BOGUS - BOard Game Specification Language

Cary Maister

COMS 4115 - Spring 2009

1 Introduction

Purpose
Bogus is intended as a relatively easy way to specify a board game, to be played interactively by a user at a computer terminal. Bogus will provide a framework for the programmer to specify a few elements that are common to many board games along with the rules that define how the gameplay proceeds.

Anatomy of a Bogus program
Any Bogus game will include the following elements:

- a board - consisting of a sequence of spaces
- a set of players
- a deck of cards from which plays draw to move around the board

Each space on the board and card in the deck will be associated with a rule. A rule is a procedure that has access to the current state of the game and can apply arbitrary logic to manipulate that state. When a card is drawn, that card's rule is invoked, which may move one or more players around the board, or modify players' attributes. When a player is moved to a space on the board, the space's rule is invoked.

A Bogus program is a single text file that initializes the board, the players and the deck, possibly using inputs from the players, then invokes a game engine which sequentially gives each player a turn drawing from the deck and using the rules to move the players around the board until the game terminates.

Flexibility
The primary goal of Bogus is to provide a simple language that can be used to specify a wide variety of games with different board layouts and different principles of how players move through the game and how the game ends. A basic example is a game where players draw cards from a deck, and each card tells the player to advance a certain number of spaces, and the first player to reach the final space wins. More complicated rules could be introduced to provide a game like snakes and ladders where players may jump around the board in a non-linear fashion, or a game like Sorry where players can land on each other's piece to force them back to the beginning of the game.

With some creative interpretation of the "board" metaphor, one could create a Bogus choose your own adventure story, where each space on the board represents a place in the story, and the player inputs their choices and move to different events and there may be more than one winning outcome. For truly enterprising, stay-in-your parents' basement eating cheetos types, there is the possibility of a Bogus text based role playing game, where players move around the game board collecting objects, battling dragons, trying to build experience and so forth.

The primary goal of Bogus is a to allow a straightforward translation from an actual board game to a Bogus program. Just like a real board game, a Bogus program defines the group of players, the game board , the dice, and an optional deck of cards. Then each game play rule is translated into a bogus rule and each space on the board has an associated rule to specify what happens when a player lands on that space.

2 Bogus Tutorial

A Bogus program consists of three sections:

1. Define structure of your Players, Spaces and Cards
2. Global declarations
3. Game rules

Some language basics are:

- Comments begin with a semi colon and extend to the end of the line
- expression syntax is very similar to C or Java
- Variables aren't explicitly defined - a variable is automatically declared the first time it is assigned to. The variable's type is determined at that first assignment and cannot be changed.
- There are two special rules: start_game, which is called one before the game begins, and start_turn, which is called once at the beginning of every turn

2.1 Defining Types

```
Player has current(string), colour(string);
Space has next(string);
Card has val1(int);
```

Once these types have been specified, the rules can acccess value of these types using statements such as:

Players.("Trillian"); # creates a new player named Trillian

Players.("Trillian").current = "start"; # Sets value of current for player Trillian

Space("start")._rule = "start_rule";

2.2 Defining globals

my_glob1 = "2"; # creates a new global variable with an int value

my_globstr = "Krikkit" # new global variable with a string value

2.3 Defining rules

rule start_game

{

    print "Enter number of players: ";

    n = to_int(read); # reads a number from stdin, converts to an interactions

    i = 0;

```
        while(i < n)
        {
                print "Input name for player + to_string(i) + ": ";
                name = read;
                Players.(name);
                i = i + 1;
        }
}
rule start_turn
{
        i = roll(6) + roll(6);    # Roll two dice to determine how many spaces to move
        cur = _player.current;
        while(i > 0)
        {
                cur = Spaces.(cur).next
        }
        _player.current = cur;
}
```

3 Bogus Language Reference Manual

3.1 Introduction

Bogus (BOard Game Specification) is a specialized language for defining "board games" that can be played by one or more users at a terminal. The language includes constructs that are particularly suited to the implementation of a board game, as well as control flow, input/output, comparison and other operations common to most high-level languages. The overall goal of Bogus is to create a language that makes it straightforward to implement the basic components of many type of board games, while also allowing the programmer to include more complex game rules and interactions between players.

3.2 Lexical Conventions

A Bogus program consists of a single text file containing characters in the ASCII character set. This file is compiled into an intermediate format which is then excuted by the Bogus interpreter to actually play the game. This section describes the various token types that make up a bogus program.

3.2.1 Conventions

All ASCII values in this manual are given in decimal.

The terms "letter" and "digit" in this manual follow the common meaning of the terms in most common programming languages that use ASCII:

A *letter* is an uppercase or lower case character between A and Z (ASCII values 65 - 122)

A *digit* is a character between '0' and '9' (ASCII values 48 - 57)

All keywords in this manual are printed in `fixed-width` type.

Bogus is case-sensitive, so for example:

`if` is a keyword, but If is an identifier (though one that should probably be avoided).

thing1, Thing1 and THING1 are three separate identifiers.

3.2.2 Whitespace

The following ASCII characters are all considered whitespace. Since Bogus is a freeform language, whitespace (outside of string literals) is generally ignored. The exception is identifier separation: a sequence of one or more whitespace characters serve to separate adjacent tokens.

Whitespace characters:

Character	**ASCII Value (decimal)**

Horizontal tab	9
Newline/line feed	10
Carriage return	13
Space	32

3.2.3 Comments

A comment begins with a hash - '#' - character, and extends until a newline (ASCII 10) character. All contents of a comment are ignored by the compiler, and the comment is semantically equivalent to a whitespace.

3.2.4 Identifiers

An identifier is a sequence of one or more letters and digits and underscores (ASCII 95), where the first character is a letter. There is no predefined limit on identifier length, though the programmer is reminded that he is the one who has to decipher short, cryptic identifiers and retype long, verbose ones.

3.2.5 Keywords

All words listed below are reserved and may not be used as identifiers:

bool

draw

else

end

exit

false

float

has

if

int

local

print

read

roll

string

true

while

Card

Cards

NoCard

NoPlayer

NoRule

Player

Players

rule

Space

Spaces

3.2.6 Constants

There are four types of constants in Bogus: integer, floating point and string.

3.2.6.1 Integer Constant

An integer constant is a sequence of 1 or more digits, which is taken to be a decimal value. The sequence may be preceded by a minus sign ('-', ASCII 45) to indicate that the integer is negative.

3.2.6.2 Floating Point Constant

A floating point constant consists of a sequence of 1 or more digits, followed by a decimal point ('.', ASCII 46) follow by a sequence of 1 or more digits. The decimal point is required, and one may omit the sequence of digits before or after the point, but not both. The floating point may be preceded by a minus sign ('-') to indicate that the value is negative

Some examples of valid floating point constants are:

1.0, .1, -2.99999, -.4

3.2.6.3 Boolean Constant

As you might expect, there are two boolean constants - `true` and `false` - which correspond to the boolean values you would expect.

3.2.6.4 String constant

A string constant is enclosed by double quotes ("", ASCII 34) and consists of characters and valid escape sequences. Valid characters are ASCII 32 through 254, excluding the double-quote character (ASCII 34), the backslash (ASCII 92) and the DEL character (ASCII 127).

Escape sequences are two character combinations that can be used to represent some special characters, as defined below:

Escape sequence	Represents
\\	\ (backslash)
\"	" (double quote)
\n	(line feed)
\r	(carriage return)
\t	(horizontal tab)

Any two character sequence begining with backslash and not listed in the table above is invalid in a string constant.

3.3 Types

3.3.1 Primitives

Bogus contains four primitive types: integers, floating point numbers (precision TBD), booleans and character strings.

Variables containing primitive types are not explicitly declared. Instead, the variable springs into existance when it is first assigned to, and its type is inferred from the assignment value. As long as a variable is in scope, it will always have the same type as when it was first created (see the scoping rules for more detail).

3.3.1.1 Integers

Integers are integral values in the range -TBD to TBD

3.3.1.2 Floating point values

Floating point values can be positive or negative and have precision TBD.

3.3.2 Booleans

Booleans can store only the Boolean values true or false.

3.3.3 Character strings

Character strings have a size limited only by memory constraints on the system where the compiler and interpreter are running. They may contain all ASCII characters from 32 through 254 (excluding ASCII 127, the DEL character) as well as the following white space characters:

Character	**ASCII Value**
Horizontal tab	9

Newline/line feed	10
Carriage return	13

3.3.4 Composite Types

3.3.4.1 Defining

Three elements of a board game are represented as composite types:

1. Player
2. Space
3. Card

The structure of each type is declared at the beginning of the program (see "Program Structure" section below). Each composite type can contain any number of variables, each of which must be of a primitive type. The names and types of the member variables are declared at the beginning of the program.

A composite type declaration takes the form:

<composite-decl> -> <type> has <member-list> ; | <type> has nothing;

<member-list> -> <member> | <member> , <member-list>

<member> -> <member-id> (<member-type>)

<member-type> -> int | float | bool | string | Player | Space | Card

<member-id> -> any valid identifier

Each composite type also has a special member value, as described below:

Composite type	**Member**	**Type**	**Description**
Player	_name	string	Player's name.
Player	_space	Space	Player's current space.
Space	_rule	Rule	Rule that is invoked when player lands on the space
Space	_name	string	Space's name.
Card	_rule	Rule	Rule that is invoked when player draws the card
Card	_name	string	Card's name.

Note the that _rule members are an exception (to the rule): their type is Rule which is not a primitive data type. Conceptually, they store a reference to a Rule that will be invoked by the game interpreter at the appropriate time in game play.

Note also that unlike user-defined members, the _rule and _name members are set once when each Player, Space or Card is created, and can never be changed.

Each composite object has a _name, which must be unique among all objects of that type; that is, no two Player objects can have the same _name, but there can be a Player and a Card that each have the same _name.

3.3.4.2 Accessing Members Directly

Elements

All user-defined members can be read and written just like regular variables. Members can be accessed using the dot (".") operator.

Example (the syntax used to retrieve a player from the list of players is discussed in the next section).

```
Player has colour(string);
...
rule some_rule
{
        Players("Cary").colour = "red";
        print Players("Cary").colour;
}
```

This code snippet will set the colour member to "red" for the Player named Cary, then print Red

Note that the _name special variables are set once for each Player or Space added to the game and then remain read-only.

3.3.5 Magic Globals

There are certain magic globals that are available in any Rule within the program. They are initialized right before the start_game Rule is executed (see "Structure Of A Bogus Program" below).

3.3.5.1 _player

As mentioned above, the Players array contains all Players in the game, in the order in which there were added. The _player global is initialized to be NoPlayer, and can be manually changed by any Rule at any time. Any Player (except the NoPlayer) can be assigned to the _player global.

Changing the current player triggers the end of a turn -- see "Gameplay" section for more details on how the game progresses.

3.3.5.2 _next

The _next variable always contains the value of the Player in the Players array.

In other words, if num_players is the number of players in the array, and i is the index of the current player, the following is always true:

```
_next == Players.( (i+1) % num_players)
```

3.3.5.3 _space

The _space variable always contains the Space occupied by _player. Any rule can assign any Space (except the NoSpace) to _space.

Changing the current space for _player triggers the execution of that space's Rule -- see "Gameplay" section for more details on how the game progresses.

3.3.5.4 "Null" Values

Each composite type has a corresponding "null" value, which has the same type as the composite, but which is not equal to any object other than itself. The name of the special null value is the type name preceded by "No": NoCard, NoSpace, or NoPlayer. It is equivalent to an object whose _name value is guaranteed to be different from any other object.

For example, this statement will always be false:

```
Player.("Slarti") == NoPlayer;
```

Another example:

```
p1 = NoPlayer;
p1 == NoPlayer;        # evalutes to true
p1 == Player.("Zaphod")
p1 == NoPlayer         # evaluates to false
```

3.3.6 Arrays

Arrays are only available the three special composite types. For each of these types, there is a corresponding global array, identified by the type name followed by the letter s.

The arrays can be accessed in two ways:

- indexed - array elements are numbered starting at 0. Array elements are numbered in the order in which they were added to the array
- associative array - elements are accessed by the _name string specified when the element was first added to the array.

Array elements can be accessed using a subscript enclosed in parentheses. The subscript can be one of two types:

- int - the index of the desired element
- string - the _name of the desired element

Note that the order of the array elements cannot change once the game has been initialized. Similarly, _name values cannot be changed after the Player or Space is added to the array.

3.3.7 Dice

The Die type represents a fair die with an arbitrary number of sides n, numbered 1 through n. A Die is rolled simply by writing:

roll(expr)

Where expr is any expression with an integer value.

```
print to_string(die(10));          # Will print an integer between 1 and 10
print to_string(die(6) + 2);       # Will print an integer between 3 and 8
```

3.4 Expressions

All Bogus expressions are terminated with a semicolon, and each expression has a value in one of the primitive types. Expressions can serve as a operands for one of the operators listed below, and an operator may return a value different from the types of its operands.

Note that all expressions have a type, and no coersion or automatic type conversions are performed. Types can be explicitly converted using the conversion operators listed later in this section.

3.4.1 Assignment Operator

Assignments are performed using equals sign.

An assignment takes the form:

<assignment> -> <lvalue> = <expr>

Where <lvalue> is either:

a. the name of an existing variable

b. the name of the member of a composite type

c. the name of a new variable

If the lvalue is a. or b., then its type must match the type of the expression value on the right of the equals sign.

If the lvalue is c., then the assignment also serves as the declaration of the variable within the current scope (see scoping rules for more details). The type of the new variable is determined by the type of the expression on the left side.

3.4.2 Concatenation Operator

The concatenation operator is a binary operator taking two string expressions as operands.

<str1> + <str2>

The value of a concatenation expression is a string containing all of <str1> followed immediately by all of <str2>.

Since the concatentation operator takes only string as arguments, to concatenate any other types together (e.g. string and int), you must convert the non-string operand expression to a string using the appropriate conversion operator.

3.4.3 Arithmetic Operators

The binary arithmetic operators +, -, *, and / are used in their standard arithmetic sense.

The first four operators may be applied either to two integers or two floats. To operate on two operands of different types, one will have to be explictly converted to match the other.

3.4.4 Comparison Operators

<, >, <=, =>, ==, !=

== and != can also do string comparison (are strings' contents identical).

The comparison operators test some relation between the two operands and their value is a bool reflecting the outcome of the test.

All the comparison operators can be used on two numbers of the same type: either two integers or two floating point values.

The == and != operators can also be used to compare two bool values or two string values.

For two bools, == is true if and only if both operands have the same truth value. != is true if and only if both operands have different truth values.

For two strings, == is true if and only if the operands are identical; specifically, if both operands have the same length and the same character at very position. != is true if and only if the operands are not identical.

3.4.5 Logical Operators

&&, ||, unary !

Logical operators can take only bool values as arguments. The && operator has a true value if and only if both operands are true. The || operator has a true value if and only if at least one operand is true.

Both binary logical operators use lazy evaluation:

- if the expression on the left of && is false, the expression on the right will not be evaluated
- if the expression on the left of || is true, the expression on the right will not be evaluated

3.4.6 I/O operators

The read keyword takes no operands and reads input from standard input, up to the first newline character. Its expression value is the string that was read, except for the terminating newline.

The print operator prints its operand to standard output and returns the operand. The operand must be a string, any other types must be converted. Note that the print operator has lower precedence than most other operators, so an expression such as

```
print "You have $" + float_to_string _player.money + " remaining"
```

will print exactly what you would expect.

3.4.7 Member Access Operator

The dot (".") operator is a special operator, in that it doesn't take two operands of matching types.

Generally, the dot is used to access a specific member of a larger object. It is used in two different ways:

To access the member of an array:

Players.(1), accesses the player at element

3.4.8 Type Conversions

The following operators return the value of their operand, but with a different type.

to_string - converts integer or float to a string

to_int - converts string or float to a string. The float has its decimal component truncated.

to_float - converts an int or string to a float

3.5 Statements

3.5.1 Conditional Statements

The if/else statement is used to conditionally execute a block of code.

The structure of an if/else statement is:

<if-stmt> -> if (<conditional>) { <stmt-list> } <else-stmt>

<else-stmt> -> <nothing> | else { <stmt-list> } | else <if-stmt>

<conditional> -> <boolean-expr>

So the following are valid if statements or if/else compound statements:

```
if (happy)
{
        print "not happy";
}

if( poet == "Vogon" )
{
        print "plug your ears"
}
else if (poet == "Grunthos The Flatulent")
{
```

```
        print "head for a different planet"
}
else
{
        print "Poem is safe"
}
```

3.5.1.1 Loops

The only loop in Bogus is the while loop.

The structure of a while loop is:

<while-loop> -> while (<conditional>) { <stmt-list> }

When the while loop is reached, the conditional is evaluated. If the value is true, the statements in <stmt-list> are evaluated. After the last statement, the conditional is evaluated again, and so on.

Example:

```
while( val < 6 * 9 )
{
        print "not there yet"
        val = val + 1;
}
```

3.5.2 draw keyword

The draw keyword draws the next card from the deck, and executes the Rule associated with that card.

Conceptually, the deck is an array of Cards, and the game engine remembers the index of the last card "drawn". Each time draw is called, the following occurs:

```
If index is equal to last element in array
        Shuffle deck
        set index equal to -1
Increment index
Execute rule at the card pointed to by index.
```

A draw statement can occur inside any rule. When it occurs, the next card is drawn, its rule is executed, and then execution resumes with the statement following the draw.

3.5.3 exit keyword

The exit keyword immediately terminates the program and causes the interpreter to exit.

3.6 Scoping rules

All variables in Bogus are lexically scoped. Each variables is visible from the point when then they are first declared. Variables remain visible until the end of the block in which they were declared. The outermost scope, which is outside of any braces, is referred to here as "global scope".

The body of a Rule is also treated as a regular block. Any variables that have been declared in global scope prior to the Rule definition are visible within that rule.

3.6.1 Masking prior declarations

An inner scope may declare a variable with the same name as as variable that already exists in an enclosing scope. When this occurs, the variable in the inner scope masks the variable in the outer scope until the end of the innermost enclosing block.

Note that the inner variable declaration must be preceded by the "local" keyword; otherwise, the parser has no way to know that you're attempting to declare a local variable, not just assign a new value to the variable in the outer scope.

This is best clarified by an example:

```
my_var = 6
if(true)
{
        print my_var;               # prints 6
        my_var = my_var * 9;        # change value of outer my_var to 54
        print my_var;               # prints 54
        local my_var = 32;          # masks outer my_var, sets local copy of my_var to 32
        print my_var;               # prints 32
}
print my_var;                       # prints 54
```

Notes on usage of the "local" keyword:

1. Placing the local keyword before a variable declaration in global scope is valid syntax, but the local keyword is ignored.
2. Placing the local keyword before an assignment statement that is not a variable declaration -- that is, when assigning to a variable that has already been declared in the innermost enclosing scope -- is a syntax error.

3.7 Rules

A Rule is a procedure that can be associated with a Space or a Card. Each rule is assigned to one or more spaces or cards. Rules are automatically invoked by the game engine as described in the "Gameplayer" section below.

The structure of a Rule is:

<rule-def> -> Rule <rule-name> { <stmt-list> }

The statements in the Rule's body are evaluated in order until the last statement is complete ("falling off the end") or until the "end" statement is encountered.

Valid <rule-name>s are any valid identifier. Note that there are two special Rule names:

- start_game - which is executed once at the beinning of the game before any other rules
- start_turn - which is evaluated at the beginning of each turn (see "Gameplay" section)

3.8 Structure Of A Bogus Program

A Bogus program contains the following elements in order:

1. Definitions of composite types:
    1. Player
    2. Space
    3. Card
2. Global variable declarations
3. Definition of start_game rule
4. Definition of start_turn rule
5. Definitions of all other rules

3.8.1 Composite Type Definitions

All composite types must be defined, in order, at the beginning of the program. If desired, a composite type maybe defined to have no members (other than the built-in members). The types must be defined in this order:

```
Player has colour(string), money (float);
Space has price(float), rent(float);
Card has nothing;              # Empty definition
```

3.8.2 Global variables

All global variables must be declared and initialized by an assignment expression.

3.8.3 Rules

Rules may be declared in any order, except for start_game and start_turn must be the first two, and must appear in that order.

3.9 Gameplay

When the program executes, the following algorithm will be executed by the interpreter:

3.9.1 Initialization

All arrays will be initialized as empty.

All global variables will be created and initialized.

Magic variables will be initialized as follows:

_player = NoPlayer

_next = NoPlayer

_space = NoSpace

The start_game Rule will be executed. When the start_game rule is complete, _player must not be equal to NoPlayer, or the game will exit with an error.

3.9.2 Taking turns

The following loop will repeat until an error occurs or until an exit statement is executed:

1. Execute start_turn rule
2. If _space has been assigned to since start_turn was called, execute Rule _space._rule

Note that the game engine never automatically changes _player (current player); this must be done explicitly by a Rule.

4 Project Plan

The project started with the initial idea, and was moved forward by sketching out implementations of the board games Candyland and Trivial Pursuit. Completing those sketches led to all the details of the language as listed in the reference manual. The next step was to write a scanner that could handle the basic skeleton of a program -- the composite type declartions, global declarations and rules. Next came implementing the call to the start_name rule, which meant that basic Bogus program could now run.

Once basic Bogus programs could run, I wrote the very simple test suite and some tests of basic expressions and control flow statements. The test suite was particularly helpful when writing the code to handle the local and global symbol tables -- first I wrote the tests, then continued to tweak the code until all variables were resolved as expected, resulting in the expect output.

The remaining step would have been to implement the actual game play engine the moves the game along and invokes rules as needed, as well as the code to provide the values of the "magic" globals as game play advances.

5 Architecture

The entire program is just a few components:

1. scanner.mll - scans and tokenizes the input programmer
2. parser.mly - reads tokens from scanner and builds abstract syntax tree
3. interpret.ml - traverses syntax tree, does all type checking, and builds symbol tables of variables and rules. Then it begins the game engine, which successively invokes the rules as necessary until the game ends.

6 Testing

The testing suite was very simple:

The script run-tests.sh reads all .bog files in the test/ subdirectory, runs them using the Bogus interpreter, and compares the output to a manually-verified output file. Each output file has the same name as the test program it is for, but with the ,out extension.

Here's the output of a run of run-tests.sh:

```
[cary@epistrophy bogus]$ ./run-tests.sh
bin-and     PASSED
bin-not     PASSED
bin-or      PASSED
compare     PASSED
convert     PASSED
dup-rules   PASSED
empty       PASSED
end         PASSED
if1         PASSED
nested-if   PASSED
start       PASSED
unary-neg   PASSED

All tests SUCCEED
```

Here are some example tests:

bin-not.bog:

```
Player has none;
Space has none;
Card has none;

rule start_game
{
    a = 1;
   ## Tests ! operator, should print FALSE
    print "Testing not test: ";
    if( !true )
    {
        print "TRUE\n";
    }
    else
    {
        print "FALSE\n";
    }

   ## Tests ! operator, and order of operations. should print TRUE
    ## Expression is equivalent to (! (3 == 2))
    print "Testing not test: ";
    if( ! 3 == 2)
    {
        print "TRUE\n";
    }
    else
    {
        print "FALSE\n";
    }

    ## Tests ! operator, and order of operations. should print TRUE
    ## Expression is equivalent to ( (! (3 == 2)) || true )
    print "Testing not test: ";
```

```
    if( ! 3 == 2 || true)
    {
        print "TRUE\n";
    }
    else
    {
        print "FALSE\n";
    }
}
```

convert.bog :

```
Player has none;
Space has none;
Card has none;

rule start_game
{

    ## String conversions
    int1 = 42;
    float1 = 6.9;
    bool1 = false;
    str1 = "Don't Panic";

    # Test int
    print to_string(int1) + "\n";
    print to_string(42)   + "\n";

    # Test float
    print to_string(float1) + "\n";
    print to_string(6.9)    + "\n";

    # Test bool
    print to_string(bool1) + "\n";
    print to_string(false) + "\n";
    print to_string(true)  + "\n";

    ## Int conversions
    # string -> int
    if(int1 == to_int("42")) # Should be equal
    {
        print "int1 == to_int(\"42\")\n";
    }
```

```
else
    {
        print "int1 != to_int(\"42\")\n";
    }

    if(int1 == to_int("43"))  # Shouldn't be equal
    {
        print "int1 == to_int(\"42\")\n";
    }
    else
    {
        print "int1 != to_int(\"42\")\n";
    }

   # float -> int
    if(int1 == to_int(42.9)) # Should be equal
    {
        print "int1 == to_int(42.9)\n";
    }
    else
    {
        print "int1 != to_int(42.9)\n";
    }
   if(-17 == to_int(-17.1))  # Shouldn't be equal
    {
        print "-17 == to_int(-17.1)\n";
    }
    else
    {
        print "-17 == to_int(-17.1)\n";
    }
   # This should fail with exception
    print to_int("7a");
}
```

7 Lessons Learned

Despite my best efforts to keep the language small, it seemed to keep growing as I tried to give it enough features to actually make it work. By the time it was done, I didn't have enough time to complete the implementation. In retrospect, especially given the limited time I had available, I would have chosen a simpler idea for the language. Even a simple board game language needed a lot of features to make it practical to write an actually interesting game.

8 Appendix - Code Listing

ast.mli:

```
type op = Add | Sub | Mult | Div |  Equal | Neq | Less | Leq |
Greater | Geq


type lazyop = And | Or


type expr =
    IntLit of int
  | BoolLit of bool
  | StringLit of string
  | FloatLit of float
  | Id of string
  | Binop of expr * op * expr
  | LazyBinop of expr * lazyop * expr
  | Assign of string * expr * bool
  | Call of string * expr list
  | Draw
  | Read
  | Noexpr
  | Negate of expr
  | ToString of expr
  | ToInt of expr
  | ToFloat of expr
  | Not of expr


type stmt =
    Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Print of expr
  | End
```

```
  | Exit

type t =
    Bool
  | Int
  | Float
  | String

type rule_decl = {
    rname : string;
    (* locals : string list; *)
    body : stmt list
}

type comp_decl_member = {
    mname : string ;
    mtype : t
}

type comp_decl = {
    cname : string ;
    cmembers : comp_decl_member list
}

type program = comp_decl list * stmt list * rule_decl list
```

scanner.mll:

```
{ open Parser }

rule token = parse
  [ ' ' '\t' '\n' ] { token lexbuf }
| '#'       { comment lexbuf }
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '"'       { string_lit "" lexbuf }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }
| "bool"    { BOOL }
| "draw"    { DRAW }
| "else"    { ELSE }
| "end"     { END }
| "exit"    { EXIT }
| "false"   { FALSE }
| "float"   { FLOAT }
```

```
| "has"    { HAS }
| "if"     { IF }
| "int"    { INT }
| "local"  { LOCAL }
| "none"   { NONE }
| "print"  { PRINT }
| "read"   { READ }
| "rule"   { RULE }
| "string" { STRING }
| "true"   { TRUE }
| "while"  { WHILE }
| "to_string" { TO_STRING }
| "to_int" { TO_INT }
| "to_float" { TO_FLOAT }
| [ '0'-'9' ]+ as value { INTLIT(int_of_string(value)) }
| [ '0'-'9' ]+'.'[ '0'-'9' ]* as value
{ FLOATLIT(float_of_string(value)) }
| [ '0'-'9' ]*'.'[ '0'-'9' ]+ as value
{ FLOATLIT(float_of_string(value)) }
| [ 'a'-'z' 'A'-'Z'][ 'a'-'z' 'A'-'Z' '_' '0'-'9' ]* as value
{ ID(value) }
| eof      { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }


and string_lit str = parse
  "\\"                      { escaped_char str lexbuf }
| "\""                      { STRINGLIT(str) }
| "\n"                     {raise (Failure("multi-line string
literals not
allowed, input character #" ^ (string_of_int (Lexing.lexeme_start
lexbuf)))) }
|  [^ '\\' '\n' '"' ]+ as part    { string_lit  (str ^ part) lexbuf }


and escaped_char str = parse
  [ '"' ]     { string_lit (str ^ "\"") lexbuf }
```

```
| [ 'n' ]     { string_lit (str ^ "\n") lexbuf }
| [ 'r' ]     { string_lit (str ^ "\r") lexbuf }
| [ 't' ]     { string_lit (str ^ "\t") lexbuf }
| eof          { raise (Failure("Unterminated string literal")) }
| _ as char    { raise (Failure("Invalid escape character \"\\" ^
Char.escaped char)) }


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

parser.mly:

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token AND OR NOT
%token EQ NEQ LT LEQ GT GEQ
%token PRINT READ
%token TO_STRING TO_INT TO_FLOAT
%token IF ELSE FOR WHILE INT DRAW END EXIT FALSE HAS LOCAL TRUE RULE
NONE
%token BOOL INT FLOAT STRING
%token <int> INTLIT
%token <string> STRINGLIT
%token <float> FLOATLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%left ASSIGN
%left AND OR
%left NOT
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left NEG

%start program
%type <Ast.program> program

%%
```

```
program:
    all_comp_decls global_decls rule_decls { $1, $2, $3 }

all_comp_decls:
    comp_decl1 comp_decl1 comp_decl1 { [ $1 ; $2 ; $3 ] }

comp_decl1:
    ID HAS comp_member_list_opt SEMI
        { { cname = $1;
            cmembers = $3 } }

comp_member_list_opt:
    NONE { [] }
  | comp_member_list { List.rev $1 }

comp_member_list:
    comp_member     { [$1] }
  | comp_member_list COMMA comp_member { $3 :: $1 }

comp_member:
    ID LPAREN prim_type RPAREN { { mname = $1; mtype = $3 } }

prim_type:
    BOOL    { Bool }
  | INT     { Int }
  | FLOAT   { Float }
  | STRING  { String }

global_decls:
    { [] }
  | assign_list     { List.rev $1 }

assign_list:
```

```
    assignment              { [$1] }
  | assign_list assignment { $2 :: $1 }

assignment:
    ID ASSIGN expr SEMI   { Expr(Assign ($1, $3, false)) }

rule_decls:
    { [] }
  | rule_decls rule_decl { $2 :: $1 }

rule_decl:
   RULE ID LBRACE stmt_list RBRACE
     { { rname = $2;
         body = List.rev $4 }}

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

stmt:
    expr SEMI { Expr($1) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | PRINT expr SEMI      { Print($2) }
  | END SEMI { End }
  | EXIT SEMI { Exit }

expr:
    INTLIT           { IntLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | STRINGLIT        { StringLit($1) }
```

```
| FLOATLIT         { FloatLit ($1) }
| ID               { Id($1) }
| READ             { Read }
| TO_STRING LPAREN expr RPAREN   { ToString($3) }
| TO_INT LPAREN    expr RPAREN   { ToInt($3) }
| TO_FLOAT LPAREN  expr RPAREN   { ToFloat($3) }
| 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 { LazyBinop($1, And,   $3) }
| expr OR     expr { LazyBinop($1, Or,   $3) }
| NOT expr         { Not ($2) }
| MINUS expr %prec NEG { Negate ($2) }
| ID ASSIGN expr   { Assign($1, $3, false) }
| LOCAL ID ASSIGN expr { Assign($2, $4, true) }
| LPAREN expr RPAREN { $2 }
```

interpret.ml:

```
open Ast

type exvals = (* expression value *)
    VInt of int
  | VFloat of float
  | VString of string
  | VBool of bool
  | VNothing

let exval_to_string ex = match (ex) with
  VInt ex -> string_of_int ex
| VFloat ex -> string_of_float ex
| VString ex -> ex
| VBool ex -> if ex = true then "true" else "false"
| VNothing -> "nothing";;

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



type exval = { v: exvals ; t : t }


(* exception ReturnException of int * int NameMap.t *)
exception ReturnException of exvals * exvals NameMap.t

(* Main entry point: run a program *)

let run (program) =
  let (comps,globals,rules) = program
```

```
  in

  (* Put rule declarations in a symbol table *)
  let rule_decls = List.fold_left
      (fun rules rdecl ->
          if NameMap.mem rdecl.rname rules then
              raise(Failure("Error: multiple declarations for rule "
^
              rdecl.rname))
          else
              NameMap.add rdecl.rname rdecl rules
      ) NameMap.empty rules

  in

  (* Invoke a function and return an updated global symbol table *)
  let rec call rdecl globals =

    (* Evaluate an expression and return (value, updated environment)
*)
    let rec eval env = function
       IntLit(l) -> VInt l , env
      | BoolLit(l) -> VBool l, env
      | StringLit(l) -> VString l, env
      | FloatLit(l) -> VFloat l, env
      | Noexpr -> VNothing, env (* must be non-zero for the for loop
predicate *)
      | Id(var) ->
       let locals, globals = env in
       if NameMap.mem var locals then
         (NameMap.find var locals), env
       else if NameMap.mem var globals then
         (NameMap.find var globals), env
       else raise (Failure ("undeclared identifier " ^ var))
      | LazyBinop(e1, op, e2) ->
```

```
       let v1, env = eval env e1 in
          (match op with
            And -> (match v1 with
                VBool(true) ->
                    let v2, env = eval env e2 in
                    (match v2 with
                        VBool(true)  -> VBool(true), env
                      | VBool(false) -> VBool(false), env
                      | _ -> raise (Failure("Only bool expressions
can be used with &&
                        operator")) )
              | VBool(false) -> VBool(false), env
              | _ -> raise (Failure("Only bool expressions can be
used with &&
             operator")) )

          |  Or -> (match v1 with
            VBool(false) ->
                let v2, env = eval env e2 in
                (match v2 with
                    VBool(true)  -> VBool(true), env
                  | VBool(false) -> VBool(false), env
                  | _ -> raise (Failure("Only bool expressions can
be used with &&
                    operator")) )
              | VBool(true) -> VBool(true), env
              | _ -> raise (Failure("Only bool expressions can be
used with &&
             operator")) )
          )
      | Binop(e1, op, e2) ->
       let v1, env = eval env e1 in
          let v2, env = eval env e2 in
       (match op with
         Add -> (match (v1,v2) with
```

```
                (VInt v1, VInt v2) -> VInt (v1 + v2)
              | (VFloat v1, VFloat v2) -> VFloat (v1 +. v2)
              | (VString v1, VString v2) -> VString (v1 ^ v2)
              | (VBool _ , _) -> raise (Failure ("Invalid argument to
+ operator"))
              | (_,_) -> raise (Failure("Operands to + operator don't
have matching
    types")))

       | Sub -> (match (v1,v2) with
                (VInt v1, VInt v2) -> VInt (v1 - v2)
              | (VFloat v1, VFloat v2) -> VFloat (v1 -. v2)
              | ( (VInt _ | VFloat _), _) -> raise (Failure("Operands
to - operator don't have matching
    types"))
              | (_,_) -> raise (Failure ("Invalid argument to -
operator")))

       | Mult -> (match (v1,v2) with
                (VInt v1, VInt v2) -> VInt (v1 * v2)
              | (VFloat v1, VFloat v2) -> VFloat (v1 *. v2)
              | ( (VInt _ | VFloat _), _) -> raise (Failure("Operands
to * operator don't have matching
    types"))
              | (_,_) -> raise (Failure ("Invalid argument to *
operator")))

       | Div -> (match (v1,v2) with
                (VInt v1, VInt v2) -> VInt (v1 / v2)
              | (VFloat v1, VFloat v2) -> VFloat (v1 /. v2)
              | ( (VInt _ | VFloat _), _) -> raise (Failure("Operands
to * operator don't have matching
    types"))
              | (_,_) -> raise (Failure ("Invalid argument to *
operator")))
```

```
      | Equal -> (match(v1,v2) with
                 (VInt v1, VInt v2) -> VBool (v1 = v2)
               | (VFloat v1, VFloat v2) -> VBool (v1 = v2)
               | (VBool v1, VBool v2) -> VBool (v1 = v2)
               | (VString v1, VString v2) -> VBool (v1 = v2)
               | (_,_) -> raise (Failure ("Trying to compare values of
different
               types")))

      | Neq -> (match (v1,v2) with
                 (VInt v1, VInt v2) -> VBool (v1 <> v2)
               | (VFloat v1, VFloat v2) -> VBool (v1 <> v2)
               | (VBool v1, VBool v2) -> VBool (v1 <> v2)
               | (VString v1, VString v2) -> VBool (v1 <> v2)
               | (_,_) -> raise (Failure ("Trying to compare values of
different
               types")))

      | Less -> (match (v1,v2) with
                 (VInt v1, VInt v2) -> VBool (v1 < v2)
               | (VFloat v1, VFloat v2) -> VBool (v1 < v2)
               | ( (VString _ | VBool _), _) -> raise
(Failure("Invalid type for
               comparison"))
               | (_,_) -> raise (Failure ("Trying to compare values of
different
               types")))

      | Leq -> (match (v1,v2) with
                 (VInt v1, VInt v2) -> VBool (v1 <= v2)
               | (VFloat v1, VFloat v2) -> VBool (v1 <= v2)
               | ( (VString _ | VBool _), _) -> raise
(Failure("Invalid type for
               comparison"))
               | (_,_) -> raise (Failure ("Trying to compare values of
different
```

```
            types")))

       | Greater -> (match (v1,v2) with
                (VInt v1, VInt v2) -> VBool (v1 > v2)
              | (VFloat v1, VFloat v2) -> VBool (v1 > v2)
              | ( (VString _ | VBool _), _) -> raise
(Failure("Invalid type for
              comparison"))
              | (_,_) -> raise (Failure ("Trying to compare values of
different
              types")))

       | Geq -> (match (v1,v2) with
                (VInt v1, VInt v2) -> VBool (v1 >= v2)
              | (VFloat v1, VFloat v2) -> VBool (v1 >= v2)
              | ( (VString _ | VBool _), _) -> raise
(Failure("Invalid type for
              comparison"))
              | (_,_) -> raise (Failure ("Trying to compare values of
different
              types")))
        ), env
      | Negate(e) ->
            let v, (locals, globals) = eval env e in
            (match v with
            (VInt v1)    -> VInt(- v1)
          | (VFloat v1)  -> VFloat(-. v1)
          | _           -> raise(Failure("Attempting to negate non-
numeric
          value"))
            ), env
      | Not(e) ->
            let v, (locals, globals) = eval env e in
            (match v with
            (VBool v1)    -> VBool(not v1)
```

```
          | _           -> raise(Failure("Trying to apply boolean !
to
          non-boolean value"))
            ), env
      | Assign(var, e, local) ->
          let do_assign (vname, exp, map) = (
              if (not (NameMap.mem vname map)) then  (* New
declaration *)
                NameMap.add var exp map
              else
                (match (NameMap.find vname map, exp) with
            (VInt v1, VInt v2)          ->  NameMap.add var exp map
          | (VFloat v1, VFloat v2)      ->  NameMap.add var exp map
          | (VString v1, VString v2)    ->  NameMap.add var exp map
          | (VBool v1, VBool v2)        ->  NameMap.add var exp map
          | (VNothing, _)               ->  NameMap.add var exp map
          |  (_,_) -> raise (Failure ("Type error: attempting to
assign
            incompatible types"))) (* end match *)
          ) (* end do_assign *)
          in

       let v, (locals, globals) = eval env e in
          if local then
                if NameMap.mem var locals then
                    raise (Failure ("Tried to redeclare local
variable"))
                else (* need to check type before assigning *)
                    (* v, (NameMap.add var v locals, globals) *)
                v, (do_assign (var, v, locals), globals)
       else if NameMap.mem var locals then  (* Assign to local *)
                v, (do_assign (var, v, locals), globals)
          else if NameMap.mem var globals then   (* Assign to global
*)
                v, (locals, do_assign(var, v, globals))
```

```
          else                                (* Declare new local *)
               v, (do_assign (var, v, locals), globals)
    | ToString(e) ->
            let v, env = eval env e in
            (match v with
            (VInt e) -> VString(string_of_int e), env
          | (VFloat e) -> VString(string_of_float e), env
          | (VBool e) -> VString(if e = true then "true" else
"false"), env
          | (VString e) -> VString(e), env
          | _ -> raise (Failure("Type error: invalid type passed to
to_string"))
            )
    | ToInt(e) ->
            let v, env = eval env e in
            (match v with
            (VInt e) -> VInt(e), env
          | (VFloat e) -> VInt(truncate e), env
          | (VString e) -> VInt(int_of_string e), env
          | _ -> raise (Failure("Type error: invalid type passed to
to_int"))
            )
    | ToFloat(e) ->
            let v, env = eval env e in
            (match v with
            (VInt e) -> VFloat(float e), env
          | (VFloat e) -> VFloat(e), env
          | (VString e) -> VFloat(float_of_string e), env
          | _ -> raise (Failure("Type error: invalid type passed to
to_float"))
            )
    | Read -> VString(input_line stdin), env
    | Call(_,_) -> raise(Failure("Call() not implemented"))
    | Draw      -> raise(Failure("Draw not implemented"))
```

```
    in

    (* Execute a statement and return an updated environment *)
    let rec exec env = function
     Block(stmts) -> List.fold_left exec env stmts
      | Expr(e) -> let _, env = eval env e in env
      | If(e, s1, s2) ->
       let v, env = eval env e in
          (match v with
          VBool b -> exec env (if b then s1 else s2)
        | _        -> raise(Failure("Invalid condition: all
conditionals must
          have boolean values")))

      | While(e, s) ->
       let rec loop env =
         let v, env = eval env e in
          (match v with
            VBool b -> if b then loop (exec env s) else env
          | _       -> raise(Failure("Invalid condition: all
conditionals must
                      have boolean values")))
       in loop env
      | Print(e) ->
              let str,env = eval env e
              in
              (match (str) with
              (VString str) -> print_string str ; flush stdout; env
            | _ -> print_string "Not Printing String" ; raise
(Failure("Trying to a print a value that isn't a
            string")) )
      | End  ->
              let v, (locals, globals) = eval env Noexpr in (* Do
nothing, but get globals *)
              raise(ReturnException(v,globals))
```

```
      | Exit ->
              print_string "Exit called\n" ; exit 0; NameMap.empty,
NameMap.empty
    in


    (* Enter the function: bind actual values to formal arguments *)
    (* Execute each statement in sequence, return updated global
symbol table *)
    snd (List.fold_left exec (NameMap.empty, globals) rdecl.body)


  (* Run a program: initialize global variables to VNothing, find and
run "main" *)
  in let globs =
      List.fold_left (fun gmap x ->
          (match x with
            Expr(Assign(var,_,_)) -> NameMap.add var VNothing gmap
          | _                     -> raise(Failure("Fatal error: global
declarations
            contain a statement that isn't an assignment")) )
      ) NameMap.empty globals


  in let globs =
      call {rname = ""; body = globals} globs
  (* in
     let _ = NameMap.iter (fun k v -> print_endline k ) globs *)
  in
    if NameMap.mem "start_game" rule_decls then
    try
      call (NameMap.find "start_game" rule_decls) globs
    with
        ReturnException(v,globals) -> (globals)
    else
      raise(Failure("start_game rule not defined"))
      (*
  in try
```

```
    call (NameMap.find "main" func_decls) [] globals
  with Not_found -> raise (Failure ("did not find the main()
function"))
  *)
```

bogus.ml:

```
let print = false

let _ =
    if Array.length Sys.argv != 2 then
        (print_endline ("usage: " ^ Sys.argv.(0) ^ " source-file") ;
exit 0)



let _ =
  let sourcefile = Sys.argv.(1) in
  let infile = open_in sourcefile in
  (*let lexbuf = Lexing.from_channel stdin in *)
let lexbuf = Lexing.from_channel infile in
  let program = Parser.program Scanner.token lexbuf in

    ignore (Interpret.run program)
```