Fantasy League Object Oriented Development (FLOOD)

Language Tutorial & Reference Manual

COMS W4115: Programming Language & Translators

Stephanie Aligbe
sna2111@columbia.edu

Elliot Katz
epk2102@columbia.edu

Tam Le
tvl2102@columbia.edu

Dillen Roggensinger
der2127@columbia.edu

Anuj Sampathkumaran
as4046@columbia.edu

March 23, 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 Quick Tutorial

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 syntax for improved readability. Where possible we compare or contrast a specific feature with Java, C++, and Python which are the main influences 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*. 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. 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 would look like. In keeping with the theme of fantasy leagues, the function `main` has been replaced by the equivalent `play`.

Listing 2.1: BasketballPlay.fld

```
1  class BasketballPlay:
2     public play()
3     {
4        League myfbl = new League("Happy League");
5        mybl.setMaxUsers(10);
6        mybl.setMinUsers(4);
7
8        mybl.addUser(new User("Eli"));
9        mybl.addUser(new User("Dillen"));
10       mybl.addUser(new User("Anuj"));
11       mybl.addUser(new User("Tam"));
12       mybl.addUser(new User("Steph"));
13
14       mybl.addAction(new Action("Field Goal Attempt", -0.45);
15       mybl.addAction(new Action("Field Goal Made", 1.0);
16       mybl.addAction(new Action("Free Throw Attempt", -0.75);
17       mybl.addAction(new Action("Free Throw Made", 1.0);
18       mybl.addAction(new Action("3-Point Shot Made", 3.0);
19       mybl.addAction(new Action("Point Scored", 0.5);
```

```
20          mybl.addAction(new Action("Rebound", 1.5);
21          mybl.addAction(new Action("Assist", 2.0);
22          mybl.addAction(new Action("Steal", 3.0);
23          mybl.addAction(new Action("Turnover", -2.0);
24          mybl.addAction(new Action("Blocked Shot", 3.0);
25
26          SnakeDraft blDraft = new SnakeDraft(mybl);
27
28          Flood.launchGui(blDraft, mybl);
29      }
```

At first glance this code example appears to be very similar to a Java program. However, upon further inspection there are notable differences. First, note the convention is to name **FLOOD** files with the *.fld* extension. Secondly, each file must contain one and only one class—as oppose to Java, there is no ability to nest classes.

Additionally, classes are defined using the semicolon. Since there is no question of scope within a file, code below the class definition is considered part of the class. Code above the class definition is an error. Import statement must immediately follow the class definition. Instance variables can be placed anywhere below the class definition outside of method blocks.

Java uses the `main` method as the beginning of execution. In keeping with the theme of fantasy leagues, the `play` method is used instead as the equivalent of `main`. Every **FLOOD** program must have a `play` method. **FLOOD** convention dictates the class containing `play` be named after the type of league being designed. For instance, in this example the type of league can clearly be inferred to be related to basketball:

Listing 2.2: BasketballPlay.fld

```
1  class BasketballPlay:
2      public play()
3      {
4          League myfbl = new League ("Happy League");
5          mybl.setMaxUsers(10);
6          mybl.setMinUsers(4);
7          ...
8          ...
9      }
```

Every **FLOOD** program must also instantiate a `League` object. The `League` object contains most of the other objects needed for the **FLOOD** program including `User`, `Player` and `Action` lists. Lists will be discussed in later sections but for now they can be thought of as Java LinkedLists (although there are subtle differences). For the minimum program we could have left the max and min user setting at their respective default value but as an example we set them above. The `League` object will later be passed to the GUI, however, here it provides the first example of extensibility within **FLOOD**. We could have written a new class that extends League to add functionality such as divisions within the league. For now the basic `League` object will suffice to create our first program.

A method in **FLOOD** is similar to a method in Java or function in C++. We call a method on an object by using the object's name followed by a period followed by the method of that object with an optional argument list in between mandatory parenthesis.

Listing 2.3: Setting the maximum number of users

```
1 mybl.setMaxUsers(10);
```

In this example we are calling `setMaxUsers` on our `mybl` object. The `mybl` object is an instantiation of the `League` class. The method `setMaxUsers` has been defined to take an argument of type `int`. We pass the value to 10 to the method which performs some action internally. In this case, it sets the maximum number of user in the league to 10.

Listing 2.4: Adding a new User

```
1 mybl.addUser(new User("Eli"));
```

We then begin adding users to the league. Once again we use this as an illustration of what can be added. We add five `User` and give each names. It would have been possible to leave this portion for later in the program.

Listing 2.5: Adding a new Action

```
1 mybl.addAction(new Action("Field Goal Attempt", -0.45);
```

Actions are added to the league using the `addAction` method. Actions consist of any

event that is associated with a point value, positive or negative, which are part of the evaluation of the strength of a `User`. The actions will be associated with a file uploaded each time-period which will contain player-action statistics. The statistics will be translated via this list of actions to points added to each `User`. The league will then know which *Users* are "winning" and rank them.

Listing 2.6: Launching the GUI

```
1 Flood.launchGui(blDraft, mybl);
```

The final part of the code launches the GUI and adds both the `League` and the `Draft`. 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** classes 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. The flow of the program will depend upon the user. 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.

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 on it, it requires to be declared at or before first usage. A declaration defines the properties of the variables. A declaration is of the form type name

followed by the list of variables to be declared as that type. This list needs to be comma separated such as:

Listing 2.7: User.fld

```
Class User:
   get int points;
   get str name, handle;
   setget int maxSize;
   list<Players> teamAthletes;

   addPlayer(Player athlete)
   {
      ...
   }

   removePlayer(Player athlete)
   {
      ....
   }
```

Here, the variables `name` and `handle` are both declared as type `str` (string). Notice that the variables are declared at the very beginning. Alternatively, as mentioned earlier, variables can be declared as and when needed. For example:

Listing 2.8: SnakeDraft.fld

```
Class SnakeDraft is Draft:
   private League game;

   public draftFunction(int turn)
   {
      return turn % game.teams.length();
   }

   /* Limit of 8 players per team, and there must be at least
      1 center, 2 guards and 2 forwards per team. */
   public bool pickPlayer(User team, Player athlete)
   {
      int centers = 0, guards = 0, forwards = 0, total = team.
         teamAthletes.size();

      if (total < 8)
      {
```

```
16            if (total < 4)
17            {
18               team.addPlayer(athlete);
19               return true;
20            }
21         }
22      }
```

Here, the integer variables `centers`, `guards`, `fowards` and total are declared right when they are needed. This as-needed method obviates the need to keep going back to the beginning of the code to declare variables, making declarations much more convenient.

FLOOD offers the set of arithmetic expressions required to create a comprehensive fantasy league of the user's choice. This set comprises of the standard addition, subtraction, multiplication, and division operators.

Listing 2.9: Arithmetic expressions

```
1  /* Limit of 8 players per team, and there must be at least 1
      center, 2 guards and 2 forwards per team. */
2  private bool evaluate(User u, Player p)
3  {
4     int centers = 0, guards = 0, forwards = 0, total = u.
         teamAthletes.size();
5
6     if (total < 8)
7     {
8        if (total < 4)
9        {
10          return true;
11       }
12
13       for person in u.teamAthletes
14       {
15          if (person.position == "center")
16             centers = centers + 1;
17          if (person.position == "guard")
18             guards = guards - centers;
19          if (person.position == "forward")
20             forwards = forwards / 2;
21          if (person.position == "mid")
22             mid++;
23          if (centers == 3 and p.position == "center")
```

```
24              return false;
25           if (forwards == 5 and p.position == "forward")
26              return false;
27           if (guards == 5 and p.position == "guard")
28              return false;
29
30           return true;
31        }
32
33        return false;
34     }
35  }
```

This 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 floats and integers.

2.3 Loops: For and While

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

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

Listing 2.10: While loop

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

The *while* loop operates as follows: The condition in parentheses is tested. If it is true (total is less than 4), the body of the loop (the four 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 that follows the loop. The body of a *while* can be one or more statements enclosed in braces, as above, or a single statement without braces, such as:

Listing 2.11: Single statement while loop

```
while (total < 4)
   centers = centers + 1;
```

The *for* statement is a loop and is a generalization of the *while*. **FLOOD** provides a easy to use *for* loop syntax that allows the user to iterate through any list, for example:

Listing 2.12: For loop

```
/* Limit of 8 players per team, and there must be at least 1
   center, 2 guards and 2 forwards per team. */
private bool evaluate(User u, Player p)
{
   int centers = 0, guards = 0, forwards = 0, total = u.
      teamAthletes.size();

   if (total < 8)
   {
      if (total < 4)
      {
         return true;
      }

      for person in u.teamAthletes
      {
         if (person.position == "center")
            centers++;
         if (person.position == "guard")
            guards++;
         if (person.position == "forward")
            forwards++;
         if (centers == 3 and p.position == "center")
            return false;
         if (forwards == 5 and p.position == "forward")
            return false;
         if (guards == 5 and p.position == "guard")
            return false;

         return true;
      }
```

```
30
31          return false;
32      }
33  }
```

Here, `u.teamAthletes` is a list of `person` and typically the user may need to iterate through this list. The *for* loop allows the user to iterate through this list conveniently without having to manually find out the length of the list or worry about the starting position of the list.

2.4 Conditionals

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

Listing 2.13: if conditional

```
1  if (person.position == "center")
2      centers++;
3  else
4      centers--;
```

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. In order to include multiple statements to be executed in case the condition is met, the '{...}' parenthesis pair can be used as follows:

Listing 2.14: Bracketed conditionals

```
1  if (person.position == "center")
2  {
3      centers++;
4      forwards++;
5  }
6  else
7  {
8      centers--;
9      forwards--;
10 }
```

If this condition is not met, then the statement or statements enclosed in the body of the else condition will be executed, i.e. decrement the value of the variables `centers` and `forwards`.

2.5 Functions, Arguments and Call by Value

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, depending on the access level of the function (`public` or `private`), perhaps anywhere else within the program, including from within other functions.

Listing 2.15: Syntax of a FLOOD function

```
1 private bool evaluate(User user, Player player)
2 {
3     ...
4 }
```

And similar to Java and C, all function arguments in **FLOOD** are passed "by value." In other words, the called function is passed the values of its arguments in temporary variables instead of the originals. This method stands in contrast to the pass "by reference" in which the function has access to the original argument as opposed to a local copy. Hence, in **FLOOD**, the called function can not and does not directly chances the variable in the calling function, but only its private, temporary copy.

2.6 Classes and Objects

Being an object oriented language, programming in **FLOOD** involves working with classes and objects. A class is the basic building block of **FLOOD**. An object is an instance of a class. The class decides the behavior and is essentially the template of the objects of that class. Classes in **FLOOD** need to begin with a capital letter, and are generally nouns whereas objects of a class need not begin with a capital letter.

Listing 2.16: Typical definition of a class in FLOOD

```
Class User:
    get int points;
    get str name;
    list<Players> teamAthletes;

    addPlayer(Player athlete) {...}
    removePlayer(Player athlete) {...}
```

For example, in the above code, `User` is a class and athlete is an object of `Player`. Classes are declared by the keyword `Class` followed by the name of the class the user wishes to assign. Objects are instantiated in the following manner by invoking either the default constructor of the class or any other overloaded constructor:

Listing 2.17: Object instantiantion

```
Player athlete = new Player();
```

Now, the object athlete is now an instance of the class `Player`. Now this object, athlete shares the same set of attributes as other objects of class `Player`, although, it might differ in the contents of those attributes.

2.7 Scope

FLOOD scope modifiers provide the ability to limit access to methods and fields as is warranted by the program. It enables encapsulation of data within classes and the ability to hide the data from outside access. **FLOOD** also provide novel approaches using `set`, `get`,

and `setget`. These modifiers default to private but generate setters and getters respective to their names. Public scope exposes a method or field to all outside access, and private scope completely hides a method or field from all outside access. Variables defined without modifiers exist and can only be accessed solely within the code block they are initially defined. For example, if an int is defined inside of a for loop, the int would only exist within the curly braces surrounding the looped code and does not exist outside of it.

2.8 FLOOD Base Classes

The **FLOOD** language has 5 base classes necessary for the creation of a fantasy league. They are described here in detail.

2.8.1 Action

`Action` is a class that encapsulates an event in a league along with an associated point value. These actions are then put into the league in order to distribute points among the users depending on how their players perform. For example, a basketball fantasy league will have the following actions along with its associated point values:

"Field Goal Attempt" ⇒ -0.45
"Field Goal Made" ⇒ 1.0

Thus, any player performing these actions will have the corresponding value added to the player's team. The base class has two instance variables as seen above with respective getters and setters but can be expanded as necessary.

Variables	
action	The action a player can make while playing.
points	The corresponding number of points awarded to a player doing the action in the League.

Functions	
Action(str a, int p)	This is the default constructor for the `Action` class. It takes a string and integer as input and sets them as the string representation of the action and the point value associated with that action.

2.8.2 Player

A `Player` is a unit tradable entity in the league. For example Michael Jordan is a `Player` in the Basketball league. This base class provides a set of data structures that entirely define a unit player, providing a group of setters/getters that allows the player name and position attributes to be set for each `Player` object. These setters and getters are not defined explicitly but rather automatically provided by the `setget` types of the variables.

Variables	
name	The name of the player.
position	The position the player plays.
Functions	
Player(str Name, str Position)	This is the default constructor for the `League` class. It takes two strings as input and sets it as the name of the player as well as his or her position.

2.8.3 User

A `User` is a person who play the fantasy league. Each `User` has a name, points and a list of *Players* associated with it. The list of *Players* is considered to be the User's team. There are a set of functions that can be used to add and remove players from a particular users team:

Variables	
points	Integer total point value of all the actions of the players on the team.
name	The team name.
list<Players> teamAthletes	List of players on the user's team.

Functions	
Player(str name)	This is the default constructor for the `Player` class. It takes a string as input and sets it as the name of the user.
addPlayer(Player athlete)	This function adds a player to the user's team.
removePlayer(Player athlete)	This function removes a player from the user's team.

which can be used in the following manner:

Listing 2.18: Adding a Player

```
/* Add a player to a team if possible* /
public bool pickPlayer(User u, Player p)
{
   bool addition;
   addition = evaluate(u,p);

   if (addition)
   {
      u.addPlayer(p);
      return true;
   }

   return false;
}
```

In the above code snippet, an object of class `Player`, `p`, is added to an object of class `User`, `u`, using the function `addPlayer` provided by the `User` class. Similarly, the `removePlayer` function can be used to remove `Player` objects from a `User` object.

2.8.4 Draft

A `Draft` is a class that specifies different rules and regulations regarding the addition, trade or drop of a `Player` from a user's team. The base class provides a very broad implementation that applies minimal restrictions but is fully expandable. Its methods include the following:

Listing 2.19: Draft methods

```
public int draftFunction(int turn);
public bool pickPlayer(User u, Player p);
```

```
3 public bool trade(User u1, Player p1, User u2, Player p2);
4 public dropPlayer(User, Player);
5 public playersLeft();
```

For example, when creating a draft for a basketball league, it is necessary that every team has at least 1 center, 2 forwards and 2 guards. Below is a possible function to enforce this, setting the max number of players per team to be 8. The evaluate function is a private method used to say if a team can afford to take the player trying to be drafted without violating the constraints concerning the number of each type of specific player.

Listing 2.20: Picking a Player

```
1  /* Add a player to a team if possible */
2  public bool pickPlayer(User u, Player p)
3  {
4     bool addition;
5     addition = evaluate(u,p);
6
7     if (addition)
8     {
9        u.addPlayer(p);
10       return true;
11    }
12
13    return false;
14 }
15
16 /* Limit of 8 players per team, and there must be at least 1
      center, 2 guards and 2 forwards per team. */
17 private bool evaluate(User u, Player p)
18 {
19    int centers = 0, guards = 0,forwards = 0, total = u.
         teamAthletes.size();
20
21    if (total < 8)
22    {
23       if (total < 4)
24       {
25          return true;
26       }
27
28       for person in u.teamAthletes
```

```
29          {
30              if (person.position == "center")
31                 centers++;
32              if (person.position == "guard")
33                 guards++;
34              if (person.position == "forward")
35                 forwards++;
36              if (centers == 3 and p.position == "center")
37                 return false;
38              if (forwards == 5 and p.position == "forward")
39                 return false;
40              if (guards == 5 and p.position == "guard")
41                 return false;
42
43              return true;
44          }
45
46          return false;
47      }
48  }
```

Variables	
game	Reference to the `League` object. Used to get the *Players* and *Users*.
Functions	
Draft(League game)	This is the default constructor for the `Draft` class. It takes a `League` object as input and stores it for use with other functions.
dropPlayer(User, Player)	This function drops a player from a user's team.
playersLeft()	This functions returns a list of Players left in free agency.
draftFunction(int turn)	This function returns the number of the user who's turn it is to pick the next player.
pickPlayer(User u, Player p)	This function adds a player to a user's team.
trade(User u1, Player p1, User u2, Player p2)	This function trades two players between two different users.

2.8.5 League

The `League` class is the pivot of all the base classes. It provides a set of variables and functions that are needed to comprehensively define any fantasy league in **FLOOD**. The variables and functions in this class are as follows:

Variables	
name	Name of the league (Basketball).
maxSize	Maximum number of players per user.
maxUser	Maximum number of Users allowed in the league.
minUser	Minimum number of teams.
list<User>	The list which stores all the Users registered with the league.
list<Player>	Complete list of players in the league.
list<Action>	Every possible action (point scoring) in the league.
Functions	
League(str Name)	This is the default constructor for the `League` class. It takes a String as input and sets it as the name of the League.
addAction(Action a)	This function takes `Action` objects as input and adds it to the calling `League` object.
addUser(User u)	This function takes a `User` object as input parameter and add adds it `User` objects to the `League` object.
addPlayer(Player p)	This function is used to add `Player` objects to the calling `League` object.
setMinUser(int MinUser)	There needs to be a programmer defined minimum number of users for the `League` to run. This function is a setter which sets value for a league.
int getMinUser()	To get the minimum number of users set for a `League` object, the programmer can make use of this getter.

loadPlayers(str textfile)	There's a `Player` list associated with each league. This list contains all the tradable objects of the league. For example, for a Basketball League, this list would comprise of all the players that the User can operate and choose from. (Michael Jordan, LeBron James, Kobe Bryant, Steve Nash). For a league of Elections, the player list would comprise of election candidates (Sarah Palin, Mitt Romney). The `loadPlayers` function loads this list of Players from the file passed as input parameter to the function.

3

Reference Manual

3.1 Introduction

The following is a brief Reference Manual for the **FLOOD** language. The reference manual is based on the K&R C manual which has for the most part inspired Java, the language **FLOOD** is based on. In certain instances, where there was no distinction between **FLOOD** and C, the C reference manual definition was used. Additionally, certain explanations were adapted from the Oracle (formerly Sun) online collection of tutorials.

Many parts of the manual should be familiar to the seasoned programmer. We were able to use **FLOOD** to experiment with ideas that have been born from our collective programming experience. The grammar will be used in the implementation of our compiler in the next step of the **FLOOD** project. The aim is to stay true to the this reference manual as much as possible, deviating only for extreme technical issues.

3.2 Lexical Conventions

The first phase in the interpretation of the source files is to do a low-level lexical transformation transforming every line to a series of tokens to be compiled in a later stage.

3.2.1 Tokens

There are 5 different types of tokens that exist: identifiers, keywords, string literals, operators and separators. White space is inherently ignored and is merely used as a means making code legible and will be removed accordingly before the source is tokenized, with the exception of certain instances of newlines as well as the separation of certain adjacent identifiers, keywords and constants.

3.2.2 Comments

The sequence of characters initially starting with /∗ and ending with ∗/ disqualify any of the surrounded characters from the lexical tokenization. Comments do not nest or occur within literals. There are no single-line comments however a single-line comment can be simulated using the above sequence on a single-line.

3.2.3 Identifiers

An identifier is a sequence of characters and digits beginning with a character. Underscore is also considered a letter. The language is case sensitive, but only the first 15 characters are significant in the unique representation of the identifier.

3.2.4 Keywords

The following words are reserved as keywords and may not be used as identifiers:

`if`	`public`	`set`	`int`	`new`	`true`
`else`	`private`	`get`	`str`	`is`	`false`
`for`	`class`	`setget`	`flt`	`return`	
`in`	`void`	`while`	`bool`	`list`	

3.2.5 Function Generation

Certain keywords will trigger function generation for commonly used functions. For instance, declaring an instance variable as `setget` will generate default setters and getters as well as declare that instance variable `private`.

3.2.6 String Literals

A character constant is a sequence of one or more characters enclosed in double quotes, as in "..." Character constants do not contain the ‘ character or newlines in order to represent them but rather use different escape characters. They are the following:

newline	NL	\n	backslash	\	\\
horizontal tab	HT	\t	double quote	“	\”

3.3 Syntax Notation

3.3.1 Expressions

Primary Expression

Identifiers, constants, strings, or expressions in parentheses.

```
identifier
constant
string
( expression )
```

Postfix Expressions

Operators in postfix expression group left to right.

```
postfix-expression:
     primary-expression
     list<type-specifier>
     list<postfix-expression>
```

```
        postfix-expression ( argument-expression-list )
        postfix-expression ++
        postfix-expression --

primary-expression:
        identifier
        constant
        ( expression )

argument-expression-list:
        assignment-expression
        argument-expression-list , assignment-expression
```

Function Calls

A function call is a postfix expression, followed by parentheses constraining a possibly empty, comma-separated list of assignment expressions, which are the arguments. The function declaration must include an access type (public or private and a return type before the identifier for the function. **FLOOD** allows the omitting of the void keyword without error. Omitting a return type automatically defaults to void.

```
type-specifier dataType foo()
```

An argument is an expression passed by a function call and a parameter is an input object received by a function definition and described by a function declaration. All arguments are passed by value. The function call fails if the number of arguments does not match the number of parameters, or the type of the arguments does not match the type of the parameters. The order of evaluations of arguments is from left to right, and recursive calls to any function are allowed.

Postfix Incrementation

A postfix expression followed by a '++' or a '−−' operator is another postfix expression. The value of the expression is the value of the operand. After the value is noted the operand is incremented ('++') or decremented ('−−') by 1 and assigned back to the expression. Note there is no Prefix Incrementation in **FLOOD**.

Unary Expression

Expressions with unary operators group right-to-left

```
unary-expression:
     postfix-expression
     unary-operator unary-expression

unary-operator: one of
     -   !
```

Unary Minus Operator

The operand of the unary − operator must have arithmetic type and the result is the negative of the operand.

Logical Negation

The operand of the ! operator must be of boolean type. The result is of the boolean type and has the value true if the operand is false, and the value false if the operand is true.

Multiplicative Expression

The multiplicative operators ∗, /, and % group left-to-right.

```
multiplicative-expression:
     unary-expression
     multiplicative-expression * unary-expression
     multiplicative-expression / unary-expression
     multiplicative-expression % unary-expression
```

The operands of ∗ and / must have arithmetic type and the operands of % must have integral type. The usual arithmetic conversions are performed on the operands, and predict the type of the result. The binary ∗ operator denotes multiplication, the binary / operator denotes division, and the % operator denotes the remainder of the division of the first operand by the second. If the second operand in either / or % is 0, the result is undefined.

Additive Expression

The additive operators + and − group left-to-right. The operands should be of arithmetic type, with one exception.

```
additive-expression:
     multiplicative-expression
     additive-expression + multiplicative-expression
     string-expression + string-expression
     additive-expression - multiplicative-expression

string-expression:
     string-expression
     string-literal
```

The result of the + operator is the sum of the operands. A string may also be added to another string. In this manner, concatenation is performed, where the result is a string containing the first operand and then the second beginning at the end of the first.

The result of the − operator is the difference of the operands. A string may not be subtracted from another string.

Relational Expression

The relational operators group left -to-right and returns booleans.

```
relational-expression:
     additive-expression
     relational-expression < additive-expression
     relational-expression > additive-expression
     relational-expression <= additive-expression
     relational-expression >= additive-expression
```

Equality Expression

```
equality-expression:
     relational-expression
     equality-expression == relational expression
     equality-expression != relational-expression
```

The equal to (==) and the not equal to (!=) operators are analogous to the relational operators except with lower precedence. (i.e. $a < b == c < d$ is true whenever $a < b$ and $c < d$ have the same truth value)

Logical *and* Expression

```
logical-and-expression:
     equality-expression
     logical-and-expression and equality-expression
```

The ***and*** operator groups left-to-right and returns true if both operands compared are true, and false otherwise.

Logical *or* Expression

```
logical-or-expression:
     logical-and-expression
     logical-or-expression or logical-and-expression
```

The ***or*** operator groups left-to-right and returns true if either operand compared is true, and false otherwise.

Assignment Expression

All assignment operators group right to left.

```
assignment-expression:
    logical-or-expression
    unary-expression assignment-operator assignment-expression

assignment-operator: one of
    =     *=     /=     %=     +=     -=
```

All require that the left operand be a modifiable expression. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place.

Comma Expression

All assignment operators group right to left.

```
expression:
     assignment-expression
     expression , assignment-expression
```

Comma expressions are used in lists of function arguments and lists of initializers in a parenthetical grouping that is evaluated left-to-right.

3.3.2 Declarations

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called definitions. Declarations have the form:

```
declaration:
     declaration-specifiers init-declarator-list;
```

The declaration in the init-declarator-list contain the identifiers being declared; the declaration-specifiers consist of a sequence of type and storage class specifiers.

```
declaration-specifiers:
     class-specifier declaration-specifiers
     type-specifier declaration-specifiers

init-declarator-list:
     init-declarator
     init-declarator-list , init-declarator

init-declarator:
     declarator
     declarator = initializer
```

Declarators contain the names being declared. A declaration must have at least one declarator and empty declarations are not permitted.

Type Specifiers

```
type-specifier:
     void
     int
     flt
     str
     list<type-specifier>
     object-specfier
```

Exactly one specifier may be given in a declaration. If the type-specifier is missing from a declaration, the declaration will result in an error.

All global variables (as well as function and class declarations) must also also be qualified, to indicate the access level of the objects being declared.

```
type-qualifier:
     public
     private
     set
     get
     setget
```

Type qualifiers may appear with any type or object specifier. A **public** variable or function can be accessed by classes other than the class in which it was declared, while a **private** one cannot. A **set** variable is declared as private and then a setter function is automatically generated for the variable. A **get** variable is declared as private and then a getter function is automatically generated for the variable. A **setget** variable is declared as private and then getter and setter functions are automatically generated for the variable.

Object Specifiers

An object is an instance of a ctlass. An object declaration has the form:

```
object-specifier:
     class-name
     class-name identifier
```

```
class-declaration:
     class class-name: [\n, \t] class-declaration-list
```

A class-declaration-list is a sequence of declarations for the functionality of the class.

```
class-declaration-list:
     class-declaration
     class-declaration-list class-declaration

class declaration:
     specifier-qualifier-list class-declarator-list;


specifier-qualifier-list:
     type-specifier specifier-qualifier-list
     type-qualifier specifier-qualifier-list

class-declarator-list:
     declarator
     class-declarator-list, declarator
```

A simple example of a class declaration is

Listing 3.1: Class declaration

```
1  class  Player:
2     private int foo;
3
4     public Player(int f)
5     {
6        foo = f;
7     }
8
9     public int getFoo()
10    {
11       return foo;
12    }
```

which consists of a constructor and a single function. Once this declaration is given, the declaration

Listing 3.2: Object declaration

```
1 Player p = new Player(10);
```

declares p to be an object of the given type. The keyword new here is used to create a Player object with constructor that must take in one parameter. With these declarations, the expression

Listing 3.3: Returns foo

```
1 p.getFoo()
```

returns the value of foo, which in this case is 10.

Declarators

Declarators have the syntax:

```
declarator:
     identifier
     ( declarator )
     declarator ( parameter-type-list )
     declarator ( identifier-list )

identifier-list:
     identifier
     identifier-list, identifier
```

List Declarators

A list declaration has the form

```
list<type-specifier> identifier
```

A list may be constructed from any type or object, or even from another list. All lists are mutable and no length need be predefined for it. Here is an example:

```
list<int> myList
```

which declares a list of integers.

Function Declarators

A function declaration has the form

```
type-qualifier type-specifier identifier ( parameter-type-list )
type-qualifier type-specifier identifier-list
```

The syntax of parameters is

```
parameter-type-list:
     parameter-list
     parameter-list-type , parameter-list

parameter-list:
     parameter-declaration
     parameter-list , parameter-declaration

parameter-declaration:
     declaration-specifiers declarator
```

Here is an example:

Listing 3.4: foo declaration

```
1 public int foo (int bar1, int bar2, str bar3)
```

The syntax of identifiers is

```
identifier-list:
     identifier
     identifier-list , identifier
```

Here is an example:

Listing 3.5: foo declaration

```
1 int foo, foobar, foobarbar;
```

Initialization

When a variable or object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and is either an expression, or a list of initializers nested in braces.

```
initializer:
     assignment-expression
     { initializer-list }

initializer-list:
     initializer
     initializer-list , initializer
```

The initializer for a list is a brace-enclosed list of initializers for its members. The list is mutable so it members can me added and removed at will.

3.3.3 Statements

Statements are executed in sequence and only for their effect; they do not have values. They fall into several groups.

```
statement:
     expression-statement
     compound-statement
     selection-statement
     iteration-statement
     jump-statement
```

Expression Statement

Expression statements have the form

```
expression-statement:
     expression;
```

Most expression statements are assignments of function calls. All effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement to place a label.

Compound Statement

So that several statements can be used where one is expected, the compound statement exists. The body of a function is a compound statement.

```
compound-statement:
     { declaration-list statement-list }

declaration-list:
     declaration
     declaration-list declaration

statement-list:
     statement
     statement-list statement
```

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended while the program is in the block, after which it resumes its force. An identifier may be declared only once in the same block. Initialization of automatic objects is performed each time the block is entered at the top, and proceeds in the order of the declarators.

Selection Statement

```
selection-statement:
     if (expression) statement
     if (expression) statement else statement
```

Both forms of the if statement, the expression, which must have boolean type, is evaluated, including all side effects, and if it evaluated to true, the first sub-statement is executed. In the second form, the second sub-statement is executed if the expression evaluates to false.

The else ambiguity is resolved by connecting an else with the last encountered if not yet matched with an else at the same block nesting level.

Iteration Statement

```
iteration-statement:
     while (expression) statement
     for (expression; expression; expression) statement
     for identifier in list statement
```

In the while statement, the sub-statement is executed repeatedly as long as the value of the expression remains true; the expression must have boolean type. The test of and all side effects from the expression occurs before each execution of the statement.

In the first for statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have boolean type; it is evaluated before each iteration, and if it becomes false, the for loop ends. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation.

In the second for statement, the statement is executed for every value in the list. This list can have any type and then identifier represents each member of the list in order. The loop ends when there are no more items in the list.

Jump Statement

```
jump-statement:
     return expression
```

A function returns to its caller by the return statement. When return is followed by an expression, the value of the expression is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears. Functions that return a void type may not have a return statement.

3.3.4 Scope and Linkage

The entire program does not have to be compiled all at once; the source text may be kept in several files containing translation units and precompiled routines can be loaded from libraries. Communication among the functions of a program may be carried out both through calls and through manipulation of the external database of the league data.

Therefore, there are two kinds of scope to consider: first, the lexical scope of an identifier which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects and functions with external linkage, which determines the connections between identifiers in other classes and libraries.

Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These items are: variables, objects and functions. Name spaces are governed entirely by the class, i.e. a class has one and only one name space.

The lexical scope of a variable, object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

Linkage

Within a translation unit, all declarations of the same object or function identifier with

internal linkage refer to the same thing; i.e. the object or function is unique to that translation unit. All declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

The first declaration for an identifier gives the identifier internal linkage if the private specifier is used, and external linkage if the public specifier is used. The declaration for an identifier within a block (and not in the global scope) may not include a specifier, and thus the identifier has no linkage and is unique to the function.

3.4 Object Orientated Programming

FLOOD is an objected oriented programming (OOP) language by design so in order to become an adept coder, a fundamental understanding of OOP concepts is necessary. To get started, in this section we review the four basic tenets of OOP:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3.4.1 Abstraction

In general, an abstraction is a model or an ideal—although not all of the details are present, the general parameters are, which then can be filled in with details. Furthermore, an abstraction is clear enough to tell one abstraction from another.

As a concrete example, say a software company has two job openings to fill, the first for a web designer and the second for a web programmer. In advertising for the position, the company would not describe a specific person to fill each position but instead in more general terms consisting of the skills and experience needed of each candidate to fill the

positions. Hence, the listings for the jobs will be two *abstractions* representing the two separate positions:

- Web Programmer
  - Experience programming in a team-based environment.
  - Experience with middleware and database programming.
  - OOP and Design Pattern programming skills.
- Web Designer
  - Experienced with creating web graphics.
  - Familiarity with animation graphics.
  - Experience with vector graphics.

Discerning the differences between the two positions and their broad requirements are straightforward enough, but the particular details are left relatively open-ended. For example, a programmer is unlikely to apply for the Designer position and a designer is just as unlikely to apply for the Programmer position. However, a pool of applicants could have a wide range of skills which would serve as the concrete details for each position—one applicant for the Programmer position may have Java middleware programming experience in addition to Oracle database skills while another may have experience in .NET and MS SQL Server. In this particular case, the *abstraction* is the Programmer job description and the details are filled in by each applicant's unique skill set and experience.

In *Object-Oriented Design with Applications* (Benjamin/Cummings), Grady Booch, a design pattern pioneer, provides the following definition of *abstraction* that is both clear and succinct:

> *An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.*

Booch's definition describes in a well-defined manner the two job descriptions. Each description provides the essential and necessary characteristics of the position while distinguishing from one another.

3.4.2 Encapsulation

In the context of OOP, encapsulation is often presented in terms of the *black box*, the notion in which the internal representation and inner workings of an object, for all intents and purposes, are hidden from outside view and inspection.

We encounter examples of *black boxes* in every day life. For example, the automobile. A car owner get into the driver's seat, inserts and turns his or her key, starts the engine, presses down on the gas pedal to accelerate and alternately the break pedal to decelerate and stop. These interactions between car and driver are enough to operate the vehicle but the majority of car owners do not know, nor usually care, about the exact science and engineering principles behind the multitude of technologies which actually make a car work. Automobiles are not transparent. They're black boxes.

Hence, an overriding benefit derived from the concept of a black box is that users need not be knowledgeable of nor worry about the myriad complexities involved in the inner workings of the box. End users just have to know how to interact with it, secure in the notion that whatever makes the black box works does so according to design.

3.4.3 Inheritance

The third key concept of OOP is inheritance. In the simplest terms, inheritance refers to how one class inherits the properties and methods of another class. If **Class A** has methods X(), Y(), and Z(), and **Class B** is a subclass of **Class A** (extends), it too will have methods X(), Y() and Z(). **Class B** is known as the *subclass* (or *derived* class) and **Class A** is the *superclass* (or *ancestor* class).

3.4.4 Polymorphism

The final important concept of OOP is polymorphism. Succinctly speaking, polymorphism as a software development paradigm is the ability to define a variable, method, or object that has more than one form. The intent of polymorphism is to implement a methodology of programming in which objects of varying types define a common interface of operations for the user.

Hence, these objects of different types then possess the ability to respond to method, fields, or property calls of the same name, each one according to an appropriate type-specific behavior. It is not necessary for the developer and the program itself to be made aware of *a priori* the exact type of the object and thus, the exact behavior is determined at run-time.

3.5 Grammar

Below is a recapitulation of the grammar that was given throughout the earlier part of this *Reference Manual*. It has the exact same content.

The grammar has undefined terminal symbols integer-constant, character-constant, floating-constant, identifier, and string; the bold style words and symbols are terminals given literally. This grammar can be transformed mechanically into input acceptable for an automatic parser-generator.

```
translation-unit:
     external-declaration
     translation-unit external-declaration

public-declaration:
     function-definition
     declaration

private-declaration:
     function-definition
     declaration
```

```
function-definition:
     declaration-specifiers declarator declaration-list
          compound-statement

declaration:
     declaration-specifiers init-declarator-list;

declaration-list:
     declaration
     declaration-list declaration

declaration-specifiers:
     type-specifier declaration-specifiers
     type-qualifier declaration-specifiers

type-specifier: one of
     void   int   flt   str

type-qualifier: one of
     public   private   set   get  setget

init-declarator-list:
     init-declarator
     init-declarator-list, init-declarator

init-declarator:
     declarator
     declarator = initializer

object-specifier:
     class-name identifier

class-declaration:
     class class-name: [\n, \t] class-declaration-list

class-declaration-list:
     class-declaration
     class-declaration-list class-declaration

class declaration:
     specifier-qualifier-list class-declarator-list;

specifier-qualifier-list:
     type-specifier specifier-qualifier-list
     type-qualifier specifier-qualifier-list
```

```
class-declarator-list:
     declarator
     class-declarator-list, declarator

declarator:
     identifier
     ( declarator )
     declarator ( parameter-type-list )
     declarator ( identifier-list )

type-qualifier-list:
     type-qualifier
     type-qualifier-list type-qualifier

parameter-type-list:
     parameter-list
     parameter-type-list , parameter-list

parameter-list:
     parameter-declaration
     parameter-list , parameter-declaration

parameter-declaration:
     declaration-specifiers declarator

identifier-list:
     identifier

initializer:
     assignment-expression
     { initializer-list }

initializer-list:
     initializer
     initializer-list , initializer

statement:
     expression-statement
     compound-statement
     selection-statement
     iteration-statement
     jump-statement

expression-statement:
```

```
     expression;

compound-statement:
     { declaration-list statement-list }

statement-list:
     statement
     statement-list statement

selection-statement:
     if (expression) statement
     if (expression) statement else statement

iteration-statement:
     while (expression) statement
     for (expression; expression; expression) statement
     for identifier in list statement

jump-statement:
     return expression;

expression:
     assignment-expression
     expression , assignment-expression

assignment-expression:
     logical-or-expression
     unary-expression assignment-operator assignment-expression

assignment-operator: one of
     =    *=    /=    %=    +=    -=

logical-or-expression:
     logical-and-expression
     logical-or-expression or logical-and-expression

logical-and-expression:
     equality-expression
     logical-and-expression and equality-expression

equality-expression:
     relational-expression
     equality-expression == relational-expression
     equality-expression != relational-expression
```

```
additive-expression
     relational-expression < additive-expression
     relational-expression > additive-expression
     relational-expression <= additive-expression
     relational-expression >= additive-expression

additive-expression:
     multiplicative-expression
     additive-expression + multiplicative-expression
     string-expression + string-expression
     additive-expression - multiplicative-expression

multiplicative-expression:
     multiplicative-expression * cast-expression
     multiplicative-expression / cast-expression
     multiplicative-expression % cast-expression

string-expression:
     string-expression
     string-literal

unary-expression:
     postfix expression
     unary-operator cast-expression

unary operator: one of
     -    !

postfix-expression:
     primary-expression
     list<type-specifier>
     list<postfix-expression>
     postfix-expression ( argument-expression-list )
     postfix-expression ++
     postfix-expression --

primary-expression:
     identifier
     constant
     string
     ( expression )

argument-expression-list:
     assignment-expression
     assignment-expression-list , assignment-expression
```

```
constant:
     integer-constant
     character-constant
     floating-constant
```