Magic: The Gathering in Common Lisp

Jeff Linahan

Abstract

Magic: The Gathering is the world's most popular trading card game. So far, attempts to program its 210-page rulebook in order to create an AI for the game have resulted in systems that are very complex, and still not able to compete with humans. I believe one of the main causes of this is the choice of programming language. Most implementations are done in languages which emphasize execution speed or portability rather than development speed or flexibility. Common Lisp was classically the lingua franca for AI research, and its following features mesh well with the challenges of programming Magic: a read-eval-print loop, macros, dynamic typing, scripting, multiple inheritance, and symbolic computation. In this project I present a proof of concept implementation consisting of a command line interface for two humans playing Magic with two hardcoded decks. I will discuss what I have learned from tackling the challenges of the project and how I would proceed if I had years to complete it.

Acknowledgements

Dr. Paula Matuszek
Dr. Frank Klassner
The Villanova Trading Card Game Club

Introduction

Magic: The Gathering is the world's most popular trading card game. It was invented in 1993 by mathematician Richard Garfield and is distributed by Wizards of the Coast with about 20 million players. Over the years there have been several attempts to implement Magic's 210 page rulebook and over ten thousand unique cards in software. The most notable of these projects is Forge, started in 2007 and written in Java. Other notable implementations include Incantus (Python), XMage, Magarena, Multiverse (Java), BotArena, Wagic (C++), Manalink (C), and Magicgrove (C#). I call my implementation Maglisp (not to be confused with Maclisp). This paper is a summary of what I've learned while programming a small subset of Magic's rules, and the design choices I would make if I spent several years finishing it. I will be comparing my implementation to Forge, as it is currently the most complete Magic implementation available. To gain an appreciation for the complexity of programming Magic, consider that for the game rules alone (not including GUI or AI,) Forge consists of over 400 source files.

Overview of Magic: The Gathering

There are several game formats of Magic with slightly different rules depending on what types of decks are allowed, how many players there are, and how the teams are set up. The simplest and most common format is Constructed. In Constructed Magic, players construct a deck of 60 cards consisting of creatures, enchantments, sorceries, lands, and other card types. At the beginning of a game the decks are shuffled, becoming each player's "library" of spells and lands, and each player's health is set to 20. Turing a card sideways is called "tapping." Lands are tapped for mana, which can be used to cast spells. Some spells summon creatures to the battlefield, which are used to attack opponents. The game ends when all but one player reaches 0 life. A player can also lose the game by attempting to draw a card from an empty library.

Inspiration.

The Villanova Trading Card Game Club was founded in 2012. Our two most popular games are Magic and Yu-Gi-Oh. In 2013 I lead the senior projects team that wrote the *Yu-Gi-Oh! 3Digital Monsters* fan game for the Oculus Rift. After taking an AI class with Dr. Frank Klassner I had an idea for implementing the rules of Magic in Lisp.

Why Lisp?

Lisp was invented in 1958 by AI researcher John McCarthy, and Common Lisp was standardized in 1994. Many Lispers are known for their adoration of the language, and consider all other languages a step backwards [2]. This attitude is not entirely undeserved; it introduced a staggering number of features into high level languages. After reading the work of Paul Graham and talking to Dr. Frank Klassner about AI, I was interested in what Lisp could do. After taking his AI class, I found that areas where Lisp excels fit nicely with the challenges involved in programming Magic. These strengths include: Homoiconicity, Lamba Expressions, Multiple Inheritance, Macros and Domain Specific Languages, Dynamic Typing, Lazy Evaluation,

Artifical Intelligence, Read-Eval-Print Loop, and Symbolic Computation. Now I will consider each of these in detail.

Homoiconicity

In Lisp, all code and all data are stored as S-expressions. This makes it very easy to generate, load, and run code at runtime. Although uncommon, there are "text changing card effects" in Magic which treat card effects as data. This is trivial to implement in Lisp. Homoiconicity also means Lisp allows for scripting out of the box. Systems like Forge have fallen victim to Greenspun's Tenth Rule, "Any sufficiently complicated C or Fortran program contains an ad hoc, informally specified, bug-ridden, slow implementation of half of Common Lisp." and have to include their own scripting language. Python allows for scripting also, but it lacks the homoiconicity needed to take full advantage of S-expressions. Languages like Java or C++ on the other hand have no homoiconicity whatsoever.

Lambda Expressions

In Magic most cards have effects written on them with directions on what to do and when. The 'golden rule' of Magic is that directions written on the cards override the game rules. In this way, all spell effects are code snippets. Each bit of English text on a card can be represented as a lambda function, which can be loaded at runtime. This means the code for the engine can be very modular.

One natural question is whether the English instructions on Magic cards can be automatically translated into computer code. Until a few years ago, I thought this to be a prohibitively difficult task. However, with advances in natural language processing and Wizards standardizing the language for card effects, this is slowly becoming more possible. There is at least one attempt to automatically generate code from card text using a context free grammar, listed in the appendix. Ideally, I envision a system where players could create their own cards by typing in the effect in English, with a Perl program compiling it into Lisp code for use in game.

Multiple Inheritance

Many cards in Magic have multiple types. There are Creatures, Artifacts, Enchantments, Lands, and more. Many can have multiple types, such as Artifact Creatures, Enchantment Creatures, Artifact Lands, etc.. Unfortunately, in many mainstream languages it is either cumbersome to do multiple inheritance (C++) or impossible (Java.) The Common Lisp Object System (CLOS) on the other hand supports multiple inheritance out of the box, and its emphasis on the generic function instead of the class as the prime mover allows methods to be combined in creative ways. [7]

Macros and Domain Specific Languages

Forge is composed of several hundred source files. The biggest reason for this is the verbosity of Java. For example, each type of trigger has its own file, and they each differ little. Lisp macros allow the programmer to abstract away boilerplate patterns in the source code and generate it automatically. With this approach, things like triggers will only be a function long instead of an entire file. While lines of code is widely understood to be a poor metric of complexity, Lisp solutions are often an order of magnitude shorter than imperative languages. [3]

Macros are the killer feature of Lisp that other high level languages like Python lack. While it possible to view code as data in Python, such solutions are unbelievably ugly. [6]

Dynamic Typing

The top level class in Magic is called an "object," which can be an ability on the stack, a card, a copy of a card, a token, a spell, a permanent, or an emblem. Unfortunately, after this things get very confusing. Many objects can change their types during the game or have multiple types. Some are cards, some are not cards but are represented by cards, and some have no representation on the board at all. Cards become new objects when changing zones but are still represented by the same card. The ability to not have to define the data types in Lisp was critical when I was just starting out and hadn't figured out the best class hierarchy. One concern I had was that type errors in Lisp would not be caught until runtime. However, this was not a problem because modern Lisp compiles very fast.

Lazy Evaluation

Lisp dialects like Clojure use lazy evaluation to make generating game trees easy. Game trees are often astronomically large or even infinite, so this is important. In [1] Barski explains how lazy evaluation can be used to generate a game tree from a "move" function, which takes the current game state as input and returns the set of possible successor game states as output. This arbitrarily large game tree can be passed through filters which do alpha beta pruning, limit the ply, and apply a fitness function.

Artificial Intelligence

Like Prolog, Lisp is a popular language for artificial intelligence development. However, Common Lisp is much better suited for AI application development because unlike languages like Prolog or Haskell it does not force a particular style of programming. Much of the research on classic AI techniques is in Lisp, so it is quite easy to adapt known solutions. [1]

Read-Eval-Print Loop

Many existing Magic implementations have been done in mainstream languages like Java or C++. These languages do not possess a Read-Eval-Print Loop, and often have a lengthy compilation step. Lisp, on the other hand, is usually interpreted. When working in environments like SLIME, there is less distinction between edit time, compile time, and run time like other languages. Using the debugger, any values or code can be changed without restarting the program, meaning even game rules can be changed. [4]

Symbolic Computation

John McCarthy invented Lisp to do symbolic differentiation. Due to the symbol data type and the quotation shorthand, Lisp is a great language for doing processing symbols, much like Perl excels at processing strings. While C and is designed for fast numeric calculations, the game logic of Magic operates mostly on keywords and whole numbers. There is no complex math or data structures involved, only loops and branches based on whether or not cards hold certain properties.

Why Not Lisp?

By far the biggest disadvantage of writing in Lisp is how few programmers know it. Most people are put off by prefix notation and parenthesis. [5]

Development

Setup

The first two weeks consisted of configuring Emacs and SLIME. As a veteran Vim user, this had a significant learning curve. The interpreters I used were Steel Bank Common Lisp and Clisp. After everything was set up that, I began development on a command line interface for playing Magic. I hardcoded two premade decks to use for testing: Price of Glory and Infernal Intervention. As I am still new to Lisp, I didn't use a build system like ASDF. I set up Emacs to load a file called l.lisp, which is simply a set of commands which load all the other files.

Command Line

The file driver.lisp sets up the two players named Bob and Alice as well as the two decks, then launches the main game loop. The game loop simply gives a turn to each player in order. The heart of the turn structure is a hierarchy of functions in a file called turn-structure.lisp. The command input and output functions are defined in interface.lisp. Color is used liberally to easily show when a new turn or phase starts. All actions that can be done during a turn are invoked with a short name on the command line.

Coding Convention

I used the Scheme convention of using an exclamation point for mutating methods and question marks for predicates. While this makes it easy to quickly see if code is written in a functional style or not, the built in Common Lisp functions don't use it.

Class Hierarchy

My first attempt to correctly program card types such as Artifact Creature, Enchantment Creature, and Artifact Land used multiple inheritance. Unfortunately, I soon learned that Magic cards can change their types during the game. This would have required quite tricky use of CLOS and Metaobject Protocol (MOP), so I abandoned it in favor of a generic Permanent type, which has slots called creature-data, artifact-data, etc. that are simply left unbound if a permanent is not that particular type.

State Based Actions

In Magic there are a set of state based actions which serve to clean up the game state. This includes sending cards to the graveyard, assigning damage, healing damage, and so on. They are checked whenever a player would gain priority. All "standard" state based actions are kept in a single file as a set of defun definitions. Pointers to these functions are kept in a global variable called *state-based-actions*. All state based actions that only apply in certain variants are defined in those variant's files. When the variant files are loaded they are added to *state-based-actions*.

Priority

The concept of priority in Magic is used to determine who may play spells and when. Usually, the active player gets priority several times during their turn. Whenever they pass (decline to play a card) all other players receive priority also in active player non-active player (APNAP) order. In casual play, players usually do not explicitly say when they are passing priority; it simply happens too often. Instead the game is played in real time, and players only talk about priority if there is a dispute. However, on a command line it is necessary to ask every time. The official Magic computer game *Duels of the Planeswalkers* instead has a timer in which the player can respond by default, after which is assumes they are passing. Of course, they can pass explicitly immediately or ask for more time to think if necessary. The command line can be made slightly less painful to use by automatically passing if the game detects a player logically cannot do anything. A second idea is to by default disable checking for priority during rarely used phases, such as the Draw and End step. If a player has an instant they are thinking of playing during this time they are expected to turn on priority checking for that phase. These two solutions are employed by Forge.

Future Ideas

I was able to convert a JSON database of all Magic cards into S-expressions using a library. If a card effect parser was written, this would allow new cards to automatically be programmed into the engine. After a month away from the project, I began forgetting the names of the commands I was using. A long term solution would be to create a GUI with a client server architecture, similar to how SLIME and Swank works. Online play would be very easy to implement with this approach.

Conclusion

I had a little Lisp experience before starting this project but had never developed a big application with it. However, once I had Emacs and SLIME set up I was surprised by how fast I was making progress. After just a couple months I was convinced; a Lisp implementation would take a fraction of the time to write make than a Java one. With no compilation step and the ability to debug anything, anywhere, there are simply very few barriers to making the machine do what you want. It would be much smaller codebase as well. Unfortunately, working alone I realized it would still take years to get a quality implementation. The most important thing I've learned from this project is the importance of using the right tool for the job. While Lisp may be a strange language, there are still areas where it excels.

References

[1] Barski, Conrad. *The Land of Lisp. 2010.*

[2] Ecklar, Julia. *Eternal flame.* http://www.gnu.org/fun/jokes/eternal-flame.en.html.

[3] Graham, Paul. *Beating the Averages*. 2001. www.paulgraham.com/avg.html.

[4] Graham, Paul. *What Made Lisp Different.* 2001. http://www.paulgraham.com/diff.html.

[5] Graham, Paul. *Revenge of the Nerds*. 2002. http://www.paulgraham.com/icad.html.

[6] Norvig, Peter. *Python for Lisp Programmers.* http://norvig.com/python-lisp.html.

[7] Seibel, Peter. *Practical Common Lisp. 2005.*

Appendix

Magic: The Gathering Comprehensive Rules
http://magic.wizards.com/en/gameinfo/gameplay/formats/comprehensiverules

Maglisp source code
https://github.com/jeffythedragonslayer/maglisp

List of MTG programming projects
http://www.slightlymagic.net/wiki/List_of_MTG_Engines

Forge Wiki
www.slightlymagic.net/wiki/Forge

Forge Blog
http://mtgrares.blogspot.com/

A Magic: The Gathering Parser
https://github.com/Zannick/demystify