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

Chapter 1

Introduction

The Playing Card Game Simulation Language (PCGSL) is designed to be a simple programming language for programming 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 one would have to in a general-purpose language. This allows the programmer to focus on creating randomized simulations of popular games or hands, as well as quick mock-ups of new games based around standard 52-card decks.

1.1 Simple

PCGSL is simple to learn. Using well known C-style imperative syntax conventions, our language...

Chapter 2

Language Tutorial

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: Ocamlyacc, Ocamllex, 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

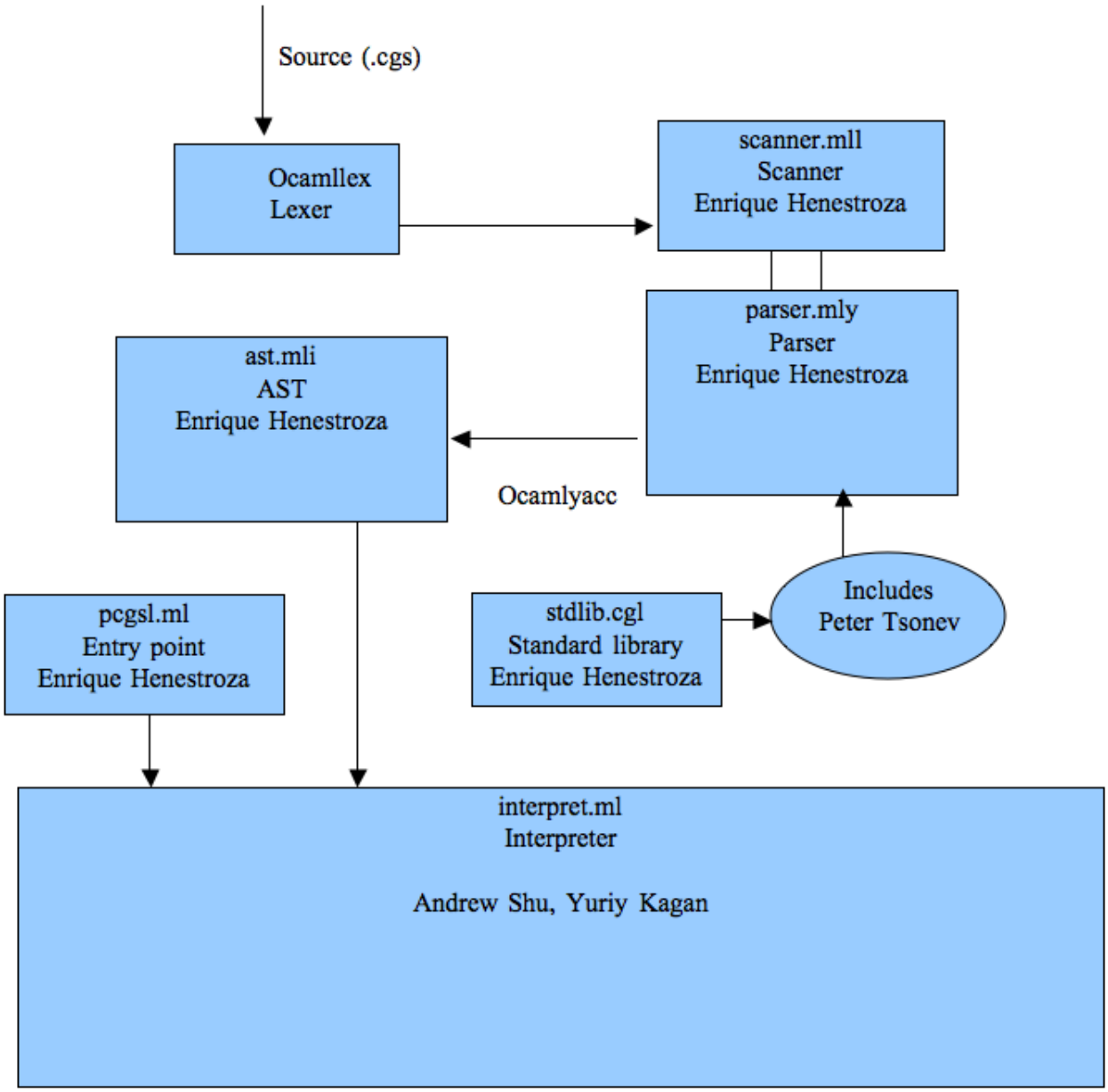
Chapter 5

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.


Source (.cgs)
Ocamllex
Lexer
scanner.mll
Scanner
Enrique Henestroza
parser.mly
Parser
Enrique Henestroza
ast.mli
AST
Enrique Henestroza
Ocamlyacc
Includes
Peter Tsonev
pcgsl.ml
Entry point
Enrique Henestroza
stdlib.cgl
Standard library
Enrique Henestroza
interpret.ml
Interpreter
Andrew Shu, Yuriy Kagan