Proposal
MoM Adventure Language

Prepared by: Marco A. Peyrot (A00815262)
Elías A. Mera (A01280762)

Prepared for: Compiler Design

Submission Date: Thursday, September 7th 2017

Marco A. Peyrot _____ Date _____

Elías A. Mera _____ Date _____

1 Purpose

The purpose of this project is to develop an Object-Oriented language to facilitate the creation of scenarios and components for the board game Mansions of Madness Second Edition [1]. Through the use of this language, owners will have the opportunity of creating additional content for the game and getting a bit into the field of computer programming. Therefore, a key emphasis of the language is to make it simple and comprehensible to promote its use and to give it more utility.

To keep the project within scope, the execution of a program made with this language will be similar to that of playing an old text-based game from the 1980's. However, it is hoped that in the future, this project will have both graphical input and output to facilitate game creation and language use.

As a side note, the language will come with the basic types to program the default game, but will let the programmer develop new ones for new game mechanics, rules, or physical components. As a result, the language should be able to support the creation of any type of text-based game.

2 Language main objective

The language's main area is education since it aims to promote learning general programming through an easy to use object-oriented language. The language accomplishes this by letting users develop and play scenarios and stories for the board game Mansions of Madness Second Edition.

3 Language Requirements

3.1 Basic elements

The language keywords are described below. However, since the base language is `Pyhton 3`, `Python 3's` reserved tokens [5] are also reserved for the MoM Adventure Language. Antlr also has its own set of reserved words, which must be accounted for.

These are shown in tables 1, 2 and 3:

Reserved Words				
False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Table 1: Python 3 Reserved Words.

Reserved Words				
import	fragment	lexer	parser	grammar
returns	locals	throws	catch	finally
mode	options	tokens	rule	

Table 2: Antlr Reserved Words.

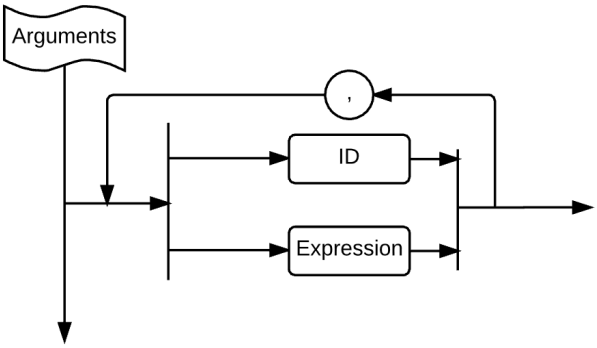
Reserved Words				
is_a	of_type	property	while	main
new	enumerate	parent	this	true
false	show	show_line	read_text	read_int
read_real	cast	provided_that	if_not	class
returns				

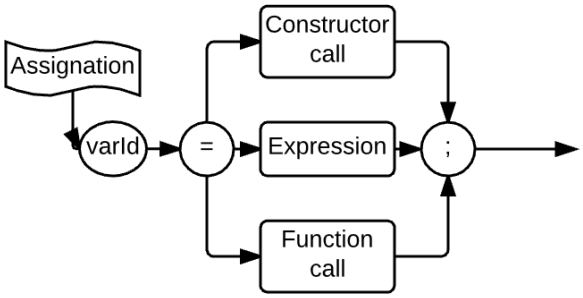
Table 3: MoM Adventure Language Reserved Words.

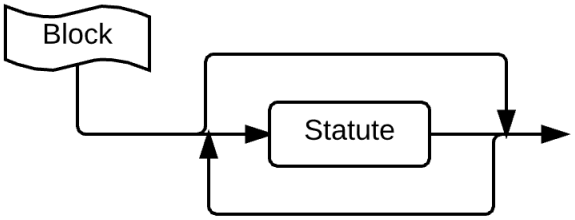
3.2 Language Structures

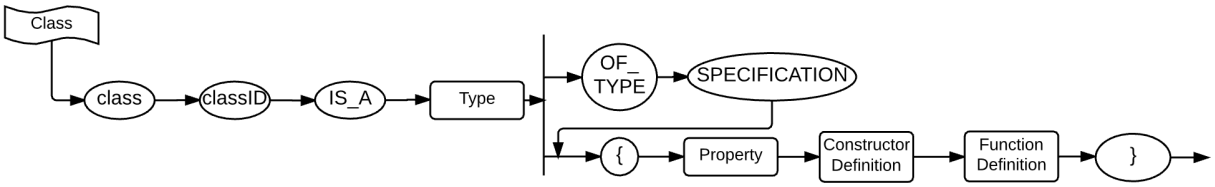
3.2.1 language Lexemes

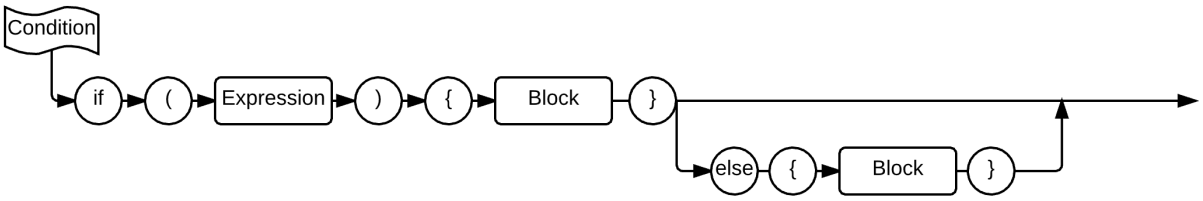
- **Variable Names**: any character sequence from 'a' to 'z' and from 'A' to 'Z' that begins with a lowercase letter. It can have the symbol '_', but no numeric characters. A special case is for the array, in which the name is followed a number (the index) surrounded by the '[' and ']' symbols.
- **Class and Method Names**: any character sequence from 'a' to 'z' and from 'A' to 'Z' that begins with a lowercase letter. It can have the symbol '_', but no numeric characters.
- **Class Identifiers**: any character sequence from 'a' to 'z' and from 'A' to 'Z' that begins with an uppercase letter. It can have the symbol '_', but no numeric characters.
- **Text constant**: any character sequence that does not contain the '"' (double quote) symbols surrounded by two '"' (double quote symbols).
- **Int constant**: any numeric character sequence, it can be preceded by a '+' or a '-' symbol.
- **Real constant**: any numeric character sequence, that can have one '.' (dot) symbol to separate the *mantissa* and the *exponent*. Both parts can be preceded by a '+' or a '-' symbol.

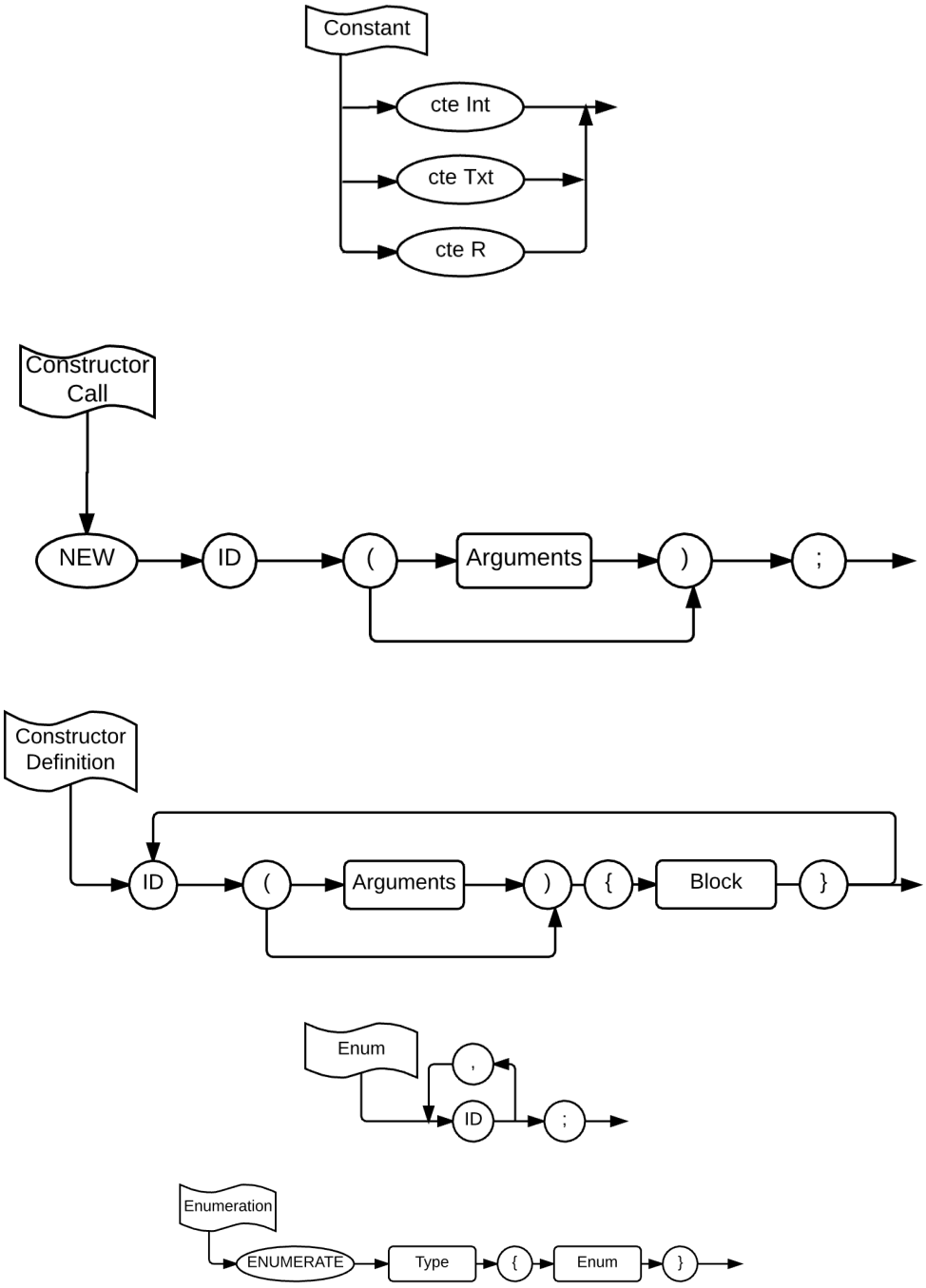

Arguments
,
ID
Expression


Assignation
varId
=
Constructor call
Expression
Function call
;


Block
Statute


Class
class
classID
IS_A
Type
OF_ TYPE
SPECIFICATION
{
Property
Constructor Definition
Function Definition
}

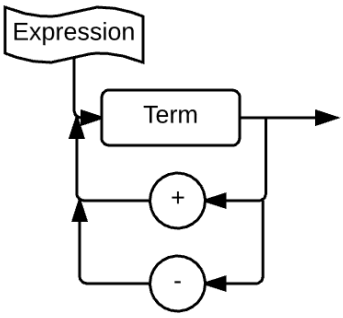

Condition
if
(
Expression
)
{
Block
}
else
{
Block
}

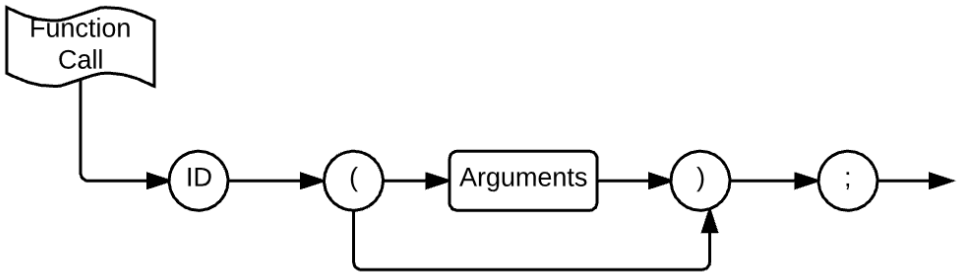


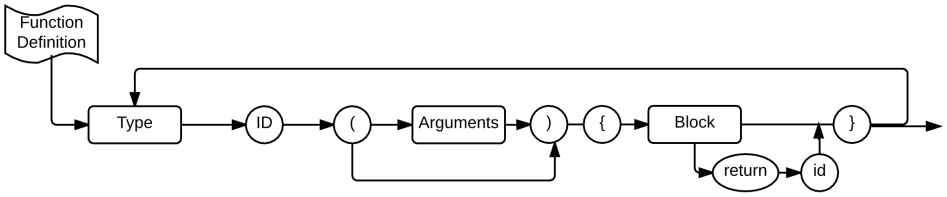
3.2.2 Syntax Structures

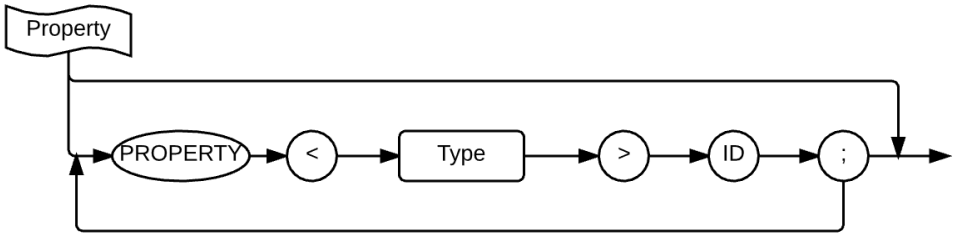
3.2.3 Code Examples

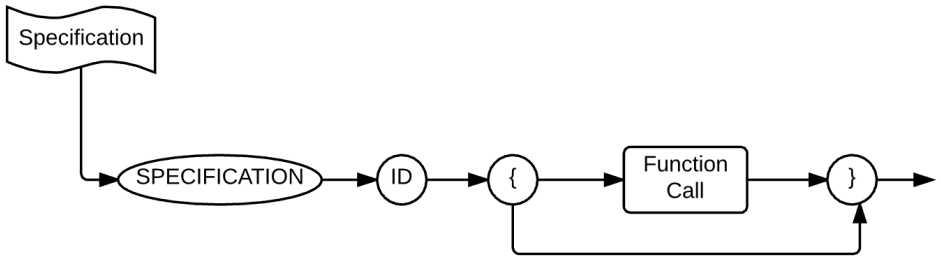
This section provides the reader with two short examples to understand the feel and like of the code, as well as its basic lexical and syntax structure. The examples are located in figures 1 and 2.

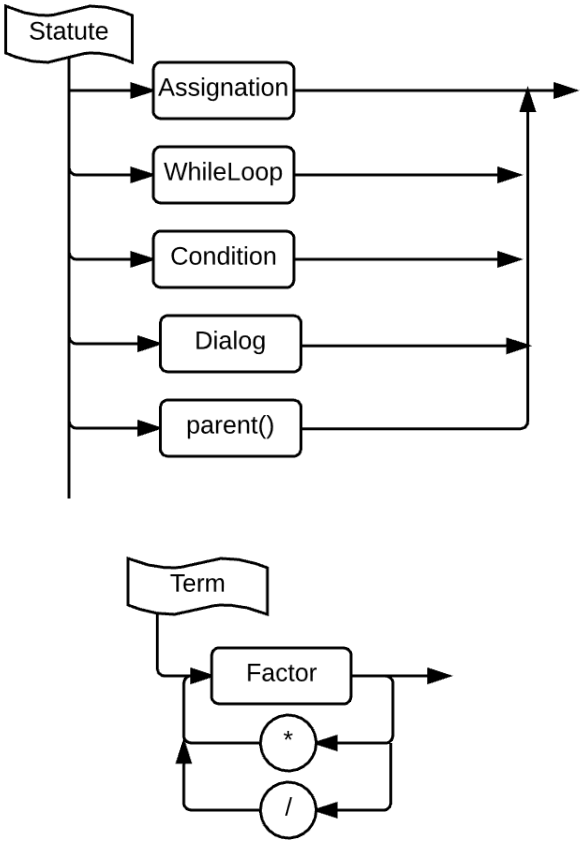

Expression
Term
+
-


Function Call
ID
(
Arguments
)
;


Function Definition
Type
ID
(
Arguments
)
{
Block
}
return
id


Property
PROPERTY
<
Type
>
ID
;


Specification
SPECIFICATION
ID
{
Function Call
}



3.3 Language Semantics

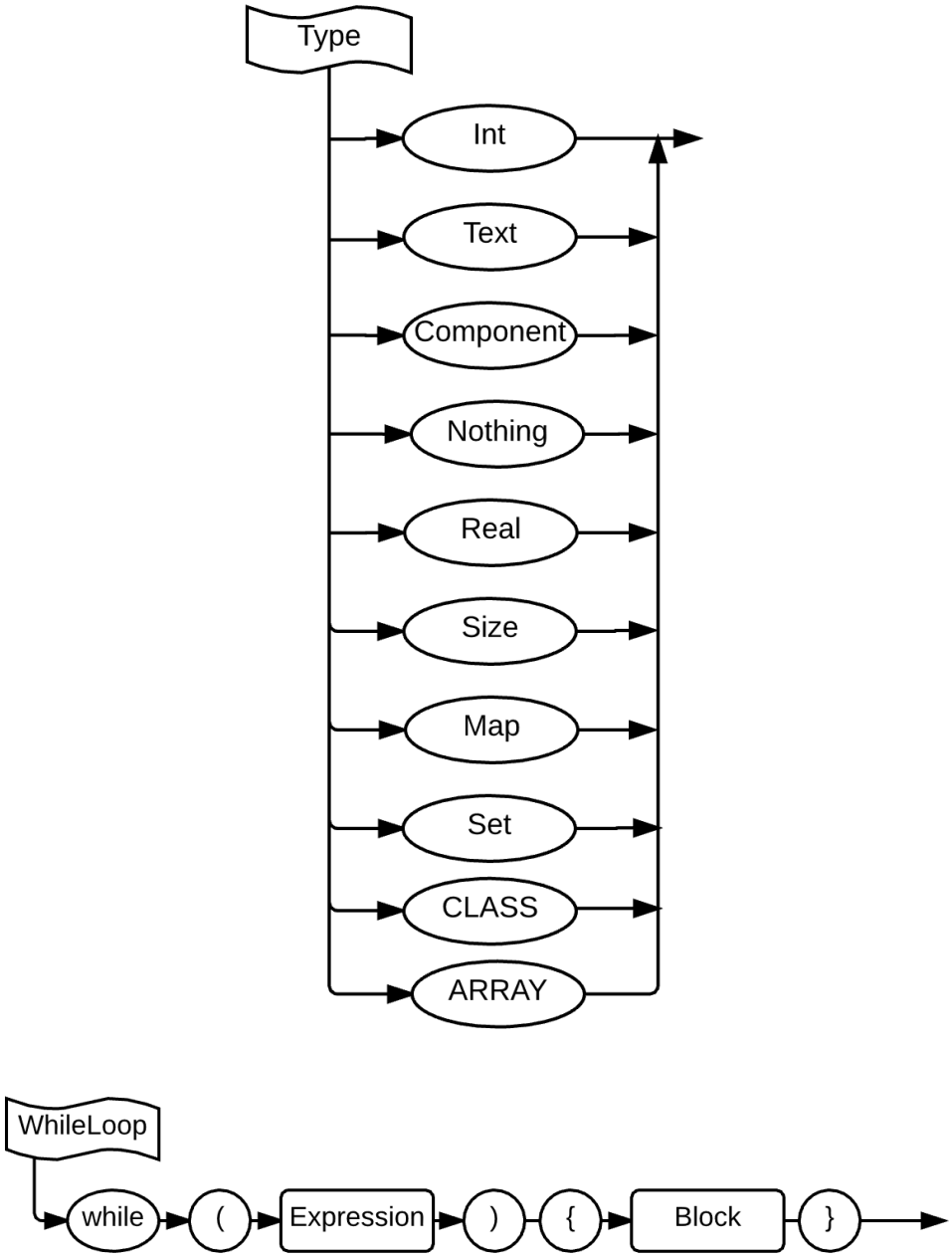
For the first proposal, a few aspects about the language must be considered:

First, the only types that can be used in arithmetic expressions are `Int` and `Real`. Their evaluation yields the target types according to table 4:

Type 1	Type 2	Target Type	Validity
Int	Int	Int	Success
Int	Real	real	Success
Real	Int	Real	Success
Real	Real	Real	Success
Int	*other*	*undefined*	Error
Real	*other*	*undefined*	Error
other	Int	*undefined*	Error
other	Real	*undefined*	Error

Table 4: Semantic rules for expression types.

Second, the `specification` type should not be considered as a class, but as a `Java` interface to define certain behavior that a class can have. Third, the `property` token is used only to identify internal fields of a class. Finally, all code, including the one that



contains the `main` method must be contained inside a class.

3.4 Special functions

- show: this special function is used to print into the standard output a stream of characters which are not followed by a default newline character.

```
1  nothing show(Text text) { ... }
```

```
Character is_a Card of_type Player {
    property<Card[10]> items;
    property<Int> items;

    Character(Int physicalLife, Int mentalLife) {
        parent();
        this.physicalLife = physicalLife;
    }

    Nothing addCommonCard(Card card) {
        if(this.numberOfItems < items.size() - 1) {
            items[items.elementCount() + 1] = card;
        }
    }

    Text name() {
        return "Father Mateo";
    }
}

specification Player {
    Int physicalLife();
    Int mentalLife();
}

Card of type Component {
    property<Type> cardType;
    property<Face> front;
    property<Face> back;
}

enumerate Type {
    SPELL, COMMON, UNIQUE;
}
```

Figure 1: Code example for MoM Adventure language (Classes).

```
Game {
    property<MapTile[10]> tiles;
    property<Clue> aClue = new Clue();

    nothing main(Text[0] arguments) {
       tiles[0] = new MapTile("LivingRoom", "living_room.jpg", 1, 2);
       aClue.setTitle("Blood on the floor");
       aClue.setMessage("You spot something on the floor.");
       aClue.readDiceRoll();

       tiles[0].addClue(aClue);

       if(aClue.isPressed()) {
           showDialog("bla blah blah", Option.CLOSE);
       }

       Int i = 0;
       while(i < cards.size()) {
           Card card = cards[i];
           i = i + 1;
           Text name = card.name();
           show_line(name);
       }
    }
}
```

Figure 2: Code example for MoM Adventure language (Entry point).

- show_line: this special function does the same as the show function, but it adds the newline character at the end of the stream. It has two versions, one with arguments and one without.

```
nothing show_line(Text text) { ... }
nothing show_line() { ... }
```

- read_text: this function reads from the standard input a stream of characters and returns them as a `Text` instance.

```
Text read_text() { ... }
```

- read_int: this function reads from the standard input a stream of numeric characters and returns them as an `Int` instance.

```
Int read_int() { ... }
```

- read_real: this function reads from the standard input a stream of numeric characters (and the '.' character) and returns them as a `Real` instance.

```
Real read_real() { ... }
```

- cast: this function receives two arguments, the type of the target variable and the variable or value to be cast. The permissible types are `Text`, `Int`, and `Real`.

```
Text cast(Text, Int value) { ... }
Text cast(Text, Real value) { ... }
Int cast(Int, Text value) { ... }
Real cast(Real, Text value) { ... }
```

3.5 Data types

The language comes with a series of default data types which can be divided into two categories:

3.5.1 Base Types

These types are essential to program an application in the MoM Adventure language.

- Text: represents a sequence of characters. It is the name for the `string` type in languages like `Java` and `C`.
- Int: represents an integer number.
- Nothing: represents no return value, equivalent to the `void` type in `C`-like languages.
- Real: represents a real number. More specifically, the *double* or *float* types common to `C`-like languages.
- Component: this is the base class for the MoM Adventure Language. It represents a game component (like a card, mat, or token). All other objects in the language inherit from this class. Its base fields are `name`, `id`, and the `size`.
- Size: represents the size of an in-game component: its height and width, both of which are `Real` numbers.
- Map: one of the three default data structures, it can also be referenced as a *dictionary*. It maps a *key* to a *value* for easy access.
- Set: another default data structure, this one acts as a container for unique values, allowing for easy access and search.

3.5.2 Convenience Types

These types are provided specifically to program an application for the Mansions of Madness game. They represent the various components that make up the game, that is: cards, tokens and map tiles, to name a few. Figure 3 shows several examples of them.

- Card: represents a physical game card, it has two *faces* and each one is divided into *sections*.
- Token: represents a physical game token or carton piece, it also has two *faces*, but with just one *section*.
- Piece: represents a bigger item of the game, like a player mat or a map tile. It is like a *Card* type, but it can have more features and it can aggregate other *components*.
- Face: represents the side of a *component*. It is a container for several *sections*.
- Message: it is a simple container that comes with a title, message, and two types of input: via a numeric or text entry, or a series of options which the player can select.
- Player: it is a special instance of component since it can have several other components attached. Its base members and fields are those found on the player sheet in the real board game.
- Section: It is a orderly form of managing the different content that a component has. Examples of this could be the information shown in a card, like a title, image, description, cost, etc.

- Creature: it is a kind of special component since it has several additional actions related to it. These actions are *move*, *attack*, and *evade*.

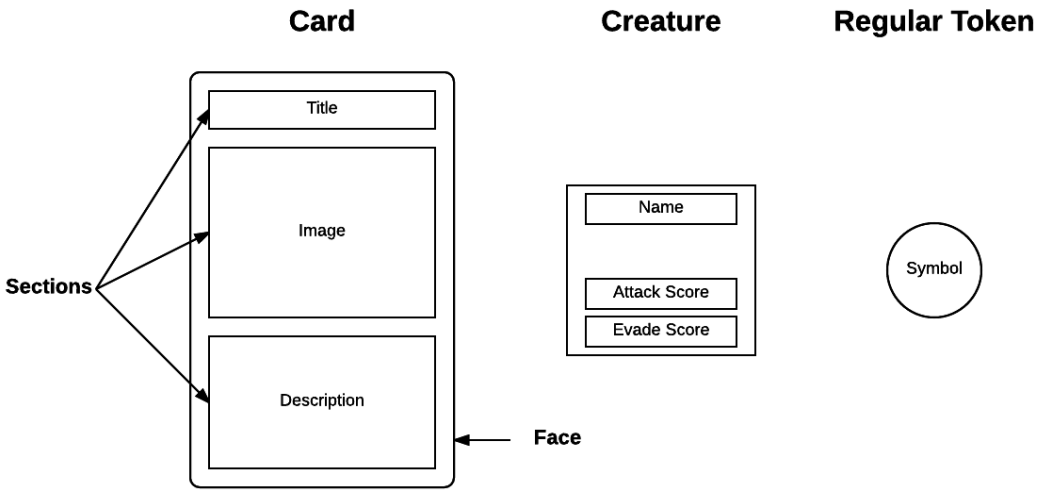


Figure 3: Example of game components.

4 Language and Computers used for Development

Regarding the technicalities of the project, three things are worth mentioning: first, the programming language that will be used, as the foundation for this new language and the virtual machine where it runs, is Python 3.6.x. Second, the project will be developed in the devices of both team members, one running the `MacOS X` operating system while the other runs `Windows 10`. Finally, the lexical and semantic analyzer that will be used to generate the scanner and parser is Antlr for Python [2].

References

[1] F. Flight. Mansions of madness second edition. https://www.fantasyflightgames.com/en/products/mansions-of-madness-second-edition/. [Online; accessed September 5th, 2017].

[2] T. Parr. About the antlr parser generator. http://www.antlr.org/about.html. [Online; accessed September 5th, 2017].

[3] T. Parr. Grammar lexicon. https://github.com/antlr/antlr4/blob/master/doc/lexicon.md. [Online; accessed September 5th, 2017].

[4] T. Parr. Left-recursive rules. https://github.com/antlr/antlr4/blob/master/doc/left-recursion.md. [Online; accessed September 5th, 2017].

[5] T. P. L. Reference. Lexical analysis. https://docs.python.org/3/reference/lexical_analysis.html#identifiers, 2017. [Online; accessed September 5th, 2017].