
// function. Returns 1 if the first hand is better, -1 if the second hand is
// better, and 0 if they are equal.
hand_compare(var h1, var h2) {

    if (h1[0] > h2[0]) {
        return 1;
    }

    if (h2[0] > h1[0]) {
        return 0 - 1;
    }

    if (h1[1] > h2[1]) {
        return 1;
    }

    if (h2[1] > h1[1]) {
        return 0 - 1;
    }

    return 0;
}
```

```
Include
{
    "stdlib/stdlib.cgl";
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var i;

    i = 0;
    while (i < |$deck|) {
        << $deck[i];

        i++;
    }
}

Play
{
}

WinCondition
{
    return [];
}
```

```

open Ast

let rec tabs i = match i with
  0 -> ""
  | x -> "\t" ^ tabs (x - 1)

let string_of_op op = match op with
  Add      -> "+"
  | Sub     -> "-"
  | Mult    -> "*"
  | Div     -> "/"
  | Equal   -> "=="
  | Neq     -> "!="
  | Less    -> "<"
  | Leq     -> "<="
  | Greater -> ">"
  | Geq     -> ">="
  | And     -> "&&"
  | Or      -> "||"
  | Concat  -> "^"

let rec string_of_t t = match t with
  Int      -> "int"
  | StringType -> "string"
  | Bool     -> "bool"
  | Card     -> "Card"
  | CardEntity -> "CardEntity"
  | ListType  -> "list"

let string_of_scope scope = match scope with
  Global -> "#"
  | Local  -> ""
  | Entity -> "$"

let string_of_vardec v = match v with
  id -> "var " ^ id

let rec string_of_varexp v = match v with
  | VarExp(id, s) -> string_of_scope s ^ id
  | GetIndex(id, s, e) -> string_of_scope s ^ id ^ "[" ^ string_of_expr e ^ "]"
and string_of_expr expr = match expr with
  Null -> "null"
  | Variable(v) -> string_of_varexp v
  | IntLiteral(i) -> string_of_int i
  | StringLiteral(s) -> "\"" ^ s ^ "\""
  | BoolLiteral(b) -> string_of_bool b
  | CardLiteral(c) -> c
  | ListLiteral(el) ->
    "[" ^ String.concat ", " (List.map string_of_expr el) ^ "]"
  | Binop(e1, o, e2) ->
    "(" ^ string_of_expr e1 ^ " "
    ^ string_of_op o ^ " " ^ string_of_expr e2 ^ ")"
  | Rand(e) -> "~" ^ string_of_expr e
  | Assign(v, e) -> string_of_varexp v ^ " = " ^ string_of_expr e
  | ListLength(e) -> "|" ^ string_of_expr e ^ "|"
  | GetType(e) -> "@(" ^ string_of_expr e ^ ")"
  | Append(e1, e2) -> string_of_expr e1 ^ " :: " ^ string_of_expr e2
  | Transfer(v, e) -> string_of_varexp v ^ " <- " ^ string_of_expr e
  | Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt t stmt = tabs t ^ match stmt with
  Break -> "break;\n"
  | Print(expr) -> "<< " ^ string_of_expr expr ^ ";\n"
  | Read(var) -> ">> " ^ string_of_varexp var ^ ";\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
  | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ") {\n" ^

```

```

    String.concat "" (List.map (string_of_stmt (t+1)) s1) ^
    tabs t ^ "}" else {\n" ^
    String.concat "" (List.map (string_of_stmt (t+1)) s2) ^
    tabs t ^ "}\n"
(*| For(e1, e2, e3, s) ->
   "for (" ^ string_of_expr e1 ^ "; " ^ string_of_expr e2 ^ "; " ^
   string_of_expr e3 ^ ") {\n" ^
   String.concat "" (List.map (string_of_stmt (t+1)) s) ^
   tabs t ^ "}\n" *)
| While(e, s) -> "while (" ^ string_of_expr e ^ ") {\n" ^
   String.concat "" (List.map (string_of_stmt (t+1)) s) ^
   tabs t ^ "}\n"
| Nostmt -> ""

let string_of_strdecl id =
  "\t" ^ id ^ ";\n"

let string_of_vdecl v =
  "\t" ^ string_of_vardec v ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^
  String.concat ", " (List.map string_of_vardec fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map (string_of_stmt 1) fdecl.body) ^
  "}\n"

let string_of_sdecl_1 sname strings =
  sname ^ "\n{\n" ^
  String.concat "" (List.map string_of_strdecl strings) ^
  "}\n"

let string_of_sdecl_2 sname vars =
  sname ^ "\n{\n" ^
  String.concat "" (List.map string_of_vdecl vars) ^
  "}\n"

let string_of_sdecl_3 sname vars body =
  sname ^ "\n{\n" ^
  String.concat "" (List.map string_of_vdecl vars) ^
  String.concat "" (List.map (string_of_stmt 1) body) ^
  "}\n"

let string_of_program (spec, funcs) =
  string_of_sdecl_1 "Include" spec.incl.includes ^ "\n" ^
  string_of_sdecl_1 "CardEntities" spec.cent.entities ^ "\n" ^
  string_of_sdecl_2 "Globals" spec.glob.globals ^ "\n" ^
  string_of_sdecl_3 "Start" spec.strt.slocals spec.strt.sbody ^ "\n" ^
  string_of_sdecl_3 "PlayOrder" spec.play.plocals spec.play.pbody ^ "\n" ^
  string_of_sdecl_3 "WinningCondition" spec.wcon.wlocals spec.wcon.wbody
  ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)

let string_of_include_file (funcs) =
  String.concat "\n" (List.map string_of_fdecl funcs)

```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    <<~10;
    <<~10;
    <<~10;
    <<~10;
    <<~10;
    <<~10;
    <<~1;
    <<~0;

}

Play
{
}

WinCondition
{
    return ;
}
```

```

{ open Parser }

rule token = parse
  [ '\t' '\n' ] { token lexbuf }
| "/" "/" { comment lexbuf }
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| '[' { LBRACK }
| ']' { RBRACK }
| ';' { SEMI }
| ',' { COMMA }
| "<-" { TRANSFER }
| "++" { PLUSTWO }
| '+' { PLUS }
| "--" { MINUSTWO }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| "+=" { PLUSEQ }
| "-=" { MINUSEQ }
| "*=" { TIMESEQ }
| "/=" { DIVIDEEQ }
| "==" { EQ }
| '=' { ASSIGN }
| "!=" { NEQ }
| '<' { LT }
| "<=" { LEQ }
| '>' { GT }
| ">=" { GEQ }
| "&&" { AND }
| "||" { OR }
| "::" { APPEND }
| '|' { BAR }
| '~' { TILDE }
| '^' { CONCAT }
| '@' { GETTYPE }
| '#' { GLOBALVAR }
| '$' { ENTITYVAR }
| "<<" { PRINT }
| ">>" { READ }
| "break" { BREAK }
| "CardEntities" { CARDENTITIES }
| "continue" { CONTINUE }
| "else" { ELSE }
| "false" { FALSE(false) }
| "for" { FOR }
| "Globals" { GLOBALS }
| "if" { IF }
| "Include" { INCLUDE }
| "null" { NULL }
| "Play" { PLAY }
| "return" { RETURN }
| "Start" { START }
| "true" { TRUE(true) }
| "while" { WHILE }
| "WinCondition" { WINCONDITION }
| "var" { VAR }
| "H2" { CARDLITERAL("H2") }
| "H3" { CARDLITERAL("H3") }
| "H4" { CARDLITERAL("H4") }
| "H5" { CARDLITERAL("H5") }
| "H6" { CARDLITERAL("H6") }
| "H7" { CARDLITERAL("H7") }
| "H8" { CARDLITERAL("H8") }
| "H9" { CARDLITERAL("H9") }
| "H10" { CARDLITERAL("H10") }
| "HJ" { CARDLITERAL("HJ") }

```



```

| "HQ"           { CARDLITERAL("HQ") }
| "HK"           { CARDLITERAL("HK") }
| "HA"           { CARDLITERAL("HA") }
| "D2"           { CARDLITERAL("D2") }
| "D3"           { CARDLITERAL("D3") }
| "D4"           { CARDLITERAL("D4") }
| "D5"           { CARDLITERAL("D5") }
| "D6"           { CARDLITERAL("D6") }
| "D7"           { CARDLITERAL("D7") }
| "D8"           { CARDLITERAL("D8") }
| "D9"           { CARDLITERAL("D9") }
| "D10"          { CARDLITERAL("D10") }
| "DJ"           { CARDLITERAL("DJ") }
| "DQ"           { CARDLITERAL("DQ") }
| "DK"           { CARDLITERAL("DK") }
| "DA"           { CARDLITERAL("DA") }
| "C2"           { CARDLITERAL("C2") }
| "C3"           { CARDLITERAL("C3") }
| "C4"           { CARDLITERAL("C4") }
| "C5"           { CARDLITERAL("C5") }
| "C6"           { CARDLITERAL("C6") }
| "C7"           { CARDLITERAL("C7") }
| "C8"           { CARDLITERAL("C8") }
| "C9"           { CARDLITERAL("C9") }
| "C10"          { CARDLITERAL("C10") }
| "CJ"           { CARDLITERAL("CJ") }
| "CQ"           { CARDLITERAL("CQ") }
| "CK"           { CARDLITERAL("CK") }
| "CA"           { CARDLITERAL("CA") }
| "S2"           { CARDLITERAL("S2") }
| "S3"           { CARDLITERAL("S3") }
| "S4"           { CARDLITERAL("S4") }
| "S5"           { CARDLITERAL("S5") }
| "S6"           { CARDLITERAL("S6") }
| "S7"           { CARDLITERAL("S7") }
| "S8"           { CARDLITERAL("S8") }
| "S9"           { CARDLITERAL("S9") }
| "S10"          { CARDLITERAL("S10") }
| "SJ"           { CARDLITERAL("SJ") }
| "SQ"           { CARDLITERAL("SQ") }
| "SK"           { CARDLITERAL("SK") }
| "SA"           { CARDLITERAL("SA") }
| ['H' 'D' 'C' 'S']("J" | "Q" | "K" | "A" | "10" | ['2'-'9']) as
  lxm { CARDLITERAL(lxm) }
| ['0'-'9']+ as
  lxm { INTLITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as
  lxm { ID(lxm) }
| "\\\"[^\\"]*\\" as
  lxm { STRINGLITERAL(String.sub lxm 1 ((String.length lxm) - 2)) }
| eof { EOF }
| _ as
  char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  '\n' { token lexbuf }
| _ { comment lexbuf }

```

```

// Standard library for the PCGSL programming language

// *****
// Functions for string conversion
// *****

//
// stringtoint - returns the int representation of the given string, within the
//               given integer limits. Or null, if no int representation found.
//
// input type(s) expected: string, int, int
// output type(s): int
//
stringtoint(var s, var low, var high) {
    var j;

    if (@s != "string" || @low != "int" || @high != "int") {
        return null;
    }

    j = low;

    while (true) {
        if (j > high) {
            return null;
        }
        if (" " ^ j == s) {
            return j;
        }
        j++;
    }
}

//
// stringtocard - returns the Card representation of the given string. Or null
//               if no Card is found.
//
// input type(s) expected: string
// output type(s): Card
//
stringtocard(var s) {
    var cards;

    if (@s != "string") {
        return null;
    }

    cards = listallcards();

    j = 0;

    while (true) {
        if (j >= |cards|) {
            return null;
        }
        if (" " ^ cards[j] == s) {
            return cards[j];
        }
        j++;
    }
}

// *****
// Functions for lists
// *****

//
// listlength - returns the length of the given list.
//

```

```

// input type(s) expected: list
// output type(s): int
//
listlength(var l) {
    if (@l != "list") {
        return null;
    }

    return |l|;
}

//
// listfind - returns the index of the element in the given list matching the
//             given input item, or null if none exists. If more than one match,
//             returns the first occurring match in the list.
//
// input type(s) expected: list, var
// output type(s): int
//
listfind(var l, var e) {
    var i;

    if (@l != "list") {
        return null;
    }

    i = 0;
    while (true) {
        if (i == |l|) {
            return null;
        }
        if (l[i] == e) {
            return i;
        }
        i++;
    }
}

//
// listremove - returns a new list that has removed the given index's element
//              from the given list. If the index is out of bounds, returns
//              a new list identical to the given list.
//
// input type(s) expected: list, int
// output type(s): list
//
listremove(var oldl, var i) {
    var newl;
    var j;

    if (@oldl != "list" || @i != "int") {
        return null;
    }

    if (i < 0 || i >= |oldl|) {
        return oldl;
    }

    newl = [];
    j = 0;

    while (true) {
        if (j >= |oldl|) {
            return newl;
        }
        if (j != i) {
            newl = newl :: oldl[j];
        }
        j++;
    }
}

```

```

    }
}

//
// listreverse - returns a new list that flips the elements of the given list.
//
// input type(s) expected: list
// output type(s): list
//
listreverse(var oldl) {
    var newl;
    var i;

    if (@oldl != "list") {
        return null;
    }

    newl = [];
    i = |oldl| - 1;

    while (true) {
        if (i < 0) {
            return newl;
        }
        newl = newl :: oldl[i];
        i--;
    }
}

// *****
// Functions for Cards and CardEntities
// *****

//
// cardsuit - returns the suit value of the given Card as an int. Heart is 1,
//             Diamond is 2, Club is 3, and Spade is 4.
//
// input type(s) expected: Card
// output type(s): int
//
cardsuit(var c) {
    if (@c != "Card") {
        return null;
    }

    if (c == H2 || c == H3 || c == H4 || c == H5 || c == H6 ||
        c == H7 || c == H8 || c == H9 || c == H10 || c == HJ ||
        c == HQ || c == HK || c == HA) {
        return 1;
    }

    if (c == D2 || c == D3 || c == D4 || c == D5 || c == D6 ||
        c == D7 || c == D8 || c == D9 || c == D10 || c == DJ ||
        c == DQ || c == DK || c == DA) {
        return 2;
    }

    if (c == C2 || c == C3 || c == C4 || c == C5 || c == C6 ||
        c == C7 || c == C8 || c == C9 || c == C10 || c == CJ ||
        c == CQ || c == CK || c == CA) {
        return 3;
    }

    if (c == S2 || c == S3 || c == S4 || c == S5 || c == S6 ||
        c == S7 || c == S8 || c == S9 || c == S10 || c == SJ ||
        c == SQ || c == SK || c == SA) {
        return 4;
    }
}

```

```

//
// cardface - returns the face value of the given Card as an int. Ace is 1.
//
// input type(s) expected: Card
// output type(s): int
//
cardface(var c) {
  if (@c != "Card") {
    return null;
  }

  if (c == H2 || c == D2 || c == C2 || c == S2) {
    return 2;
  }

  if (c == H3 || c == D3 || c == C3 || c == S3) {
    return 3;
  }

  if (c == H4 || c == D4 || c == C4 || c == S4) {
    return 4;
  }

  if (c == H5 || c == D5 || c == C5 || c == S5) {
    return 5;
  }

  if (c == H6 || c == D6 || c == C6 || c == S6) {
    return 6;
  }

  if (c == H7 || c == D7 || c == C7 || c == S7) {
    return 7;
  }

  if (c == H8 || c == D8 || c == C8 || c == S8) {
    return 8;
  }

  if (c == H9 || c == D9 || c == C9 || c == S9) {
    return 9;
  }

  if (c == H10 || c == D10 || c == C10 || c == S10) {
    return 10;
  }

  if (c == HJ || c == DJ || c == CJ || c == SJ) {
    return 11;
  }

  if (c == HQ || c == DQ || c == CQ || c == SQ) {
    return 12;
  }

  if (c == HK || c == DK || c == CK || c == SK) {
    return 13;
  }

  if (c == HA || c == DA || c == CA || c == SA) {
    return 14;
  }
}

//
// listallcards - returns a list containing all Cards.
//
// input type(s) expected:

```

```

// output type(s): list of cards
//
listallcards() {
    return [H2, H3, H4, H5, H6, H7, H8, H9, H10, HJ, HQ, HK, HA,
            D2, D3, D4, D5, D6, D7, D8, D9, D10, DJ, DQ, DK, DA,
            C2, C3, C4, C5, C6, C7, C8, C9, C10, CJ, CQ, CK, CA,
            S2, S3, S4, S5, S6, S7, S8, S9, S10, SJ, SQ, SK, SA];
}

//
// containscard - returns true if the given CardEntity contains the given Card,
//                  and false otherwise.
//
// input type(s) expected: CardEntity, Card
// output type(s): boolean
//
containscard(var e, var c) {
    var i;

    if (@e != "CardEntity" || @c != "Card") {
        return null;
    }

    i = 0;

    while (true) {
        if (i >= |e|) {
            return false;
        }
        if (e[i] == c) {
            return true;
        }
        i++;
    }
}

//
// locatecard - returns the CardEntity, from a given list, that contains the
//              given Card. Returns null if no such CardEntity is found.
//
// input type(s) expected: list of CardEntities, Card
// output type(s): CardEntity
//
locatecard(var entities, var c) {
    var i;

    if (@entities != "list" || @c != "Card") {
        return null;
    }

    i = 0;

    while (true) {
        if (i >= |entities|) {
            return null;
        }
        if (@entities[i] == "CardEntity") {
            if (containscard(entities[i], c)) {
                return entities[i];
            }
        }
        i++;
    }
}

//
// shuffle - randomly reorders the Cards in the given CardEntity.
//
// input type(s) expected: CardEntity

```

```

// output type(s): null
//
shuffle(var e) {
  var r;
  var l;
  var i;
  var j;

  if (@e != "CardEntity") {
    return null;
  }

  l = [];
  i = 0;
  j = 0;

  while (i < |e|) {
    l = l :: e[i];
    i++;
  }

  while (true) {
    if (i <= 0 || j >= |e|) {
      return null;
    }
    r = ~i;
    e <- l[r];
    l = listremove(l, r);
    i--;
    j++;
  }
}

//
// transferall - transfers all Cards to the first given CardEntity from the
//                second given CardEntity.
//
// input type(s) expected: CardEntity, CardEntity
// output type(s): null
//
transferall(var e1, var e2) {
  if (@e1 != "CardEntity" || @e2 != "CardEntity" || e1 == e2) {
    return null;
  }

  while (|e2| > 0) {
    e1 <- e2[0];
  }
}

```

```

Include
// Include library files according to some path environment variable.
{
    "stdlib/stdlib.cgl";
}

CardEntities
// Card Entities are addressed with '$'
// All cards are initially located in the first Card Entity, in normal order.
{
    dealer;
    player0;
    player1;
    flop;
}

Globals // Global variables are addressed with '#'
{
    var currentPot;
    var lastBid;
    var chips;
    var players;
    var doneBidding;
    var message;
}

Start // Deal cards, set chips
{
    var i;
    var j;
    var c;
    <<"Hello World";
    //
    // initialize global variables
    //
    #currentPot = 0;
    #lastBid = 0;
    #chips = [100, 100]; // start players off with 100 chips
    #players = [$player0, $player1]; // have an array of players
    #doneBidding = [true, true];
    #message = "Please select a card.";

    // test card and randomness
    c = H2;
    c = C10;
    c = SA;
    c = D2;
    i = ~1;
    i = ~(5 + 4 / 3);

    // shuffle the deck
    //shuffle($dealer);

    // deal out 5 cards to each player
    //for (i = 0; i < 5; i++) {
    //    for (j = 0; j < listLength(#players); j++) {
    //        #players[j] <- $dealer[0];
    //    }
    //}

}

Play
// Play functionality associated with a "round" of play
{
    var i;

    <<"Calling Play Function";
    //for (i = 0; i < size(#players); i++) {
    //    play(#players[i]);

```



```

    //}
    i = 0;
    while (i<5){
        i = i+1;
        << i;
    }

    play($dealer);
    while (#doneBidding[0] && #doneBidding[1]) {
        play($player0);
        play($player1);
    }
    //evaluateHandWinner();
}

WinCondition
// Condition for the game to end - evaluated after each round.
// Must return a list of Card Entities (or null if no winner yet).
{
    <<"Checking Win Condition";
    if (#chips[0] <= 0 && #chips[1] <= 0) {
        return [];
    } else {
        if (#chips[0] <= 0) {
            return [$player1];
        } else {
            return [$player0];
        }
    }

    return null;
}

play(var e)
// Play function
{
    var i;
    var bid;

    bid =5;
    <<"Player going: ";
    <<e;
    //i = indexOf(#players, e);
    #doneBidding[0] = false;
    #doneBidding[1] = false;
    //<< " " ^ e; // print the cards of this card entity
    i =0;
    while (i<5)
    {
        i=i+1;
        <<e[i];
    }
    if ( bid == null) {
        <<"bid ok";
    }
    <<"Bid: " ^ bid;
    <<returnFive();
    bid < #lastBid;
    <<"bid 2 ok";
    >>i;
    <<"You gave input " ^ i;

    if (bid == null || bid < #lastBid) // haven't bet or overbet
    {
        >> bid; // input a bet (auto convert input to type of 'bid')
        #chips[0] -= bid; // subtract bet from your current chips
        #currentPot += bid; // add bet to the current pot
        #lastBid = bid; // record the last bid to check overbets
    }
}

```

```
        #doneBidding[0] = true;
    return;
}

returnFive ()
{
    return 5;
}
evaluateHandWinner(var lc, var round)
{
    //The best poker hand wins

    return @1;
}

testfunc()
{
    return 0;
}
```

```

Include
{
    "stdlib/stdlib.cgl";
}

CardEntities
{
    deck;
    player1;
    player2;
}

Globals
{
}

Start
{
    var i;
    var r;

    // test out variables holding CardEntity reference
    var p2;
    p2 = $player2;

    i = 0;

    while (i < 26) {
        $player1 <- $deck[0];
        p2 <- $deck[0];
        i++;
    }

    // print out each player's cards

    i = 0;
    << "player 1:";
    while (i < |$player1|) {
        << $player1[i];

        i++;
    }

    i = 0;
    << "player 2:";
    while (i < |p2|) {
        << p2[i];

        i++;
    }
}

Play
{
}

WinCondition
{
    if (|$player1| > |$player2|) {
        return [$player1];
    } else {
        if (|$player2| > |$player1|) {
            return [$player2];
        } else {
            return [];
        }
    }
}

```

```
Include
{
    "stdlib/stdlib.cgl";
}

CardEntities
{
    deck;
    player1;
    player2;
    player3;
    player4;
}

Globals
{
}

Start
{
    var i;
    var r;

    // use a list for convenience
    var playerlist;
    playerlist = [$player1, $player2, $player3, $player4];

    i = 0;

    while (i < 52) {
        r = ~4;
        << r;
        playerlist[r] <- $deck[0];
        i++;
    }

    // print out each player's cards

    i = 0;
    << "player 1:";
    while (i < |$player1|) {
        << $player1[i];

        i++;
    }

    i = 0;
    << "player 2:";
    while (i < |$player2|) {
        << $player2[i];

        i++;
    }

    i = 0;
    << "player 3:";
    while (i < |$player3|) {
        << $player3[i];

        i++;
    }

    i = 0;
    << "player 4:";
    while (i < |$player4|) {
        << $player4[i];

        i++;
    }
}
```

```

        i = 0;
        << "What's left in the deck?";
        if (|$deck| == 0) {
            << "Nothing! Cool!";
        } else {
            while (i < |$deck|) {
                << $deck[i];

                i++;
            }
        }
    }

Play
{
}

WinCondition
{
    if (|$player1| > |$player2| && |$player1| > |$player3| && |$player1| > |$player4|) {
        return [$player1];
    } else {
        if (|$player2| > |$player1| && |$player2| > |$player3| && |$player2| > |$player4|) {
            return [$player2];
        } else {
            if (|$player3| > |$player1| && |$player3| > |$player2| && |$player3| > |$player4|) {
                return [$player3];
            } else {
                if (|$player4| > |$player1| && |$player4| > |$player2| && |$player4| > |$player3|) {
                    return [$player4];
                } else {
                    return [];
                }
            }
        }
    }
}

```

```
Include
{
}

CardEntities
{
    deck;
}

Globals
{
}

Start
{
    var int;
    var string;
    var card;
    var cardEntity;
    var list;

    int = 1;
    string = "string";
    card = H2;
    cardEntity = $deck;
    list = ["a", "b"];

    <<int;
    <<string;
    <<card;
    <<cardEntity;
    <<list[0];

}

Play
{
}

WinCondition
{
    return ;
}
```