Fantasy League Object Oriented Development (FLOOD)

Project Report

COMS W4115: Programming Language & Translators

Team 9
is

Stephanie Aligbe
System Tester
sna2111@columbia.edu

Elliot Katz
Project Manager
epk2102@columbia.edu

Tam Le
System Architect
tvl2102@columbia.edu

Dillen Roggensinger
System Integrator
der2127@columbia.edu

Anuj Sampathkumaran
Language Guru
as4046@columbia.edu

May 10, 2011

Contents

1

Introduction

As defined on Wikipedia: *fantasy sport (also known as* ***rotisserie****,* ***roto****, or* ***owner simulation****) is a game where participants act as owners to build a team that competes against other fantasy owners based on the statistics generated by the real individual players or teams of a professional sport.*

The popularity of fantasy sports has exploded in recent years. A 2007 study by the *Fantasy Sports Trade Association (FSTA)* estimated nearly 30 million people in the U.S. and Canada, ranging in age from 12 and above, participated in organized fantasy sports leagues. In comparison, an estimated 3 million people played in the early 1990s, swelling to 15 million by the early 2000s. This impressive growth looks to continue since the study revealed teenagers in both the U.S. and Canada play fantasy sports at a higher rate than the national average, with 13 percent of teens playing in the U.S. and 14 percent playing in Canada.

The FSTA study also estimated the spending habits and overall economic impact of fantasy sports players. Consumers engaging in this ever growing hobby spent $800 million directly on fantasy sports products and an additional $3 billion worth of related media products (such as DirecTV's NFL Sunday Ticket and satellite radio's coverage of MLB). Moreover, the growing popularity of fantasy sports is not restricted to North America alone.

A recent 2008 study by a European-based market research company estimated the number of fantasy sports players in Britain range between 5.5 and 7.5 million and vary in age between 16-64, of which 80 percent participated in fantasy soccer.

Thus, the **FLOOD** programming language is targeted to address a specific problem domain: the creation of fantasy gaming league applications. From the ground up, the language is designed to make it as straitforward as possible for programmers to create fantasy gaming applications. This task will entail defining a league and its type (sports, financial markets, election polls, etc.), establish the rules of governance, enumerate the users/teams and individual league players, and set various other control parameters.

In this sense, **FLOOD** is very much a "high-level" and "domain-specific" language. Such as **R** and **S** are languages designed specifically to perform statistics calculations, **SQL** for relational database queries, and **Mathematica** and **Maxima** for symbolic mathematics, **FLOOD** is dedicated to solving the particular problem of fantasy gaming. Given the ubiquity of fantasy sports as a recreational hobby for millions of people, the creation of a domain-specific language to attack this problem in a clear and concise native programming etymology, as opposed to the application of a general purpose language such as **C** and **Java**, is a challenging and worthwhile undertaking.

2

A Tutorial for Getting Started Quickly

The goal of this tutorial is to jump into the **FLOOD** language by creating a simple fantasy league. We won't concern ourselves with implementation details just yet. The language is meant to be simple yet sufficiently extensible but the tutorial will not delve into the various features that make this possible.

Experienced Java programmers will be familiar with most of the syntax since it is a subset of the popular C-like syntax. In certain instances we have opted to use Python style syntax for improved readability or Pascal syntax for ease of use. Where possible we compare and contrast a specific feature with Java, Python or some other language which comprise the main influence of **FLOOD**.

Some examples which aren't strictly necessary for the sample program are included to clarify points of possible confusion. These are noted when used. Additionally, we repeat portions of the **FLOOD** standard library for ease of reading. The full library is included in the *Reference Manual* and *Appendix* where appropriate. Please be aware of the distinction made between programmer, the person actually writing a **FLOOD** program (most likely yourself if you are reading this), and *User*, the end-user of the application (which could be

thought of as a team entitity). Additionally, *Player* refers to a player of the fantasy league, whether this is an actual person as in the case of a quarterback in football or a non-human entitity as is the case with *AAPL* (Apple) in the stock market.

2.1 Getting Started - The "Hello World" Program

FLOOD has a very specific application domain. It's goal is to create fantasy leagues and thus the ability to print to console is fruitless. However, the GUI will have a message box where messages can be displayed to the *User*. **FLOOD** hides the details of the user-interface so the programmer can concentrate on the rules of their specific fantasy league. While the first simple program may seem more complicated than the simple Java "hello world" example, most programs generally will not be any more elaborate than this case.

Since **FLOOD** is built on top of Java byte-code, the programmer will not need to worry about system compatibility issues. Heeding the eternal wisdom of Kernighan & Ritchie, we also point out that compilation will only succeed if the programmer hasn't botched anything, such as missing characters, misspelled keywords, or some other mistake that can cause an error in compilation.

Let's start by showing what the *main* class will look like. Defining the *main* class should look somewhat similar to those familiar with Python.

Listing 2.1: Sample.fld

```
1  /*
2  * Define the league and configure various settings
3  */
4  DefineLeague
5      /* Set league parameters */
6      Set LeagueName("Happy League");
7      ...
8      ...
9      Set MinTeamSize(5);
10
11 /*
12 * Define custom functions
```

```
*/
DefineFunctions
    Void myFunction1(Str param1, Int param2)
    {
        ...
        ...
    }

    ...

    Void myFunction2(Bool param1, Flt param2)
    {
        ...
        ...
    }
```

Let's now review the notable differences between Python, Java and **FLOOD**. First, the convention in **FLOOD** is to name each file with the *.fld* extension. In addition, each file is a self-contained program and unlike Java, there is no ability to add separate classes. And finally, the first letter of any keyword in **FLOOD** is capitalized.

The language was originally intended to be object-oriented in order to facilitate modularity and extensibility of code. However, due to the restricted time constraints imposed by the course loads of various team members, it was decided to reduce the scope of the language without losing the basic functionalities as originally promised. Therefore, the abilities to add classes and inherit from pre-existing libraries were left for a possible future iteration of the language.

The entry point for a **FLOOD** program is to specify the `DefineLeague` keyword. Since there is no question of class scope within a file, code below this program block is considered part of the program while any code above (excluding comments, naturally) is in error. It is important to note the `DefineLeague` block is semantically equivalent to the `main` function in a standard Java program and all the parameters and settings specified within this block can be viewed as passing parameters via the constructor method in a Java program. Moreover, it is not possible to declare local variables here since meaningful

computations do not exist in this section of a **FLOOD** program.

Listing 2.2: Defining the league and setting parameters

```
/*
* Define the league and configure various settings
*/
DefineLeague
   /* Set league parameters */
   Set LeagueName("Happy League");
   Set MaxUser(10);
   Set MinUser(12);
   Set MaxTeamSize(10);
   Set MinTeamSize(5);

   /* Add Users */
   Add User("Eli");
   Add User("Dillen");
   Add User("Anuj");
   Add User("Tam");
   Add User("Steph");

   /* Add Actions */
   Add Action("Field Goal Attempt", -0.45);
   Add Action("Field Goal Made", 1.0);
   Add Action("Free Throw Attempt", -0.75);

   /* Add Players */
   Add Player("Lebron James", "forward");
   Add Player("Chris Bosh", "forward");
   Add Player("Dwyane Wade", "guard");
```

Every **FLOOD** program implicitly instantiates a *League* object. The *League* object contains most of the other objects needed for the **FLOOD** program including *User*, *Player* and *Action* lists.

For the minimum program it may be possible to leave the maximum and mininum user settings at their respective default values, but as an example we set them above. In order to change certain attributes of the *League* use the `Set` keyword with the attribute to be changed. Additionally, to create new *Users*, *Actions* and *Players* use the `Add` keyword. Since local variables can not be set in the *League* definition, all attributes must be literals.

Note that *Users* and *Players* are simply a *string* representing a name while `Action` is a name-value pair consisting of the name of the `Action` and the value that will be added to a *User's* points through that `Action`.

A **FLOOD** developer need not worry about passing any values to the back-end—**FLOOD** encapsulates passing the *League* to the GUI and calling the GUI.

Listing 2.3: Defining functions

```
/*
* Defining custom user functions
*/
DefineFunctions
    Void myFunction1(Str param1, Int param2)
    {
        ...
        ...
    }

    ...

    Void myFunction2(Bool param1, Flt param2)
    {
        ...
        ...
    }
```

Before custom functions can be defined, the keyword `DefineFunctions` must be specified. Certain **FLOOD** functions are built into the language and the compiler will check whether the programmer has overridden them. They are as follows:

Listing 2.4: Predefined FLOOD functions

```
Int draftFunction(int turn) {...}
Bool draftPlayer(User u, Player p) {...}
Bool trade(User u1, Player[] p1, User u2, Player[] p2) {...}
Bool dropPlayer(User u, Player p) {...}
```

If one of the above-mentioned functions is defined in the source file, then **FLOOD** will simply use the programmer's defined function. However, if any of the functions were left out, then **FLOOD** will generate default code for them (*Note: the appendix has a full listing

for reference). A function in **FLOOD** is similar to a method in Java or a function in C++. We call a function by using the function's name with an optional argument list in between mandatory parenthesis.

In the following example, assume the function was defined as:

Listing 2.5: Predefined FLOOD functions

```
1 Int draftFunction(Int turn)
2 {
3    /* Number of Users is 10 */
4    Int currentTurn;
5    currentTurn = turn % 10;
6    Return currentTurn;
7 }
```

There are several things to note in the above code snippet. **FLOOD** uses /∗ . . . ∗/ comments similar to Java and can span multiple lines. Variables are declared before they are used and all variables are instantiated to default values. Using a variable that hasn't been instantiated will result in a logical error rather than a semantic error. All variable declarations must occur before any other code in the function body.

A `Return` must only occur at the end of the function and must match the stated return type. In the example above, `currentTurn` is declared as an `Int` and then instantiated. Only following the rest of the production body does `currentTurn` gets returned.

A function can be called similarly to Java:

Listing 2.6: Function call

```
1 draftFunction(10);
```

or in an assignment:

Listing 2.7: Function call in an assignment

```
1 Int a;
2 a = draftFunction(10);
```

Hiding the GUI is one of the main features of **FLOOD**. There is no need to program any part of the user-interface. The GUI will know how to hook into the **FLOOD** source and

connect the buttons on the interface to the actions defined by the programmer. The output will be a GUI window which will control the flow of the program and in turn, the flow will depend upon the user's interactions. For instance, the only way to update the scores of the *Users* is by adding player-action files to the program. *Player-Action* files would contain new statistics such as:

LeBron James	7 Rebound
Lebron James	4 Assist
Carmelo Anthony	5 Steal
Carmelo Anthony	25 Point Scored

This is everything needed to run a simple **FLOOD** program. This basketball league will mirror equivalent leagues in *Yahoo! Sports* or *ESPN Fantasy* without network connectivity. The minimum "Hello World" program is included below:

Listing 2.8: Minimal FLOOD program to create a basektball fantasy league

```
1  DefineLeague
2     Set LeagueName("Basketball League");
3     Add User("Anuj");
4     Add User("Tam");
5     Add Action("Field Goal", 2.0);
6     Add Action("Rebound", 1.0);
7     Add Player("Lebron James", "forward");
8     Add Player("Kobe Bryant", "guard");
9     Add Player("Dwight Howard", "center");
10    Add Player("Kevin Durant", "forward");
11
12 DefineFunctions
13   /* None declared */
```

2.2 Variables & Arithmetic Expressions

Just as any robust programming language requires a comprehensive computational model, **FLOOD** provides the user with a set of arithmetic expressions and variable types to work with. To use a variable and work with it, the variable must be declared before it is used for

the first time. A declaration defines the properties of the variables. A declaration is of the form *type* followed by the *name* of the variable to be declared of that type.

Listing 2.9: Variable declarations

```
/* variable declarations */
Int i;
Bool b;
Flt f = 1.2; /* ...assigning value at point of declaration */
Str s;

/* ...or assigning values at a later point. */
i = 1;
b = True;
f = 2.1;
s = "Hello World";
```

Here, variables are declared before they are used. It's also possible to assign values to the variables in the declaration. An important distinction between **FLOOD** and a programming language like Java is the location of actual declaration which, as noted ***must be at the top*** of the function body. Following the declarations, the variables can be used as needed. Note that there is no coercion between `Flt` and `Int` since **FLOOD** does not support implicit type coercions, as shown here:

Listing 2.10: Implicit type coersion is not supported

```
Int i;
Flt f;
i = 2.0; /* Error since 2.0 is a Flt */
f = 1; /* Error since 1 is an Int */
```

More examples of assignment expressions which will throw errors due to mismatch types:

Listing 2.11: More errors due to mistmatch type declarations and assignments

```
Int i;
Int f;
f = 1.0;
i = f / f; /* Error: Int used in a Flt expression */
```

FLOOD offers the set of arithmetic expressions required to create a comprehensive fantasy league of the developer's choice. This set comprises of the standard addition, sub-

traction, multiplication, division and modulus operators. In addition, statements can include functions as in the case:

Listing 2.12: Function used in an arithmetic statement

```
1 Int i;
2 i = someFunction() + 10 * 4; /* someFunction() returns an Int
     */
```

The above code snippet provides a glimpse of the arithmetic capabilities of **FLOOD**. The '+', '−', '/', and '∗' operators are binary operators and can be used to add, subtract, divide and multiply `Flt` (floats) and `Int` (integers).

2.3 Loops & Conditionals

Creating a fantasy league in **FLOOD** can range from simple computations to complex algorithms involved in drafts. To facilitate the latter, a **FLOOD** developer has the choice of using loops to make life easier.

The syntax for the *while* loop follows the standard convention:

Listing 2.13: while loop

```
1 While (total < 4)
2 {
3    centers = centers + 1;
4    guards = guards - centers;
5    forwards = forwards / 2;
6    total + 1;
7 };
```

The *while* loop operates as follows: the condition in parentheses is tested. If it is true (`total` less than 4), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is checked again and if true, the body is executed again. When the test becomes false (`total` equals or exceeds 4) the loop ends and execution continues at the statement immediately following the loop.

FLOOD provides the developer with a conditional in the form of the *if* expression that is defined as follows:

Listing 2.14: If conditional

```
If (position == "center")
{
   position = position + 1;
};
```

Here, the program checks the condition enclosed in the bracket. If this condition is met (in this case, if the person's position is `center`, then the program executes the next statement. **FLOOD** also incorporates $if \ldots else$ conditional statements as follows:

Listing 2.15: If...Else conditional statement

```
If (points > 100)
{
   trade = True;
}
Else
{
   trade = False;
};
```

If the first condition (`points` $>$ 100) is not met, then the statement or statements enclosed in body of the `Else` condition will be executed i.e. assign `False` to the `trade` variable.

2.4 Functions & Scope

As in countless other programming languages, **FLOOD** employs the concept of a function (also referred to as a method, subroutine, or procedure), a logical grouping of code within the larger program which carries out a specific task and is relatively independent of the rest of the code base. The idea of a function is analogous to the notion of the "black box" when discussing the concept of encapsulation in object-oriented programming. For a well-designed function, the particulars of "how" a the function performs its task(s) is not of

critical importance and just knowing "what" it does suffices. Functions can be "called" or "executed" any number of times and from within other functions.

Listing 2.16: Syntax of a FLOOD function

```
1 Bool evaluate(User u, Player p)
2 {
3     ...
4 }
```

An important requirement to remember is that functions ***must be defined before*** they are used, as shown here:

Listing 2.17: A function must be defined before being called

```
1  Bool function1()
2  {
3     Return False;
4  }
5
6  ...
7
8  Bool function2()
9  {
10    Return function1();
11 }
```

In this example, function `function1()` is defined before `function2()` and therefore can be used in the body of `function2()`.

The scope of a variable is limited to the function it is declared in—***variables cannot be declared global***. In order to modify variables between functions they must be passed as parameters to the specific functions.

2.5 Users & Players (Arrays)

The types `User` and `Player` are specific to **FLOOD**. They can only be declared as a formal parameter in the argument lists of functions:

Listing 2.18: User and Player declared in function argument list

```
Bool draftPlayer(User u, Player p) {...}
Bool trade(User u1, Player[] p1, User u2, Player[] p2) {...}
Bool dropPlayer(User u, Player p) {...}
```

The GUI knows to look for these specific functions and populate them correctly. It is possible to define custom functions that take `User` and `Player` as arguments, though **FLOOD** convention recommends against it.

Within **FLOOD**, arrays exists only in the context of `User` and `Player`. Both types can be passed as single values or array types. The array declaration is similar to Java:

Listing 2.19: Passing User and Player arrays

```
Bool trade(User u1, Player[] p1, User u2, Player[] p2) {...}
```

Square brackets are placed after the type before the name of the variable. An array can be accessed in the body of the function using an `Int` index. Both of the examples below are correct:

Listing 2.20: Assessing array elements using index

```
Int i = 5;
draft(a[1]);
draft(b[i]);
```

FLOOD has a few built-in functions that give the programmer more flexibility in writing functions. The functions are associated with `Users` and `Players` and as such are included in this section. The first function `ArrayLength` simply returns the length of the array that is passed to it. Note that Python similarly uses this kind of syntax to find the length of a list. The other two functions `AddPlayer()` and `RemovePlayer()` are functions that alert the GUI to the addition or removal of a `Player` from a `User`. The syntax is as follows:

Listing 2.21: Some FLOOD utility functions and usage

```
Int i;
i = ArrayLength(p);  /* Where p is an array of Players */
AddPlayer(u, p); /* Where u is a User and p is a Player */
RemovePlayer(u, p); /* Where u is a User and p is a Player */
```

2.6 Alert & Error

FLOOD programs are run through a GUI so print statements conform to this interface. As an alternative to print streams to a console, **FLOOD** allows the programmer to display boxes of text. There are two kinds of boxes, *Alert* and *Error*:

Listing 2.22: Launching Alert and Error message boxes

```
1 Alert("Alert Box Title", "Alert message......");
2 Error("Error Box Title", "Error message......");
```

Both message boxes have the same structure. The keyword `Alert` or `Error` must be followed by the standard parenthesis with two arguments. Both arguments must be of type `Str`. The first argument will be the box's title while the second argument will be the message body.

3

Reference Manual

3.1 Introduction

3.2 Lexical Conventions

3.2.1 Tokens

3.2.2 Comments

3.2.3 Identifiers

3.2.4 Keywords

3.2.5 Function Generation

3.3 Syntax Notation

3.3.1 Expressions

Primary Expression

Postfix Expressions

Unary Expression

Multiplicative Expression

3.4 Object Orientated Programming

3.4.1 Abstraction

3.4.2 Encapsulation

3.4.3 Polymorphism

3.5 Grammar

4

Project Plan

5

Language Evolution

6

Language Architecture

7

Development Enviornment

8

Testing

9

Conclusions

9.1 Lessons Learned as a Team

9.2 Lessons Learned by Team Members

9.2.1 Stephanie Aligbe

9.2.2 Elliot Katz

9.2.3 Tam Le

9.2.4 Dillen Roggensinger

9.2.5 Anuj Sampathkumaran

9.3 Advice for Future Teams

9.4 Suggestions for Future Improvement

Appendix A

FLOOD Source Code