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. Fantasy gaming has become so entrenched in popular culture, there's even a American comedy sitcom called *The League* in which the main characters obsess over winning their fantasy football league.

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 straightforward 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. However, 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 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* 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, either 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
13 */
14 DefineFunctions
15    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. 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 time constraints, it was decided to reduce the scope of the language without losing the basic functionalities as originally promised. Therefore, the ability to add classes and inherit from pre-existing libraries was 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 an 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 the process of 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 $/*\ldots*/$ 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` get 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, Rebound, 7
Lebron James, Assist, 4
Carmelo Anthony, Steal, 5
Carmelo Anthony, Point Scored, 25

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

```
DefineLeague
   Set LeagueName("Basketball League");
   Add User("Anuj");
   Add User("Tam");
   Add Action("Field Goal", 2.0);
   Add Action("Rebound", 1.0);
   Add Player("Lebron James", "forward");
   Add Player("Kobe Bryant", "guard");
   Add Player("Dwight Howard", "center");
   Add Player("Kevin Durant", "forward");

DefineFunctions
 /* 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 name` where `type` is the data type of the variable and `name` is the identifier of the variable.

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, subtraction, 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, multiply and obtain the modulo of `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" 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

```
Bool evaluate(User u, Player p)
{
   ...
}
```

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

```
Bool function1()
{
   Return False;
}

...

Bool function2()
{
   Return function1();
}
```

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

```
1 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

```
1 Int i = 5;
2 draft(a[1]);
3 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`. For the full list of functions, see the *Reference Manual* section *Function Calls*. The syntax is as follows:

Listing 2.21: Some FLOOD utility functions and usage

```
1 Int i;
2 i = ArrayLength(p);  /* Where p is an array of Players */
3 AddPlayer(u, p); /* Where u is a User and p is a Player */
4 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

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.

We begin with a high level view of the structure of a **FLOOD** program. It is divided into two logical blocks. The program begins with the very first block which sets the attributes of the *League*. This block needs to be present and is specified as follows:

Listing 3.1: DefineLeague

```
1 /*
2 * Define the League and configure various settings
3 */
4 DefineLeague
5    /* Set League parameters */
```

```
6     Set LeagueName("The League Name");
7     ...
8     ...
9     Set MinTeamSize(5);
```

In this block, various attributes and properties of the *League* are set such as the name of the *League*, the list of *Users*, the list of *Players*, etc. We enumerate the list of possible statements allowed in this block below:

Listing 3.2: Setting league name

```
1 Set LeagueName("The League Name");
```

The above statement sets the name of the *League*.

Listing 3.3: Setting maximun number of Users

```
1 Set MaxUser(10);
```

Sets the a limit on the maximum number of *Users* of the *League*.

Listing 3.4: Setting minimum number of Users

```
1 Set MinUser(2);
```

Sets the a limit on the minimum number of *Users* of the *League*.

Listing 3.5: Setting maximum size of each team

```
1 Set MaxTeamSize(10);
```

Places a limit on the size of each *User's* team.

Listing 3.6: Setting minimum size of each team

```
1 Set MinTeamSize(2);
```

The minimum number of players that need to be in a *User's* team.

Listing 3.7: Adding a User

```
1 Add User("Name of User");
```

Adds a `User` to the *League*.

Listing 3.8: Adding an Action

```
Add Action("Name of Action", 10.0);
```

Adds an `Action` to the *League* and the number of points associated with it.

Listing 3.9: Adding a Player

```
Add Player("Name of Player", "Position");
```

Adds the `Player` to the *League* object, along with the position of that `Player`.

The next logical block is comprised of defining functions and invoking functions within them if needed. This segment needs to begin with the `DefineFunctions` keyword as follows:

Listing 3.10: Defining functions

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

    ...

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

The variables in a function have to be defined at the start of the function and a function needs to be defined before it can be invoked.

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 and new line are inherently ignored and are merely used as a means of making code legible.

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. Additionally, the language is case sensitive.

3.2.4 Keywords

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

`DefineLeague`	`DefineFunctions`	`LeagueName`	`Set`	`Add`
`MaxTeamSize`	`MinTeamSize`	`Action`	`Return`	`Str`
`MaxUser`	`MinUser`	`User`	`True`	`False`
`AddPlayer`	`RemovePlayer`	`Player`	`Void`	`While`
`GetUserName`	`GetNumPlayers`	`GetPlayerName`	`If`	`Else`
`GetPlayerPosition`	`GetPlayerPoints`	`ArrayLength`	`Int`	`Flt`

3.2.5 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 Type Specifiers

FLOOD supports the following datatypes:

- `Void`: Indicates absense of type information.
- `Flt`: A number with decimal values precise up to 6 decimal places.
- `Int`: An integer number.
- `Str`: A String of characters.
- `Bool`: True/ False values.

```
type-specifier:
     Void
     Flt
     Int
     Str
     Bool
```

3.4 Syntax Notation

3.4.1 Variable Declaration

Variables can be declared at the start of each function and must be declared before any statement. Variables can be either simply declared, or can be set to a value during definition as follows:

```
variable-declaration:
     type-specifier variable
     type-specifier variable = value
```

3.4.2 Expressions

Primary Expressions

Identifiers, constants, strings, or expressions in parentheses.

```
primary-expression:
   identifier
   constant
   string
   ( expression )
```

Arithmetic Expressions

FLOOD supports the arithmetic expressions of addition, subtraction, multiplication, division and modulus.

The addition (+) ,subtraction (−), multiplication (∗) and division (/) operators are left associative. The modulus (%) operator is non-associative.

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.

Multiplication and division can be performed only on `Int` and `Flt` types. Additionally, **FLOOD** does not enforce coercion of operand types. This means that any of the above operations can be performed only on operands of the same type. For example, an `Int` variable added to an `Int` variable.

Relational Expressions

The relational operators group left-to-right. The relational expression returns a boolean value.

```
relational-expression:
   variable < variable-or-constant
   variable > variable-or-constant
   variable <= variable-or-constant
   variable >= variable-or-constant
   variable == variable-or-constant
   variable != variable-or-constant
   (relational-expression)
```

The operators (==) and (!=) have lower precedence than the operators (<), (>), (<=), (>=).

Boolean Expressions

The three basic operators involved in boolean expressions are boolean *AND* (&&), boolean *OR* (||) and boolean *NOT* (!). The *AND* and *OR* operators group left-to-right. The *NOT* operator is right associative. The *AND* operator returns true if both operands compared are true, and false otherwise. The *OR* operator returns true if at least one operand is true, and false otherwise. The *NOT* operator returns true if the operand evaluates to false, and false otherwise.

```
boolean-expression:
   boolean-expression && boolean-expression
   boolean-expression || boolean-expression
   relational-expression && relational-expression
   relational-expression || relational-expression
   relational-expression && boolean-expression
   relational-expression || boolean-expression
   boolean-expression && relational-expression
   boolean-expression || relational-expression
   ! boolean-expression
   (boolean-expression)
```

3.4.3 Function Declaration

FLOOD allows the user to declare their own functions in addition to the default functions provided as follows:

```
function-declaration:
   returnType functionName (argumentList)
   {
       statements
   }
```

Now the function *functionName* can be invoked by any other function succeeding it in the program. It is important to note that the user needs to declare the function before invoking it. Another point worth noting is that all variables need to be declared at the start of the function declaration before they can be used.

3.4.4 Statements

Statements in **FLOOD** encompass conditionals, loops, assignments and function calls. Every statement needs to be succeeded by a semicolon.

Conditionals

A conditional statement in **FLOOD** is a statement which checks for a certain condition and processes the succeeding statements depending on the boolean value of the condition evaluated. The syntax is as follows:

```
If (expression)
{
    statements
};

If (expression)
{
    statements
}
```

```
Else
{
    statements
};
```

Here, *expression* can be a boolean expression or relational expression. If the value of the expression evaluates to `True`, then the statement is evaluated, otherwise control is passed to the next statement.

Loops

FLOOD provides the user with a looping structure in the form of the *while* loop. The syntax is as follows:

```
While (expression)
{
    statements
};
```

Here, *expression* can be either a relational expression or boolean expression. In the *while* body, *statements* is executed repeatedly as long as value of *expression* remains true. The *expression* must have boolean type.

Function Calls

The programmer can invoke functions at any point in the `DefineFunctions` scope as long as the function has been declared before the function call.

```
functionName(parameterList);
```

There are additionally a number of built-in functions the users can invoke without needing to define them:

```
AddPlayer(user, player);
```

The above function adds the `Player` object *`player`* to the `User` object *`user`*.

```
RemovePlayer(user, player);
```

The above function removes the `Player` object *`player`* to the `User` object *`user`*.

```
Int ArrayLength(user, player);
```

The above function takes as an argument either a `User` or `Player` *`array`* object as the parameter and returns the length of the array.

```
Str GetUserName(user);
```

The above retuns the name of the `User` object *`user`*.

```
Int GetNumPlayers(user);
```

The above returns the number of players of the `User` object *`user`*.

```
Str GetPlayerName(player);
```

The above returns the name of the `Player` object *`player`*.

```
Str GetPlayerPosition(player);
```

The above returns the postion of the `Player` object *`player`*.

```
Flt GetPlayerPoints(player);
```

The above returns the points of the `Player` object *`player`*.

Assignments

Values can be assigned to variables depending on their type. For example, a `Bool` variable can be assigned a value of either `True` or `False`. Similarly, an `Int` variable can be assigned any integer value.

Listing 3.11: Assignments

```
var1 = True; /* if var1 is of type Bool */
var2 = 1; /* if var2 is of type Int */
var3 = 1.0; /* if var3 is of type Flt */
var4 = "string"; /* iff var4 is of type Str */
```

Additionally, the return value of an invoked function can be assigned to a variable.

Listing 3.12: Function assignment

```
var5 = function(); /* if the return type of the function
   matches the data type of var5 */
```

3.5 Scope

The scope to be considered while programming in **FLOOD** is lexical scope of a variable. Variables can only exist within a function, and need to be defined at the start of the function body. Therefore, the scope of a variable is limited to the function in which it is defined. Global variables are not supported in **FLOOD**.

3.6 Grammar

Below is a recapitulation of the grammar that was given throughout the earlier parts of this *Reference Manual*.

program: definitions functions

definitions: DefineLeague definitionlist

```
definitionlist: definitionlist definitionproductions
     empty

definitionproductions:
     Set LeagueName ( string-constant )
     Set MaxUser ( int )
     Set MinUser ( int )
     Set MaxTeamSize ( int )
     Set MinteamSize ( int )
     Add User ( string-constant )
     Add Action ( string-constant, float )
     Add Action ( string-constant, -float )
     Add Player ( string-constant, string-constant )

functions: DefineFunctions functionProductions

functionProductions:
     functionProductions returnType functionName ( argumentLists )
                         { declarations statements returnProduction }
     functionProductions returnType functionName ( argumentLists )
                         { empty }
     functionProductions returnType functionName ( argumentLists )
                         { empty statements returnProduction }
     functionProductions returnType functionName ( argumentLists )
                         { declarations empty returnProduction }
     empty

returnType:
     Void
     Str
     Bool
     Int
     Flt

functionName: identifier

argumentLists:
     argumentLists , argumentList
     argumentList
     empty

argumentList:
     returnType identifier
     User [ ] identifier
     Player [ ] identifier
```

```
     User identifier
     Player identifier

statements:
     statements statement
     statement

statement:
     conditionals
     loop
     relational-expression
     assignment
     functionCall

returnProduction:
     Return identifier
     Return string-constant
     Return int
     Return float
     empty

conditionals:
     If ( relational-expression ) { statements }
     If ( relational-expression ) { statements } Else { statements }
     If ( relational-expression ) { empty }
     If ( relational-expression ) { empty } Else { empty }
     If ( relational-expression ) { statements } Else { empty }
     If ( relational-expression ) { empty } Else { statements }
     If ( boolean-expression ) { statements }
     If ( boolean-expression ) { statements } Else { statements }
     If ( boolean-expression ) { empty }
     If ( boolean-expression ) { empty } Else { empty }
     If ( boolean-expression ) { statements } Else { empty }
     If ( boolean-expression ) { empty } Else { statements }

loop:
     While ( relational-expression ) { statements }
     While ( relational-expression ) { empty }
     While ( boolean-expression ) { statements }
     While ( boolean-expression ) { empty }

declarations:
     declarations declaration
     declaration
```

```
declaration:
     Flt identifier
     Int identifier
     Bool identifier
     Str identifier
     Flt identifier = float
     Int identifier = integer
     Bool identifier = True
     Bool identifier = False
     Str identifier = string-constant

relational-expression:
     identifier <= constant-or-variable
     identifier >= constant-or-variable
     identifier != constant-or-variable
     identifier < constant-or-variable
     identifier > constant-or-variable
     identifier == constant-or-variable
     ( relational-expression )

boolean-expression:
     boolean-expression && boolean-expression
     boolean-expression || boolean-expression
     relational-expression && relational-expression
     relational-expression || relational-expression
     relational-expression && boolean-expression
     relational-expression || boolean-expression
     boolean-expression && relational-expression
     boolean-expression || relational-expression
     ( boolean-expression )
     ! boolean-expression
     identifier
     True
     False

constant-or-variable:
     float
     integer
     identifier

arithmetic-expression:
     arithmetic-expression + arithmetic-expression
     arithmetic-expression - arithmetic-expression
     arithmetic-expression * arithmetic-expression
     arithmetic-expression / arithmetic-expression
```

```
     arithmetic-expression % arithmetic-expression
     ( arithmetic-expression )
     identifier
     float
     integer

assignment: leftSide = rightSide

leftSide: identifier

rightSide:
     arithmetic-expression
     functionCall
     string-constant
     True
     False

functionCall:
     functionName ( parameterList )
     AddPlayer ( identifier , identifier )
     RemovePlayer ( identifier , identifier )
     ArrayLength ( identifier )
     GetUserName (identifier)
     GetNumPlayers (identifier)
     GetPlayerName (identifier)
     GetPlayerPosition (identifier)
     GetPlayerPoints (identifier)

parameterList:
     parameterList , parameterList
     identifier
     integer
     float
     string-constant
     identifier [ integer ]
     identifier [ identifier ]

empty:
```

4

Project Plan

4.1 Processes

Our team was a mix of three undergraduates, two Computer Science majors pursuing the *Applications* track and the *Vision and Graphics* track and a Computer Engineer, as well as two graduates, one pursuing the *Foundations* track and the other pursuing the *Systems* track. The team members hail from Vietnam, Nigeria, India, Israel and Switzerland. The team had backgrounds from professional industry, the military and international educational systems. As such, the team came in with very different ideas on development and team-work.

Our team decided early on that group working sessions and pair-programming would be an effective way to ensure constant communication among team members. We set out a schedule to meet every Monday night. Gradually we added Thursday night and then one weekend afternoon. Information was shared among team members through online documentation using Google Docs and a special Google Group alias. Every submission was drafted on Google Docs and every team member contributed. Additionally, the Product Manger's log and Bug Tracker were both available online and updated frequently. The code repository was hosted on GitHub.

Modules that were worked on by more than one team member would be pair-programmed

with one team member acting as the "driver" and one as the "observer." Additionally, the observer would act as a code reviewer which speed up development time. Working sessions were generally held in an open space where team members could work on their own sections but also communicate with each other in real-time.

Our team tried to ensure maximum development freedom for each member. There were no restrictions on development tools as long as it was tested and found consistent with the existing systems in place.

4.2 Roles and Responsibilities

Responsibilities for each team member were generally less defined than the roles listed below. Although a division is made, team members frequently contributed code and ideas to other parts of the product. Entire sections of code have evolved over the course of the project making it almost impossible to clearly discern the primary author.

- Elliot, *Product Manager* - Semantics and Error Checking.
- Anuj, *Langauge Guru* - Parser and Grammar.
- Tam, *System Architect* - Parser and Grammar.
- Dillen, *System Integrator* - Backend GUI and Error Checking.
- Stephanie, *System Tester* - Semantics and Unit Testing.

4.3 Style Sheet

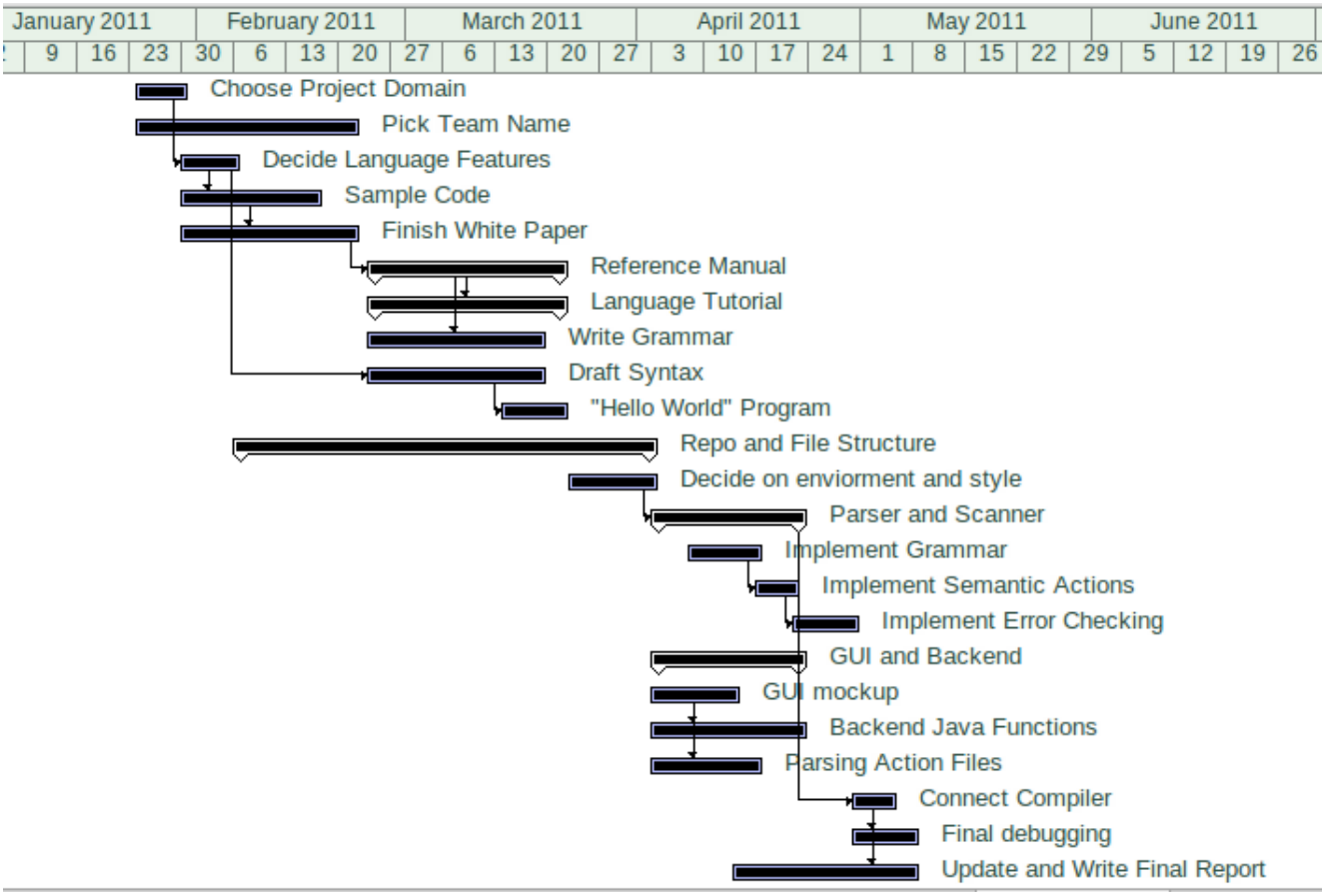
Recognizing that every team member has a unique style of coding, only very specific recommendations were made where differences would change readability. Small issues (such as where to put curly brackets) were left to the team member who wrote the specific section of code.

- Comments:
  - Every method must have a description above it.
  - Sophisticated functions must have comments within the body.
  - Comments should not include `TODO` messages.
- Bugs:
  - All bugs must be tracked using the Google Doc spreadsheet "Bug Tracker."
  - If a bug was found in code that was added to the repo, the team member who found the bug must notify all team members by email.
  - When a bug is corrected, all unit tests regarding the specific section must be re-run to assure that no new code takes away from existing functionality.
- Repository:
  - Check in code to the repository after every major change.
  - Check in code after every session and every four hours within one session.
  - Do not check code in unless it has been tested and a unit test has been written for the functionality added.
  - Update from the repository before a commit to minimize conflicts.
- Code Style:
  - Variable and Function names should be clear and understandable.
  - White Space should be used liberally in order to make code easily readable.
  - The semantic checks exist with the semantic object. The backend runs through GUI. The grammar must treat both as a blackbox.

- Java naming conventions are the preferred method for style (including but not limited to camel-case, indentation and programming practices) as appears in *Code Conventions for the Java Programming Language, Revised April 20, 1999.*

- Modules:
  - Every module should exist within its own Java file and ideally its own folder.
  - Unit tests should exist with the folder that includes the makefile and given names indicative of its use.

4.4 Project Timeline



4.5 Meeting Log

Meeting	Date	Minutes
1	Jan 24th	Brainstormed ideas for programming language. An idea that was discussed: we give users the option of using simple statements to design say maybe a Rubik cube game with a high degree of customisation; not only can the sides of the cubes be coloured, we can leave it to the imagination of the users to say, maybe use texts on the smaller cubes and come up with a creative game involving the Rubik cube or say even a sphere.
2	Jan 26th	More ideas discussed: The Programmer gets to upload the instrumental, type the lyrics in the code/upload the lyrics as well. They get to choose speed of highlighting for any given interval. The onus of syncing it with the corresponding part in the music is on the user. Another idea: Just graphics and the ability to build complex games from shapes. You could have enough power to build tower defense, angry birds, asteroids etc.. without the fancy pictures. Just with pure shapes.
3	Jan 30th	Settled on some sort of Fantasy League Game. Some questions discussed: Is there a problem with connecting to the Database? What kind of data-types? Will it be object-oriented? How extensible should our language be? How should we design the syntax? Started draft of white paper.
4	Feb 6th	Continued Working on white-paper. Agreed that language should contain OO and be similar to java but take elements from Python. Created Git Repo. Created team google-group.
5	Feb 9th	Agreed on Language Breakdown A FLOOD program can: • Create a League and define attributes such as Team Size. • Establish Rules for play using the existing libraries or overriding them. • Define features of the league such as draft and trade functions. • Connect the league to a database of the user's choice. • Trigger the build for the front end GUI and deploy the program to users.
6	Feb 13th	Continuing work on white-paper. Sketch of what a program would look like. Settled on possible name for project: FLOOD First Commit to new repo.
7	Feb 17th	Discussing feedback from language brainstorming session.

Meeting	Date	Minutes
8	Feb 21th	Final Meeting to finish white-paper. Finalized language features and highlights, data-types, keywords and control-flow. Discussed possible application of a FLOOD program.
9	Feb 23rd	Submitted White Paper.
10	Mar 5th	Discussed Feedback from White-Paper. Where is our computation? How do we want to implement OO?
11	Mar 14th	All day session to work on RM and LT.
12	Mar 17th	All day session to work on RM and LT.
13	Mar 19th	Finishing the grammar, continuing the reference manual. Updated the Language Tutorial to reflect changes in grammar.
14	Mar 23rd	Submitted LT and RM.
15	Apr 3rd	Discussed ways to implement OO and whether to alter to the language and simplify it. Discuss times for final presentation.
16	Apr 8th	Feedback on LT and RM. Beginning to scale down the language to make it easier to a parser and scanner.
17	Apr 10th	Project File Structure finalised and committed to repo. Division of labor and initial deadlines drawn up. First files committed to repo. Java and FLOOD files used for testing created. Meeting Times for the rest of the project: Monday 6:15 pm, Thursday 6:15 pm. One weekend day to be chosen on Thursday 1:00 pm - Longer meeting to work (in clic lab).
18	Apr 11th	Revamped Syntax and starting sketching GUI mockups.
19	Apr 14th	Working meeting in Clic lab (see commit comments).
20	Apr 21th	Progress Report. Basic FLOOD file can be parsed.
21	Apr 26th	Working Meeting in Clic lab (see commit comments).
22	Apr 28th	Working Meeting in Clic lab (see commit comments).
23	May 2nd	Working Meeting in Clic lab (see commit comments).
24	May 4th	Putting the final product together. Finished updating the reference manual and language tutorial. Error Checking in place. File FLOOD program can be compiled.
25	May 6th	Last minute debugging. Adding more error productions. Last run-through of every unit test with whole team present.
26	May 8th	Preparing Presentation. Looking over Final Report last time.

5

Language Evolution

At the onset of our language development, we decided to provide the user with as much flexibility as possible. With this as the main focus, our language was aimed at providing object-oriented development to the users. Bearing this in mind, we began developing our language in an incremental manner. We first decided on what the output of the front-end should be, based on what the back-end would expect. In short, we settled on the rendezvous point for the front-end and back-end and worked our way towards it.

As we deliberated on what this intermediate Java code should be, we realized it would be sufficient if the back-end is object-oriented. We therefore decided that the onus of object-oriented programming should not be on the programmer, keeping in mind that users of **FLOOD** will most likely not be familiar with object-oriented concepts. Moreover, we concluded that given the nature of the domain, the programming language being object-oriented would make the learning curve steeper for the user with not much added value. Fully convinced along these lines and the ease-of-use for the programmer in mind, we decided to do away with the object-oriented paradigm.

With the face of the language now modified to provide the user with as much flexibility as needed, yet within the domain of the language as well as keeping it simple to program, we set out to decide the logical division of the program code to aid the user in programming. We

carefully analysed fantasy leagues and realized there exists a common denominator between all fantasy leagues. We decided to push this common denominator to the back-end, leaving the user with the bare minimum yet retain full flexibility to design a fantasy league with the rules of his/her choice.

The language now was developed with a logical division of the program the developer would have to write. The first would need to be the definition of the league; the programmer would need to specify details of the number of users, the names of the users, the players and their scores, valid actions and points associated with these valid actions. This allows the user to define rather quickly the parameters and setup the league.

The next logical block we decided on was defining functions and invoking them within other functions. This section needs to begin with the keyword `DefineFunction`. There are four default functions that each league needs to have which are: `draftPlayer`, `drop`, `trade`, `draftFunction`. The user can override these functions, although the user can choose not to define these functions, and the default function definitions for these functions will be invoked.

Over and above these four functions, the user can define custom functions and invoke them. In its first incarnation, functions could take any number of parameters of the following types: `Bool`, `Str`, `Int`, `Flt`. Variable declarations were allowed at the beginning of the function before any other statements. The language at this point also provided for looping structures, arithmetic expressions, relational expressions as well as conditionals. The back-end required array constructs of objects of a couple of back-end classes, namely the `User` and `Player` classes. Since the programmer does not know the contents of the `User` and `Player` classes, we decided to introduce array constructs in the formal parameters and actual parameter list of functions as follows: functionName(User[] u,Player[] p) statements In this manner, as and when changes were made in the grammar, they were incorporated in the back-end, and as and when changes were made to the back-end, they were incorporated in the grammar. The evolution of the language thus involved the grammar and the back-end

evolving in tandem and incrementally.

6

Language Architecture

7

Development Enviornment

During the development of the FLOOD language, development was spread across two operating systems. The majority of the group developed with Ubuntu (10.04/10.10/11.04) while one other worked exclusively in Windows which ended up working out rather smoothly despite some issues setting up the Git repo on Windows and maintaining multiple makefiles. Yet this cross system development also ensures that our language is not system dependent. In order to maintain a current version of all parts of the language, our group set up an online repository on Github where changes could be made at any time at the programmers convenience.

The majority of java code for the source files used in the intermediate code compilation level was written nearly exclusively in Eclipse 3.6 Helios. The majority of the backend was written from scratch with a select few data structures to promote efficiency. Those libraries included primarily the Java APIs HashMap and ArrayList classes when necessary so that looking up references to various Users, Players and Actions would all be O(1). They were also used in a manner to prevent duplicate copies of any particular instance of class so that references (values of references since Java is pass-by-value) were used instead.

In addition to these source files, a GUI was also created to allow user interaction with the program during runtime, a key component of any fantasy league. The development

of the GUI was restricted to purely Javas Swing and AWT library components with the framework created using Googles WindowBuilder Pro web toolkit. With the assistance of the development environment provided by that plugin to Eclipse, the GUIs rough layout was generated with ease but was completely devoid of any functionality besides its good looks. The remainder of the GUI was developed by hand adding various functionalities to different components and promoting ease of integration with files generated by the compiler. No other outside sources were used in the remainder of the production of the source files that integrate with the compiler output besides the occasional Google search on how some feature of Swing truly worked, disregarding ones intuition. The final features added to the GUI were the ability for the program to write its current state to a file that could be saved in a directory and then imported at a later date, so that a fantasy league could be easily maintained over the course of any sports season without leaving a program running for the entire duration.

The parser and grammar were implemented with JFlex and BYacc respectively. These are both the Java versions of Lex and Yacc that the language FLOOD compiles to in its intermediate code generation. The majority of the development of the parser and grammar were done in a combination of Gedit and terminal along with some good ol fashioned paper and pencil. Paper and pencil is really the best (and almost only) way to really work out and recognize shift/reduce conflicts which inevitably helped to generally understand their origins and prevent them. The only exclusion to this trend was the development of the semantic actions that enforced type checking and other semantic checks, such as determining if functions and variables had been previously declared, which were primarily developed in Eclipse 3.6 Helios. This was done due to the fact that it was exclusively written in Java in which instantaneous compilation helped with simple syntax errors that may have been overlooked.

The most important tool, however, used in the joint production of FLOOD was indubitably Google Docs. The ability to jointly work on the White Paper/Language Tutori-

al/Reference Manual and watch them come together smoothly as opposed to very disjoint and extremely differently written components was truly invaluable. The ability to share and jointly edit others documents such as schedules, bug reports and other files simultaneously without the burden of constantly pulling and pushing vastly improved efficiency especially when every group member can just consult a checklist to see what components were still incomplete and most urgent. Additionally, the real time updating of files provided a great base for the conversion to Latex of the final reports. It was a crucial component in the merging of everyones work throughout the process.

8

Test Plan

In the early stages of developing the grammar for FLOOD most testing simply followed this scheme:

-
-
-
-

9

Conclusions

9.1 Lessons Learned as a Team

We feel that as a team we have a better understanding of the magnitude of large programming projects. We learned that initial ideas are seldom translated into the final product. When the project first began, we had intended to write a very large language with many features. At the time, none of us truly understood the work that went into implementing even a small language. As we progressed in the semester, the difficulty of the task became more apparent and we had to change our entire language. We feel that we learned a great deal about the more complex aspects of programming languages specifically because we were not able to implement what we had originally intended. Trying to brainstorm features like inheritance and polymorphism taught us about how those components work in popular programming languages.

We learned that teamwork doesnt mean everyone does their own part. It means that people help out and dont stay contained within their respective responsibilities. It means that if someone is having an off-week and/or has an exam the next day, other team members can carry the weight that meeting. We learned that work-schedules need to be implemented early and ambitiously so that deadlines can be stretched within reason if needed.

We each learned different ideas on programming and development from our teammates. Every session of programming there were arguments of proper programming practices or the design of the language. Team members contributed ideas to our language from a wealth of programming experience from languages such as Java, C++, Python and Pascal. We learned that that every team member has particular strengths and those should be utilized as often as possible.

9.2 Lessons Learned by Each Team Members

9.2.1 Stephanie

The greatest lesson I learned from this assignment was to start small and work your way up from there. Designing a project that implements something you have no idea how to do is extremely difficult. It is all too easy to fall into a very complicated design when the process for creating the pieces is not yet known. In the beginning, my team and I were rather ambitious and naively so. We had dreams of a object oriented language complete with inheritance, polymorphism, and encapsulation. We even wanted to expand the language onto the Android platform if we had time. Somehow we had forgotten what college was: four years of spending every minute of every hour wishing you had more time. We soon realized the we were in way over our heads, and that there was only so much weight Java would carry for us. And so, we had to significantly modify our approach to this project. We first started by simply making cuts to the language, but that left a convoluted and disorganized language. So instead we began anew. We identified all the bare minimum requirements needed for a fully functional language for generating fantasy leagues. From there we could build a base language and then add to that. This process takes away depressing decisions to remove ideas, and replaces it with the exciting realization that we can add things.

Projects of this magnitude need to be constantly worked on. You cant simply wait until a week before a deadline, put in endless hours of work and expect that to be sufficient. When

a project spans four months work must be put into it every week of every month. By getting into lulls where no work is being done on the language, you risk losing focus and generating a bit of a learning curve, if you will, in order to return to the language. Right from the beginning, we made sure to meet regularly, regardless of how much we needed to do. There needed to, at the very least, be a constant reminder of the project, no time for the language to be forgotten.

While at times it may appear to be rather difficult, it is important to always be aware of what each team member is working on. Not in the sense that you know specifically what they are doing every minute they are working, but just where they are. Indeed, that is primarily the job of the project manager, but I found that it is rather helpful for each member of the team to communicate with the team what their next steps were and how they were going to go about completing their goals. This way there is minimal disconnect between the various parts of the language, especially between semantics and the grammar, where changes in one greatly affect the other. Our group maintained this awareness with the use of Github, Google Docs, and a Google group, in addition to our frequent team meetings and coding sessions. This way, all changes were documented and spread throughout the entire group.

The project is a not a group of people designing pieces of a puzzle that will fit together, after quite a bit of massaging and molding, to produced a result. It is a team building block after block, each block being guided onto the last, until the structure is complete. The idea to split the group into 5 specific roles helps with ensuring that everyone has a part to do, however I find that this can actually create a disconnect. Instead of creating a strict line on who does what, I feel that having each person contribute to each part of the language builds a more continuous final product. Moreover, it makes the process of merging the parts of the language significantly more efficient because each member of the teams understands the language and many errors are avoided or resolved quickly by simply just knowing what to look for. In summary it creates a cohesive team, with a thorough understanding of the language.

Commitment is a vital key to any successful team project, and for a project that has lasted as long as this, it is all too easy to feel overwhelmed. Over the course of the semester fluctuations in ones workload and personal life can take quite a toll on ones ability to function in a group. However, it is in these times that the true power of the group shines through. In the first couple weeks after I fractured my hand, the pain was enough to inhibit my ability to type. In that same time, I had a web site to build, an essay to write, and coding sessions for this project. Had it not been for my team, I would have likely given in to the stress and crumpled under the workload. However, my determination to not let my teammates down allowed me to continue pushing forward. This level of commitment can develop in a team regardless of whether or not the team member knew each other before the formation of the team. When the members of FLOOD were established, I only knew half of my teammates. However, in a team of only 5, it is hard to remain cliquish and as a result the team developed rather strongly. Throughout the course, I've have become rather close friends with my team members and will look forward to taking classes with them in the future.

9.2.2 Elliot

While this is not my first time as a project leader, it is my first time leading a software development project. The most important lesson Ive learned is the need for constant communication among the team. At times though not often, information would be bottle-necked to a certain person and other team members would not be informed of important changes. The best way to deal with this situation is transparent documentation and meetings. Ive also learned how difficult it can be for each team members unique coding and work style to be integrated into a cohesive unit. For instance, our version control choice was unfamiliar to some team members at first and it took some time for everyone to get used to working with it.

A project of this magnitude can easily overwhelm any group no matter how experienced. The main goal should be constant pressure. Constant pressure does not mean stress but

rather always making progress regardless of the difficulty or clarity of the task. Tasks should be constantly broken down into smaller pieces and distributed as evenly as possible. When a task is assigned, it shouldnt be assumed that the team member will be able to complete it but rather teammates should look out for each other and help when needed. The role of a product manager is not to dictate and assign tasks as they please. Every team member should be given the opportunity to pick specific tasks that they find most interesting. Tasks that were deemed uninteresting were distributed evenly so that no one took more than their fair share of grunt work. Decisions should not only be democratic but accepted by the whole team. If even one team member didnt like an idea or an implementation detail, the team worked hard to come up with an alternative that was acceptable to everyone. A product manager should also keep accurate records and have awareness of every moving piece in the project. Ultimately, as the saying goes, the "The Buck Stops Here". This is a lesson every good product manager must learn and usually the hard way.

On a personal note, I feel comfortable saying I understand how a compiler works on a very applied level. I still dont feel comfortable with much of the theory that goes into compilers and in a way it has given me a healthy respect for theorists who have paved the way for modern programming languages. I learned to appreciate the intricacies of programming languages and notice subtle differences that were not obvious to me beforehand. I have already seen a difference in my code production in other projects. In my opinion, one of the most fascinating aspects of software development is the relationship among the developers. Working with other people, especially in an academic setting, is always challenging but it can also be very rewarding. I truly enjoyed the time spent working with my teammates and regardless of the final outcome of this project or the class in general, I feel that I have learned a great deal from them.

9.2.3 Tam

9.2.4 Dillen

1. Choose your group members wisely Choosing the right people for a group is beyond the most important part of an assignment. Although it may seem unclear what everyones exact roles may be for a given project, be sure to pick people who are passionate and have deep interest in different components as well as the overall idea of a project. Nothing can be more detrimental to a group that to have members who are unenthusiastic or not willing to pull their own weight, not due to a lack of ability (because lets be honest, no one knew how a compiler worked when we first started) but rather a lack of motivation. Our group functioned extremely well because we all split up the parts into relatively equal work loads and when one part overflowed due to unforeseen complexities, those with easier tasks jumped in to help. Another key part of our success was the general interest in our idea, which became my next lesson learned.

2. Thoroughly develop your idea, and then do it again It is important to spend adequate time to thoroughly develop an idea and delve past the surface challenges and really research potential dead-ends. Unlike most groups (I believe), we met every week from the moment our group was formed until the end of the semester, even when we all lacked any detailed view of how to complete the task. The first several meeting were literally 4+ hour long discussions on potential programming languages where everyone would express interest, play devils advocate and expand on ideas. The most difficult part for us was to propose ideas feasible for a semester long project without simply creating a markup language. This was crucial in creating a thorough blueprint for our language and the writing of the White Paper/Language Tutorial/Reference Manual.

3. Dont reinvent the wheel One of the most valuable parts of any programming project is to utilize tools provided online to ease the pain. Development tools such as WindowBuilder Pro for the GUI, BYacc and JFlex for the grammar and parser, Github for the online

repository and Google Docs for the written portions provided greater ease and efficiency during the entire process. USE THEM!

4. Pick a good project manager who can bring everyone together It is easy to get lost in time when everyone is specializing on a specific part of a project and its the product managers job to keep bringing everyone together and making sure everything is running smoothly and no individual is lagging behind. Collaboration and communication are absolutely key in the software development world because otherwise, come submission time, everyone has a bunch of extremely well developed but independent portions of the project with integration being nearly impossible. I noticed that our group was exceptionally productive when we all worked in one long line in the CLIC lab, each with the Google Bug Report Document constantly asking questions, finding bugs and delegating work with various priority. I realize it is difficult for a bunch of college students to find time to all meet and work for a few hours but those hours spent working together are far more valuable than any work you will do by yourself.

9.2.5 Anuj

Grammar design is not a walk in the park Designing the grammar for a language is not easy. Designing a small grammar for small units, like say, just arithmetic expressions, is easy. As the grammar evolves, the complexity increases linearly. It helps to have a clear idea of what the language will look like, and stub out a rough design of the grammar first. This exposes the potential shift/reduce and reduce/reduce errors right from the beginning. It also helps to understand fully how these errors occur, and how to eliminate them.

Decide the scope of the language/project realistically One of the most important lessons I learnt from this project is to do a strong feasibility study, before proposing a project. A thorough study should be performed to roughly estimate what problem statement the project proposes to tackle. This should neither be too ambitious, nor too trivial. Getting the right balance is a tricky task, and it seems safer to project the latter, and extend the

scope of the project if time permits. Another important lesson I learnt is never to commit any major feature in the name of the project, unless absolutely certain that the feature will be implemented.

Working in a group To begin with, I had an interesting time working with my group. It not only made working on a project of this magnitude seem effortless, but I also learned a lot from my group on various fronts. This made me realise how important team members are, and the dynamics of the group, can make or break a project. While it may seem that more hands on deck means less work per person, I also realised that it requires good management on the part of the project manager to coordinate and integrate the project well; something I was witness to during the course of this project.

9.3 Advice for Future Teams

- Never promise features of your programming language in its name. We tried very hard to come up with a cool name that described our language. The name is still cool but it doesnt describe our language anymore.

- Start early and meet often. It may seem futile to meet when you have nothing to discuss but just by being in the same room you may start thinking of an idea that changes your project.

- Ask for help no matter how small of an issue. We utilized our TA, Hemanth, as much as possible and we are truly indebted to him for all his valuable advice.

- Dont be scared of unfamiliar technology/Free software is your friend. Our entire project was written and shared in Google Docs, the code will forever be saved on GitHub and the documentation lives beautifully in Latex.

- Revisit your idea as much as possible. Our language has morphed from an Object-Oriented language with Java-like features into a smaller more compact version. If we

had decided to go-ahead with our original idea we would not have finished our project or it would have been poorly implemented.

- Pair Programming may seem like a waste of time but it dramatically reduces net production time. Most of the code produced was written at one work station with two team members working at a time. Often code that seemed very difficult to one programmer was trivial to another. When a programmer was tired, they could switch and development continued.

- Pick a project that is interesting to you rather than trying to impress anyone. Every team member was truly interested in our project which made brainstorming more fun and productive.

- A team that eats together works better. Its better to work on a full stomach and its more fun to brainstorm over dinner. We personally recommend the Dominoes 7-7-7 deal.

9.4 Suggestions for Future Improvement

Its very difficult to write this section as a team since each team member had different topics that they enjoyed. We all agree that the project was the highlight of the course and very much enjoyed working on it. Some of us feel that the classes were a little theory-heavy in what was originally expected to be a more applied course in compilers. Some of the theory, especially the later topics such as code-generation and optimization as well as data-flow analysis might have been better served with more homework and possibly programming exercises. The first homework gave everyone a taste of Flex and Yacc which helped both in understanding the theory and what was expected later in the project. Those of us who had taken Computer Science Theory enjoyed seeing how the theory we had previously learned applied in the real-world. We all feel that this was a difficult class but in the end very

rewarding.

Appendix A

FLOOD Source Code